TRƯỜNG ĐẠI HỌC FPT

Neural Ordinary Differential Equations Network and its Extensions

Graduation Thesis Final Report

Luong Thuy Chung
Nguyen Nhat Mai

A thesis submitted in part fulfillment of
the degree of BSc. in Computer Science with
the supervision and moderation of **Dr. Vu Khac Ky**.

Bachelor of Computer Science
Hoa Lac campus - FPT University
September 21, 2020

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Mitchell (2006) defined that Machine Learning seeks to answer the question "How can we build computer systems that automatically improve with experience, and what are the fundamental laws that govern all learning processes?". Deep learning is a sub-domain of machine learning that involves algorithms inspired by the structure and function of the brain known as artificial neural networks. Like information processing taking place in the brain, information from input data is passed from neuron to neuron, layer of neurons to layer of neurons. How the depth of neural networks affects learning ability has been studied extensively by scientists. However, the deeper the network, the more difficult it becomes to learn. This partly prevented the development of deep learning.

The advent of the residual network (ResNets) He et al. (2016) has brought a huge step forward in the development of deep learning, as the number of classes has increased dramatically. Resnets has transformed deep learning into a new anecdote with results that outperformed older models. However, the question arises, what if the Resnets layer count reaches infinity.

In 2018, Chen et al. (2018) launched the Neural Ordinary Differential network - a new family of neural networks, which answers the above question. Neural ODEs-Net is a combination of modern neural network and numerical methods by using available ODE solvers. With the development of ODE solvers over 120 years, Neural ODEs-Net has achieved many initial successes. In the scope of this thesis we will present Neural ODEs-Net, strengths, weaknesses, architecture and its extensions.

1.2 Outline

Sequentially, we will go from the most basic of neural networks and numerical methods to the Neural Ordinary Differential Equations Network and its extensions. Our thesis contains 5 chapters:

Chapter 1: It is a general introduction about Neural ODEs-Net, the scope of this thesis.

Chapter 2: We provide a detailed overview of the background of ordinary differential equations and neural networks. In this chapter, we present about initial-value problem, some numerical methods, architecture of a simple neural network, and its problems.

Chapter 3: We present about Neural ODEs-Net with its learning process and its implementation. The forward and backward propagation will be elucidated. Chapter 2 also includes its architecture for a supervised problem. Defining and implementing Neural ODEs-Net brings some strong benefits of effective memory and adaptive computing with using the adjoint sensitive method for computing gradients.

Chapter 4: It is straightforward to understand the properties of Neural ODEs-Net. From those properties, the limitation of using Neural ODEs-Net is pointed out in the practice. At the same time, we also present some models to improve the representation ability of Neural ODEs-Net. Because any performance deficiency will obviates the need for using Neural ODEs-Net.

Chapter 5: It includes experimental results of both the original Neural ODEs-Net and its extensions in the CIFAR10 dataset. We will compare their accuracy and judge about its representation ability.

1.3 Related Works

The idea of inter-presenting ResNets as approximate ODE solvers was previously spurred by researches about reversibility and approximate computation in ResNets. They were proposed by Chang et al. (2017) and Lu et al. (2018). LeCun et al. (1988) and Pearlmutter (1995) presented the use of the adjoint sensitive method for training neural networks, but they did not demonstrate it practically. Taking advantage of the power of modern ODE solvers, Chen et al. (2018) investigated Neural ODEs-Net with the same properties of previous researches in more generality by directly using ODE solvers.

Chapter 2

Background

2.1 Ordinary Differential Equations

The term *Differential Equation* is begun by Leibniz, the Bernoulli brothers, and others from the mid 17^{th} century. It became a branch of Mathematics, both in pure math and applied math (Archibald et al., 2005). Differential Equation plays a prominent role in many fields. In Physics, it represents the relationship between physical quantities and their rate of change.

In Mathematics, a differential equation is a mathematical equation that represents the relationship between some unknown functions and their derivatives. There are some types of differential equations. According to the properties of the equation, some of the most common types are ordinary or partial, linear or non-linear, and order equation. In the scope of this capstone project, we just mention the ordinary differential equation and the initial-value problem.

An ordinary differential equation is a differential equation which contains an unknown functions of a variable and its derivatives Ang and Park (2008). An ordinary differential equation in function $y(x)$ is given by:

$$a_0(x)y + a_1(x)y' + a_2(x)y'' + a_3(x)y^{(3)} + ... + a_{n-1}(x)y^{(n-1)} + a_3(n)y^{(n)} + p(x) = 0, \tag{2.1}$$

where, y' is the first order derivative, y'' is the second order derivative, $y^{(k)}$ is the k^{th} order derivative of function $y(x)$. In general, the equation of the form:

$$F(y, y', y'', y^{(3)}, ..., y^{(n-1)}, y^{(n)}) = 0, \tag{2.2}$$

which is called the implicit ordinary differential equation of order n (Simmons, 2016), or the explicit ordinary differential equation of order n form (Kreyszig, 2009) is given by:

$$f(y, y', y'', y^{(3)}, ..., y^{(n-1)}) = y^{(n)}. \tag{2.3}$$

To solve an ordinary differential equation, we "undo the derivatives" in the ordinary differential equation Archibald et al. (2005). However, in the scope of this capstone project, we do not dive deep into how to solve the general ordinary differential equation; we just focus on a particular problem, Initial-Value Problem.

2.1.1 Initial-Value Problem (IVP)

Sometimes, we get a ordinary differential equation and its initial values that the solution of the equation must satisfy given initial conditions, which is called *Initial-Value Problem* (IVP) (Burden and Faires, 1985). A n^{th} order initial-value problem includes two parts:

(i) A n^{th} order ordinary differential equation is given by the form:

$$y^{(n)} = f(t, y, y', y'', y^{(3)}, ..., y^{(n-1)}), \tag{2.4}$$

where f is a function of t, y and derivatives of y, y is a function of single variable t. f is continuous in open set Ω in the $(t, y, y', ..., y^{(n-1)})$ space.

(ii) Initial conditions which gives the values of $y(t), y'(t), ..., y^{(n-1)}(t)$ at particular point of t_0 can be written in the form:

$$\begin{cases} y(t_0) &= y_0 \\ y'(t_0) &= y_1 \\ y''(t_0) &= y_2 \\ & ... \\ y^{(n-1)}(t_0) &= y_{n-1} \end{cases}, \tag{2.5}$$

where, $(t_0, y_0, y_1, ..., y_{n-1}) \in \Omega$.

In the scope of this capstone project, we introduce the first order initial-value problem and some numerical method to approximate its solution. Give function $f : \Omega \subset \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, y is the function of a single variable t, and its first order y', the first order scalar initial-value problem is given by:

$$y'(t) = f(t, y(t)), \tag{2.6}$$

with a initial condition, which is a point in the domain of f, $(t_0, y_0) \in \Omega$ that $y(t_0) = y_0$.

Initial-value problems are quite complicated to solve, so there are two approaches to solve them. The first one is to simplify the problem to another one which can be solved and then use the solution of the simplified problem to approximate the original problem. The second approach, which is more common, is to use some methods to approximate the solution to the original problem. In the following part of this section, the second approach will be introduced.

2.1.2 The Existence and Uniqueness of Solutions

In this section, we introduce some methods to approximate a certain point satisfying the initial-value problem. Through this chapter, we consider to find solutions of an initial-value problem

$$y'(t) = f(t, y),\ y(t_0) = y_0. \tag{2.7}$$

A vector of function $\boldsymbol{y}$ of t is called *solutions* of an initial-value problem if it satisfies that problem (Mattheij and Molenaar, 2002).

For approximating the solutions to IVPs, we need some definitions and results from theories.

Definition 2.1.1. *A function $f(t, y)$ satisfies a **Lipschitz condition** on a set D if there is a constant $L \geqslant 0$ such that*

$$|f(t, y_1) - f(t, y_2)| \leq L|y_1 - y_2|,$$

*wherever $(t, y_1), (t, y_2)$ are in D. L is a constant which is called **Lipschitz constant** for f.*

Theorem 2.1.2 (The Existence and Uniqueness Theorem for First-Order Ordinary Differential Equations.)**.** *(Burden and Faires, 1985) Let $f(t, y)$ is continuous on $D = \{(t, y)|t_0 \leqslant t \leqslant T and - \infty \leqslant y \leqslant \infty\}$. If f satisfies a Lipschitz condition on D in the variable y, then the initial-value problem*

$$y'(t) = f(t, y),\ t_0 \leqslant t \leqslant T,\ y(t_0) = y_0$$

has a unique solution $y(t)$ for $t \in [t_0, T]$

Now, we know that IVP 2.7 has a unique solution if the function f satisfies Lipschitz condition. Besides, IVPs are also known as evolution problems (Mattheij and Molenaar, 2002) and solutions to them are denoted as

$$y(t) = \Psi(t; t_0, y_0), \tag{2.8}$$

where, $y(t)$ is a solution to problem 2.7 and $y(t_0) = y_0$. The curve $\Psi(t; t_0, y_0)|$, $t \in [t_0 - \epsilon, t_0 + \epsilon]$, ϵ is small, is defined to be the *trajectory* (orbit) of an IVP (Mattheij and Molenaar, 2002); if $t \geqslant t_0$, it is called an positive orbit.

2.1.3 Numerical Methods for Initial-Value Problems

Approximating an initial-value problem is not to find a continuous approximation, which is close to the solution to a problem. It is to find a point, sometimes space, approximating a value of the initial-value problem. There are many techniques to do this approximation, and they are known as numerical methods for solving initial problems. They are divided into three large categories: the Taylor series methods, linear multistep methods, and Runge–Kutta methods (Griffiths and Higham, 2010). In the scope of this capstone project, we will introduce some of them, Euler's method, Taylor methods (the second-order and the n^{th}-order Taylor method), and Runge-Kutta methods (RK2 and RK4). They are used to approximate solutions of the initial-value problems of the form:

$$y'(t) = f(t, y),\ t_0 \leqslant t \leqslant T,\ y(t_0) = y_0 \tag{2.9}$$

Before going into details of these three methods, Euler's Method, which can be interpreted as the generalization of three large above categories of methods, will be introduced.

Euler's Method

Euler's method is named after Leonhard Euler, a Swiss mathematician, physicist, astronomer, geographer, logician and engineer; he used it in his book *Institutionum calculi integralis* (Hairer et al., 2000). Euler's method is the simplest method to solve the ordinary differential equation with initial conditions.

Euler's method is used to approximate the solution of problem 2.9. Solutions to this method are not expected to be a continuous approximation; it will be approximated at some certain points, which is called mesh points, in the interval $[t_0, T]$. Mesh points is evenly distributed over the interval $[t_0, T]$. Assuming that there are N approximated points; mesh points are selected by

$$t_i = t_0 + ih, \qquad \text{for each } i = 0, 1, 2, ..., N, \tag{2.10}$$

in which, h is called step size, which is distance between two adjacent points; h can be given by $h = t_{i+1} - t_i = \left\lfloor \frac{T-t_0}{N} \right\rfloor$.

Following Taylor series of $y(t + h)$, we get the approximation with remainder

$$y(t + h) = y(t) + hy'(x) + \frac{1}{2!}h^2 y(\xi), \qquad \xi \in [t, t + h], \tag{2.11}$$

where $\frac{1}{2!}h^2 y(\xi)$, the remainder term, is called local truncation error *(LTE)*, which is denoted as $R_1(t)$ Then, if there exists a positive number M such that $|y''(t)| \leqslant M$, the remainder term follows

$$|R_1(t)| \leqslant \frac{1}{2}Mh^2. \tag{2.12}$$

From problem 2.9, we get $y'(t) = f(t, y(t))$. Replacing it into the Taylor series 2.11, we will obtain

$$y(t + h) = y(t) + hf(t, y(t)) + R_1(t). \tag{2.13}$$

With mesh points, given by the form 2.10, and $t = t_i$ $(i < N)$, we substitute into above formula

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + R_1(t_i). \tag{2.14}$$

Reconstructing formula 2.14 by denoting $w_i \approx y(t_i)$ to remove the remainder term, the Euler method is

$$w_0 = y_0, \tag{2.15}$$

$$w_{i+1} = w_i + hf(t_i, w_i), \qquad for\ each\ i = 0, 1, 2, ..., N - 1. \tag{2.16}$$

The Taylor Methods

In the above part, we introduce Euler's method for solving an initial-value problem. It is the most basic method, but in practice, it is seldom used because its error has a relationship with the step size h. In particular, the local truncation error is $\mathcal{O}(h^2)$, which means that it is proportional to the spare of the step size, and the global error is proportional to the step size. Euler's method is often the basis to construct more complex methods, one of them is introduced in this section, the family of Taylor methods.

As Euler's method, the Taylor methods are constructed basing on the Taylor series expansion. In this section, we introduce the second-order Taylor method in particular, and the n^{th}-order Taylor methods in general.

The second-order Taylor method. The second-order Taylor series expansion is given as the form:

$$y(t+h) = y(t) + hy'(t) + \frac{h^2}{2!}y''(t) + \mathcal{O}(h^3). \tag{2.17}$$

Setting $t = t_i$, we obtain

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \mathcal{O}(h^3). \tag{2.18}$$

As in the above part, $y'(t_i)$ can be computed from initial-value problem 2.9:

$$y'(t) = f(t, y(t)). \tag{2.19}$$

Thus, $y''(t)$ is computed by differentiating both sides of equation 2.19: $y''(t) = f'(t, y(t))$.

Substituting $y'(t)$ and $y''(t)$ into formula 2.18, we obtain

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2}f'(t_i, y(t_i)) + \mathcal{O}(h^3). \tag{2.20}$$

Denoting $w_i \approx y(t_i)$ to remove the remainder term, the second-order Taylor method is written as

$$w_0 = y_0, \tag{2.21}$$

$$w_{i+1} = w_i + hf(t_i, w_i) + \frac{h^2}{2}f'(t_i, y(t_i)), \qquad \text{for each } i = 0, 1, 2, ..., N-1. \tag{2.22}$$

The n^{th}-order Taylor method is the general form of the second-order Taylor method. It follows the n^{th}-order Taylor series expansion, which is given with remainder:

$$y(t+h) = y(t) + hy'(t) + \frac{h^2}{2!}y''(t) + \frac{h^3}{3!}y^{(3)}(t) + ... + \frac{h^n}{n!}y^{(n)}(t) + \mathcal{O}(h^{p+1}). \tag{2.23}$$

Similar to the second-order Taylor method, the n^{th}-order Taylor method is given by

$$w_0 = y_0, \tag{2.24}$$

$$w_{i+1} = w_i + hT^{(n)}(t_i, w_i), \qquad \text{for each } i = 0, 1, 2, ..., N-1, \tag{2.25}$$

where,

$$T^{(n)}(t_i, w_i) = f(t_i, w_i) + \frac{h}{2}f'(t_i, w_i) + ... + \frac{h^{n-1}}{n!}f^{(n-1)}(t_i, w_i).$$

Euler's method is also known as the first-order Taylor method.

Runge-Kutta Methods

As mentioned above, the family of Taylor methods has the high-order local truncation error, but its advantages are to require to compute the derivatives of $f(t, y)$. In above section, we introduce a method to eliminate to compute the derivatives, the family of linear multistep methods, and in this section, one more method will be introduced, the family of Runge-Kutta Methods.

In the later 1800s, some methods like ones in this section were used to approximate solutions to initial-value problems by Carl Runge. In 1901, Martin Wilhelm Kutta generalized the methods Runge used to solve first-order differential equations. Therefore, nowadays, we call these techniques the Runge-Kutta methods. The Runge-Kutta methods still have high-order local truncation error but completely eliminate the computation of derivatives (Burden and Faires, 1985). In the scope of this capstone project, we will introduce the Runge-Kutta second-order and the Runge-Kutta forth-order.

Before going into details of each methods, we need to consider Taylor's Theorem in two variables.

Theorem 2.1.3. *(Burden and Faires, 1985) Let $(t_0, y_0) \in D = \{(t, y)|t \in [a, b], y \in [c, d]\}$. Suppose that f(t, y) and its partial derivatives of order k $(1 \leq k \leq n+1)$ are continuous on D. For every $(t, y) \in D$, there exists ξ and μ such that $t \leq \xi \leq t_0$, $y \leq \mu \leq y_0$ and*

$$f(t, y) = P_n(t, y) + R_n(t, y),$$

where,

$$\begin{aligned} P_n(t,y) = f(t_0,y_0) &+ \left[(t-t_0)\frac{\partial f}{\partial t}(t_0,y_0) + (y-y_0)\frac{\partial f}{\partial y}(t_0,y_0)\right] \\ &+ \left[\frac{(t-t_0)^2}{2}\frac{\partial^2 f}{\partial t^2}(t_0,y_0) + (t-t_0)(y-y_0)\frac{\partial^2 f}{\partial t \partial y}(t_0,y_0) + \frac{(y-y_0)^2}{2}\frac{\partial^2 f}{\partial y^2}(t_0,y_0)\right] \\ &+ \ ... \\ &+ \left[\frac{1}{n!}\sum_{i=0}^{n}\binom{n}{i}(t-t_0)^{n-i}(y-y_0)^j\frac{\partial^n f}{\partial t^{n-i}\partial y^j}(t_0,y_0)\right] \end{aligned}$$

and

$$R_n(t,y)\frac{1}{(n+1)!}\sum_{i=0}^{n+1}\binom{n+1}{i}(t-t_0)^{n+1-i}(y-y_0)^i\frac{\partial^{n+1} f}{\partial t^{n+1-i}\partial y^i}(\xi,\mu).$$

In theorem 2.1.3, $P_n(t,y)$ is known as the n^{th}-order Taylor polynomial in two variable for f about (t_0,y_0), and $R_n(t,y)$ is the remainder term.

The Runge-Kutta Second-Order Methods. To derive a Runge-Kutta method, first, we have to determine values for a_1, α_1, and β_1 such that $a_1 f(t+\alpha_1, y+\beta_1)$ is a approximation of $T^{(2)}(t,y) = f(t+y) + \frac{h}{2}f'(t,y)$. Since

$$f'(t,y) = \frac{df}{dt}(t,y) = \frac{\partial f}{\partial t}(t,y) + \frac{\partial f}{\partial y}(t,y)\cdot y'(t), \tag{2.26}$$

we have,

$$T^{(2)}(t,y) = f(t,y) + \frac{h}{2}\frac{\partial f}{\partial t}(t,y) + \frac{h}{2}\frac{\partial f}{\partial y}(t,y)\cdot y'(t). \tag{2.27}$$

The first-order Taylor series expansion for $f(t+\alpha_1, y+\beta_1)$ about two variables (t,y) is given by

$$a_1 f(t+\alpha_1, y+\beta_1) = a_1 f(t,y) + a_1\alpha_1\frac{\partial f}{\partial t}(t,y) + a_1\beta_1\frac{\partial f}{\partial y}(t,y) + \mathcal{O}(h). \tag{2.28}$$

Matching coefficients in Equations 2.26 and 2.27, we get

$$\begin{aligned} a_1 &= 1, && (2.29)\\ a_1\alpha_1 &= \frac{h}{2}, && (2.30)\\ a_1\beta_1 &= \frac{h}{2}f(t,y), && (2.31) \end{aligned}$$

so, parameters are

$$\begin{aligned} a_1 &= 1, && (2.32)\\ \alpha_1 &= \frac{h}{2}, && (2.33)\\ \beta_1 &= \frac{h}{2}f(t,y), && (2.34) \end{aligned}$$

then,

$$T^{(2)}(t,y) = f\left(t+\frac{h}{2}, y+\frac{h}{2}f(t,y)\right) + \mathcal{O}(h^2). \tag{2.35}$$

Replacing $T^{(2)}(t,y)$ into second-order Taylor's method 2.25 and construct $w_i \approx y(t)$ to remove remainder term, we have a second-order Runge-Kutta method, which is also know as the Midpoint method.

$$\begin{aligned} w_0 &= y_0, && (2.36)\\ w_{i+1} &= w_i + hf(t_i + \frac{h}{2}, w_i + \frac{h}{2}f(t_i,w_i)), && (2.37) \end{aligned}$$

for each $i = 0, 1, ..., N-1$.

To approximate derivatives in the second-order Taylor's method, we need only three parameters. In a more complicated situation, it is required more parameters for higher-order Taylor methods. For example, to approximate

$$T^{(3)}(t,y) = f(t,y) + \frac{h}{2}f'(t,y) + \frac{h^2}{2}f''(t,y), \tag{2.38}$$

we have

$$a_1 f(t,y) + a_2 f(t + \alpha_2.y + \delta_2 f(t,y)). \tag{2.39}$$

We can also prove that $a_1 = a_2 = \frac{1}{2}$ and $\alpha_2 = \delta_2 = h$ makes equation 2.38 equal to equation 2.39 (Burden and Faires, 1985). In that case, we get another second-order Runge-Kutta method, which is called the Modified Euler method.

$$w_0 = y_0, \tag{2.40}$$

$$w_{i+1} = \frac{h}{2}\left[f(t_i, w_i) + f(t_{i+1}, w_i + hf(t_i, w_i))\right], \tag{2.41}$$

for each $i = 0, 1, ..., N-1$.

The Runge-Kutta Forth-Order Method. Another method in the family of Runge-Kutta method is the Runge-Kutta Forth-Order method, which has local truncation error $\mathbb{O}(h^4)$.

$$w_0 = y_0, \tag{2.42}$$

$$k_1 = hf(t_i, w_i), \tag{2.43}$$

$$k_2 = hf\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_1\right), \tag{2.44}$$

$$k_3 = hf\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_2\right), \tag{2.45}$$

$$k_4 = hf(t_{i+1}, w_i + k_3), \tag{2.46}$$

$$w_{i+1} = w_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{2.47}$$

for each $i = 0, 1, ..., N-1$.

2.2 Neural Networks

Over the past few decades, today *Artificial Intelligence* has become a popular topic in many fields of science and technology. The Google Translate multi-language translation system, Apple's Siri virtual assistant, the AlphaGo machine, Tesla's self-propelled car and Amazon's product suggestion system are a few of the standout apps. *Machine learning* is a subset of artificial intelligence and a sub-field of computer science. Its models are capable of self-learning based on the input data without any specific programming. This section covers a class of the most popular machine learning algorithms - feed-forward neural network, which is one of the most popular deep learning models nowadays.

Feedforward neural networks are information-processing models based on the re-simulation of nervous system activity of animals. This includes the transmission of information between biological neurons. However, the algorithms behind are often not used to accurately mimic what happens in the brains of animals. A classic example of this is image categorization: when you see an image of a cat, for example, your brain immediately recognizes it as an image of a cat, instead of a wolf, or any other animal. Although biology has had a lot of research into this process of image classification, how the brain can recognize images so quickly is still a studied topic. At this point, we have not been able to write a static model capable of analyzing images of animals and accurately classifying which species it belongs to.

We need a mathematical function that maps from input to output, from the cat image to the conclusion that the image is of the cat. However, as stated in the above paragraph, we do not know exactly what this function looks like, so we need an approximation function. The process of building the model to find this approximate function is called training (or learning) and inspired by human learning. In order to know a new animal, we first need to be given some examples for the brain to accept and learn information. Analogous in feed-forward neural networks, we need to provide samples labeled as input first, implying that these are examples of training models. When working

with approximation functions, we have to calculate the error between predicted outputs and desired outputs, and minimize this error during the training process. Thus, the soul of feed-forward neural networks is the optimization such that cost functions depending on the error are minimal.

In summary, it is ideal to consider the feedforward neural networks as functional approximation machines intended to accomplish factual speculation, in some cases drawing a few experiences from what we know about the brain, rather than as cerebrum work models. In this section, we introduce feedforward the neural network, which includes component, architecture and learning process.

2.2.1 Feedforward Neural Networks

Feedforward neural network is known as the simplest neural network and one of the most popular neural networks (Schmidhuber, 2015). In the feedforward neural network, the external data is received by A input layer and passed through the network to produce output. Figure 2.1 describes the architecture of a feedforward neural network.

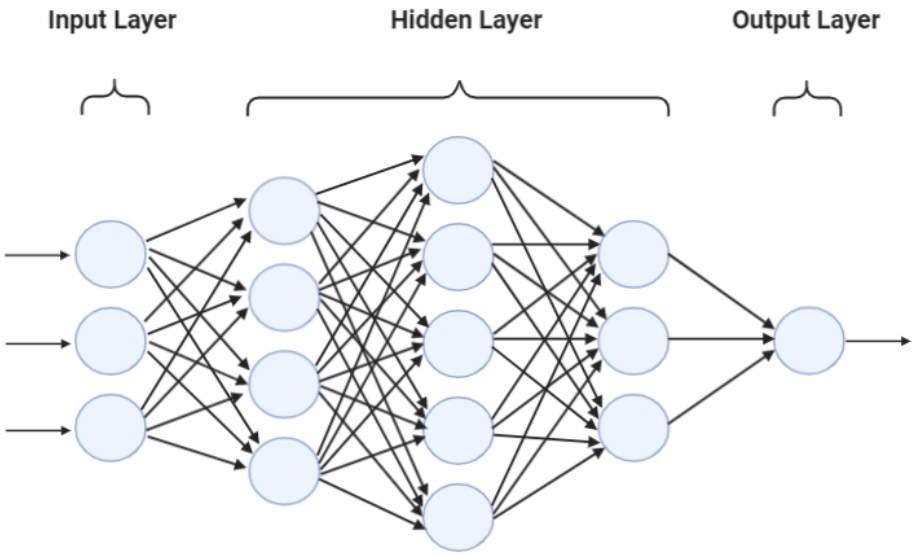

Figure 2.1: A feedforward neural network with 3 hidden layers

A feedforward neural network follows three attributes:

- Units, or artificial neurons, are organized into layers. The first layer takes the external data as input of the network; the last layer produces output, which is also predicted value. Layers between them are called hidden layers; there is no connection between them and external data.
- Each unit in one layer just connects to units in the previous and next layers.
- Units in the same layer do not connect to each other.

In the feedforward neural network, output of a layer is input of the immediately following layer so layers compose each other by a mapping called activation function. This makes model of network form the approximate value:

$$\hat{\boldsymbol{y}} = f^{(L)}(\boldsymbol{W}^{(L)\top} f^{(L-1)}(...(\boldsymbol{W}^{(3)\top} f^{(2)}(\boldsymbol{W}^{(2)\top} f^{(1)}(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + \boldsymbol{b}^{(2)}) + \boldsymbol{b}^{(3)})...) + \boldsymbol{b}^{(L)}), \quad (2.48)$$

where, L is the number of layers; $\boldsymbol{W}^{(k)}$ and $\boldsymbol{b}^{(k)}$ are parameters, which are weights and bias, of the k^{th} layer; $f^{(k)}(.)$ is the activation function of the k^{th} layer.

We have just taken a quick glance at the feedforward neural network; in the following part of this section, we will go into detail about components, architecture and learning process of a feedforward neural network.

Components of neural network

Units. Feedforward neural networks, and neural networks in general, consist of units, which take inspiration from biological neurons. Each unit, also known as an artificial neuron, takes one or more

data inputs and produces output, which can be shipped off numerous different units. The input of a unit is external data or output of one or more units.

Besides units, the network consists of connections that are defined as the relation between two units. These relations are measured by weights which are the strength of the connections. Weights influence the measure of the impact an adjustment in the input will have upon the output. Along with weights, biases, which are constants, are added into the input of the following layer. Biases speak for to how isolated the predictions are from the desired values, and they also ensure that the input of units can be activated when they are equal to zero.
To produce the output of a unit, first we sum the weighted inputs and add bias to it; the result of this process is called activation $\boldsymbol{a}$. $\boldsymbol{a}$ is given by:

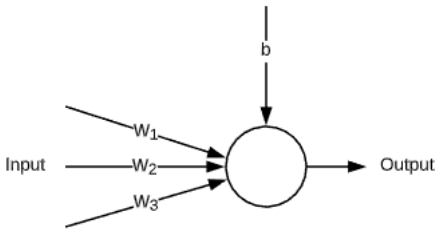

Figure 2.2: A unit in the neural network

$$\boldsymbol{a} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b} \tag{2.49}$$

In the final step, activations will be passed through the activation function. The value we receive after this process is the output of the unit.

Activation Function, a part of a unit in the neural network, is a mathematical mapping that determines the output of units. Activation function is also known as a transformation function, which transforms the input in other form, for example, in binary classification problems, the input is put through an activation function to receive the output in range 0 and 1.

The activation function may be divided into two categories: linear activation functions and non-linear activation functions. The linear activation function, or the identity activation function, is defined as $f(x) = x$. The linear activation function returns to output in no certain range $(-\infty, \infty)$, that doé not help anything in transform the input. In practical, the non-linear activation function is used to transform the input. In the scope of this capstone project, two of the most popular non-linear activation functions, sigmoid functions and ReLU, are introduced.

Sigmoid Function. Many Machine Learning problem required to classify input into two classes. Sigmoid function is a mapping which maps input to output which is in range $(0, 1)$; it is defined as

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.50}$$

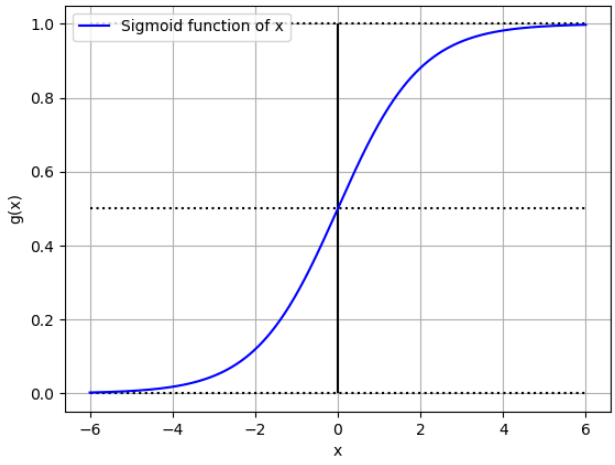

Figure 2.3: Sigmoid Function

The sigmoid function curve looks like a S-shape, it goes through the point $(0, 0.5)$. It is monotonic and has two horizontal asymptotic at $y = 0$ and $y = 1$.

The main reason why sigmoid function is used is that it returns the result between 0 and 1. Sigmoid function is also differentiable that means the slope of sigmoid curve can be found at any points; it is easy to calculate derivatives to update parameters at backpropagation step.

For multiclass classification problem, the softmax function is used instead of sigmoid function.

Rectified Linear Units (ReLU). Sigmoid function returns the value in range $(0, 1)$, that can cause a network to get stuck when training (we will explain deeper in following sections). To solve the problem caused by sigmoid function, the Rectified Linear Unit, or ReLU, activation function is given by

$$g(\boldsymbol{z}) = max\{0, \boldsymbol{z}\}, \tag{2.51}$$

where $\boldsymbol{z}$ is an activation; in the scope of this capstone project, $\boldsymbol{z}$ is written $\boldsymbol{z} = \boldsymbol{a} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$

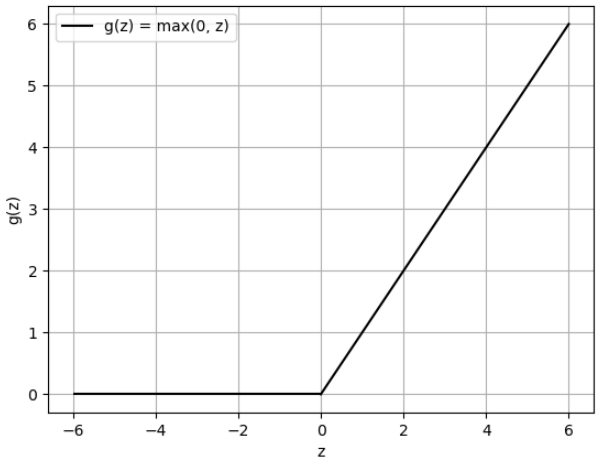

Figure 2.4: Rectified Linear Units Activation Functions

Layers. Units in a neural network are organized into multiple blocks, which are called layers. In feedforward neural networks, units in the same layers do not connect to each other; they only connect to units in the previous layers and units in the next layers.

There are three types of layers in a neural network. Each network has only one input layer and one output layer. Input layer is the first layer of a network, while output layer is the last layer of a network. Besides input layer and output layer, a multi-layer neural network can have zero or more hidden layers. Layers in the network are arranged in the order of input layer, hidden layers, and output layer. The number of layers in a network is denoted by L, which is calculated by summing the number of hidden layers and the number of output layers. The first layer, or the input layer, is denoted as the 0^{th} layer.

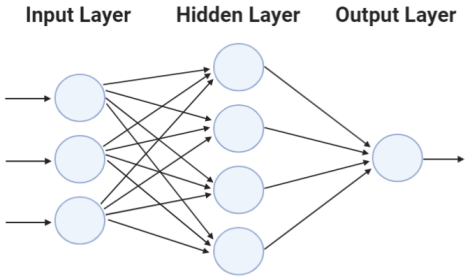

Figure 2.5: Layers of a feedforward neural network.

If there is more than one layer in a network, it is called a multi-layer neural network. The fully-connection between 2 layers is that every neuron in a layer connects to every neuron in another layer. In a feedforward neural network, except input layer, layers take input from the output of the immediately preceding layer. Each unit in the layer sends its output to the following layer, and it

will be transformed by operation 2.49, and become input of the following layer. Output of the output layers is also output of the models, which are predicted values. Outputs of the hidden layers are not shown that is the reason why they are called hidden.

Architecture of feedforward neural network

The term **architecture** refers to the structure of a network. In the feedforward neural networks' model, output of the k-th layer is given by:

$$\boldsymbol{h}^{(k)} = f^{(k)}(\boldsymbol{W}^{(k)\top}\boldsymbol{h}^{(k-1)} + \boldsymbol{b}^{(k)}), \tag{2.52}$$

where, $\boldsymbol{W}^{(k)}$ and $\boldsymbol{b}^{(k)}$ are the matrix of weights and vector of biases of the k-th layer. The $\boldsymbol{h}^{(k-1)}$, $\boldsymbol{h}^{(k)}$ are outputs of k^{th} and $(k-1)^{th}$ layer.
Following operation 2.52, in the feedforward neural network, input of a layer is the transformation of its immediately preceding layer's outputs. Layers in the feedforward neural network compose each other. This makes the feedforward neural network look like a directed acyclic graph, it has no feedback connection or no cycle. In figure 2.6, feedforward neural network's structure is like a chain; its architecture is also named chain structure.

Figure 2.6: Architecture of the feedforward neural network.

The Universal Approximation Theory. The goal of feedforward neural network model is to find a function f that maps any attributes $\boldsymbol{x}$ to output $\hat{y} = f(\boldsymbol{x})$. We expect that we can find a function f which produces output $\hat{y}$, very closed to the desired value y. The Universal Approximation Theorem shows that a feedforward neural network can represent any functions.

Cybenko (1989) showed that two-layer feedforward neural networks with sigmoid activation functions can approximate any continuous function on $D \subset \mathbb{R}^d$. In 1989, Hornik et al. proved that two-layer feedforward neural network with any "squashing" function can approximate any function $f(.)$. In 1990, it was proved that the derivatives of function $f(.)$ can be approximated by using the derivatives of the feedforward neural network (Hornik et al., 1990). In 1993, the universal approximation theorem was proved for a wider class of activation functions by Leshno et al..

The universal approximation theorem shows that there exists a feedforward neural network which is large enough to represent any functions. It is expected that the network has more layers, it can produce the output closer to the desired valued. However, in practice, increasing more numbers of layers in a network will lead to some problems, which we will discuss in some following sections.

2.2.2 Gradient-Based Optimization

Cost Function

The feedforward neural network model is a mapping that maps the input $\boldsymbol{x}$ to the output $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b})$ that is the best approximation of the desired value $f^*(\boldsymbol{x})$. There is difference between the predicted value $\hat{\boldsymbol{y}}$ and the desired value $\boldsymbol{y} = f^*(\boldsymbol{x})$, which generates an error. The goal of feedforward neural network model is to estimate parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ that the model produces a value very close to the desired value. Therefore, the strategies is to change parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ of model by minimizing the error of the network model.

The error of a network model is calculated by the **cost function**. It measures the performance of a neural network model with the given data. It is given by:

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}_i, y_i), \tag{2.53}$$

where, m is the size of given dataset; x_i is the input data; $\hat{y}_i$ and y_i is the predicted value and the desired value of the i^{th} data sample.

$L(.)$ is the loss function, which evaluates the difference between the predicted value and the desired value of a sample. In some case, it is ideal that the predicted value $\hat{y}$ is equal to the desired value y. However, in deep learning problems, it is not expected to get the optimal value; the predicted value $\hat{y}$ is expected to be closed to the desired value y.

There are many ways to defined the loss function. In the scope of this capstone project, we mention two of the most popular functions, they are *Mean Squared Error* (MSE) and *Cross-Entropy*.

Mean Squared Error, or MSE, is the most common loss function used in regression problems. It is measured by squaring of the distance between the predicted value $\hat{y}$ and the desired value y. The MSE loss is given by

$$L(\hat{y}, y) = (\hat{y} - y)^2. \tag{2.54}$$

Therefore, the cost function of the network become the mean of square of the errors

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y} - y)^2. \tag{2.55}$$

This loss function is used in regression problem, which is expected that the large errors have big effects on the cost functions.

Cross-Entropy Loss is a function which estimates the distance between two probability distributions. In Statistics, cross-entropy between two probability distributions $\boldsymbol{p}$ and $\boldsymbol{q}$ is defined as

$$H(\boldsymbol{p}, \boldsymbol{q}) = \mathbf{E}_{\boldsymbol{p}}[-\log \boldsymbol{q}]. \tag{2.56}$$

With $\boldsymbol{p}$ and $\boldsymbol{q}$ are discrete, 2.56 is rewritten

$$H(\boldsymbol{p}, \boldsymbol{q}) = -\sum_{i=1}^{C} p_i \log q_i. \tag{2.57}$$

In Deep Learning, the output of a unit belongs to one of two distribution $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$, where $\boldsymbol{y}$ is the probability of the desired output to be classified in the first class; and $\hat{\boldsymbol{y}}$ is the probability of the predicted output to be classified in the first class. Thus, the loss value of a sample is given,

$$L(y, \hat{y}) = y\log(\hat{y}) + (1 - y)\log(1 - \hat{y}). \tag{2.58}$$

The cost function of the model is sum of loss function, which is written,

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m}\sum_{i=1}^{m}(y_i \log(\hat{y_i}) + (1 - y_i)\log(1 - \hat{y_i})). \tag{2.59}$$

Gradient Descent

Gradient Descent is a optimization algorithm, in which local minimum of object function is found by moving iteratively in the direction of steepest descent. In some Machine Learning and Deep Learning problems, gradient descent is used to update parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ with cost function $J(\boldsymbol{W}, \boldsymbol{b})$ as a objective function.

Proposition 2.2.1. *To find the local minimum of a function, take the step in the direction of the negative of the gradient of a function at the current point.*

Proof. To prove proposition 2.2.1, we have problem

$$\min_{x} f(x) \tag{2.60}$$

Let $\mathbf{u}$ is a unit vector. The directional derivative of $f(x)$ in the direction of $\mathbf{u}$ is

$$D_{\mathbf{u}} f(x) = \nabla f(x) \cdot \mathbf{u}. \tag{2.61}$$

Now, proposition 2.2.1 becomes: Let

$$\min_{\mathbf{u}} \nabla f(x) \cdot \mathbf{u}, \tag{2.62}$$

where, $||\mathbf{u}|| = 1$, have local minimum if $\angle(\mathbf{u}, \nabla f(x)) = \pi$.
Easy to prove,

$$\begin{aligned} \nabla f(x) \cdot \mathbf{u} &= ||\nabla f(x)||\, ||\mathbf{u}||\, \cos\theta,\ \theta = \angle(\mathbf{u}, \nabla f(x)) && (2.63)\\ &= ||\nabla f(x)||\, 1\, \cos\theta && (2.64)\\ &= ||\nabla f(x)||\, \cos\theta. && (2.65) \end{aligned}$$

$\cos\theta \in [-1, 1]$, so $\nabla f(x) \cdot \mathbf{u} \geq -||\nabla f(x)||$.
The equal sign at $\theta = \pi + k2\pi$. □

Let $J(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b})$ is a cost function which is defined and differentiable. As mentioned below, the function $J(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b})$ decreases fastest if it moves in the direction of the negative of the gradient of $J(.)$. Hence, the update of $\boldsymbol{W}$ and $\boldsymbol{b}$ is written

$$\begin{aligned} \boldsymbol{W} &\leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J(\boldsymbol{W}, \boldsymbol{b}), && (2.66)\\ \boldsymbol{b} &\leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J(\boldsymbol{W}, \boldsymbol{b}), && (2.67) \end{aligned}$$

where, $\alpha \in \mathbb{R}_+$ is called learning rate, which is small enough.

Stochastic Gradient Descent

In general, gradient descent is quite slow when applying in very large datasets. Stochastic gradient descent , or SGD, is an extension of the gradient descent algorithm to solve the problem of training models with large datasets.

The gradient of a cost function by parameters is computed by

$$\nabla_{\boldsymbol{W}} J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{W}} L(x^{(i)}, y^{(i)}, \boldsymbol{W}, \boldsymbol{b}) \tag{2.68}$$

and

$$\nabla_{\boldsymbol{b}} J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{b}} L(x^{(i)}, y^{(i)}, \boldsymbol{W}, \boldsymbol{b}). \tag{2.69}$$

The computation cost of operation (2.68) and (2.68) is $\mathcal{O}(m)$. It means if the size of the training dataset grows too large, the time need to compute and update parameters becomes too long. The solution for this problem is to divide the dataset into small part, and update parameters based on that samples, this method is called Stochastic Gradient Descent (SGD). On each step of SGD algorithm, we sample data to small pieces size m' uniformly, which is known as minibatch of examples, $B = \{x^{(1)}, x^{(2)}, ..., x^{(m')}\}$. The size of minibatch m' is chosen such that $1 \leq m' < m$, the value of m' can be a few hundred, depending of the properties of the datasets. Sampling dataset into minibatch data reduces the time complexity of the algorithm.

The gradient is written

$$\boldsymbol{g}_{\boldsymbol{W}} = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\boldsymbol{W}} L(x^{(i)}, y^{(i)}, \boldsymbol{W}, \boldsymbol{b}), \tag{2.70}$$

and

$$\boldsymbol{g}_{\boldsymbol{b}} = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\boldsymbol{b}} L(x^{(i)}, y^{(i)}, \boldsymbol{W}, \boldsymbol{b}). \tag{2.71}$$

Parameters of model $\boldsymbol{W}$ and $\boldsymbol{b}$ are updated as

$$\begin{aligned} \boldsymbol{W} &\leftarrow \boldsymbol{W} - \alpha \boldsymbol{g}_{\boldsymbol{W}} J(\boldsymbol{W}, \boldsymbol{b}), && (2.72)\\ \boldsymbol{b} &\leftarrow \boldsymbol{b} - \alpha \boldsymbol{g}_{\boldsymbol{b}} J(\boldsymbol{W}, \boldsymbol{b}), && (2.73) \end{aligned}$$

where, α is learning rate.

2.2.3 The Backpropagation Algorithm

The previous section described architecture a neural network. Learning process of training a feedforward neural network consists of two phases: Forward Propagation and Backward Propagation.

Forward propagation refers to flows of information in order from the input $\boldsymbol{x}$ that pass through a collection of hidden layers to the output $\hat{\boldsymbol{y}}$ by calculating and storing intermediate variables. Learning (or training) in deep neural networks requires computing the gradients for finding the optimal value of cost function with back-propagation (or sometimes simply as "backprop" for short algorithm. In other words, back-propagation is a method to compute the partial derivatives of a cost function with respect to the parameters of networks. In the scope of this capstone project, we will discuss about the gradient of a function with chain rule and the way applying it to the back-propagation in deep neural networks.

Chain Rule in Calculus

In calculus, the chain rule is a formula to compute the derivative of a function, in which this function is composed by other differentiable ones.

First of all, we consider the chain rule with one variable (or the scalar case). Let function g is differentiable at x and function f is differentiable at $g(x)$, their composite $F = f \circ g$ maps from a real number x to a real number that is a result of the function $f(g(x))$. Therefore, the composite function F is differentiable at x and is defined by $F(x) = f(g(x))$. The chain rule, expressing the derivative F', is written in Lagrange's notation as follows:

$$F'(x) = f'(g(x)).g'(x). \tag{2.74}$$

We assume that $y = g(x)$ and $z = f(g(x)) = f(y)$. The variable z depends on the variable x via the intermediate variable y ,if z depends on y and y depends on x. This relationship can also be described by the chain rule in Leibniz's notations as the following way:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{2.75}$$

The proof of chain rule: Let Δy be the change in y corresponding to change of Δx in x and Δz be the change in y corresponding to change of Δy in y, we have:

$$\Delta y = g(x + \Delta x) - g(x), \tag{2.76}$$
$$\Delta z = f(y + \Delta y) - f(y). \tag{2.77}$$

It is tempting to write

$$\begin{aligned}
\frac{dz}{dx} &= \lim_{\Delta x \to 0} \frac{\Delta z}{\Delta x} \\
&= \lim_{\Delta x \to 0} \frac{\Delta z}{\Delta y}.\frac{\Delta y}{\Delta x} \\
&= \lim_{\Delta x \to 0} \frac{\Delta z}{\Delta y}. \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \\
&= \lim_{\Delta y \to 0} \frac{\Delta z}{\Delta y}. \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \\
&= \frac{dz}{dy}\frac{dy}{dx}.
\end{aligned} \tag{2.78}$$

Next, we consider the chain rule in the multivariable case with $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, g is a function mapping from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}^k$. In terms of components, f and g are expressed as $\mathbf{y} = g(\mathbf{x}) = (g_1(\mathbf{x}), g_2(\mathbf{x}), ..., g_n(\mathbf{x}))$ and $\mathbf{z} = f(\mathbf{y}) = (f_1(\mathbf{y}), f_2(\mathbf{y}), ..., f_k(\mathbf{y}))$. The chain rule for Jacobian matrices is the following formula:

$$\frac{\partial(z_1, ..., z_k)}{\partial(x_1, ..., x_m)} = \frac{\partial(z_1, ..., z_k)}{\partial(y_1, ..., y_n)}\frac{\partial(y_1, ..., zy_n)}{\partial(x_1, ..., x_m)}. \tag{2.79}$$

We may simplify the above formula to get:

$$\frac{\partial(z_1, ..., z_k)}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial(z_1, ..., z_k)}{\partial y_l}\frac{\partial y_l}{\partial x_i}. \tag{2.80}$$

This can be rewritten as a dot product in vector notation:

$$\nabla_{\mathbf{x}}\mathbf{z} = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}}\mathbf{z}. \tag{2.81}$$

Chain rule in Back-propagation

In order to use chain rules to calculate derivatives, the original function must be a composite function first. As being presented in section 1.2, the function denoting to feed-forward propagation is a composite function. As a result, the cost function J depending on $\hat{\mathbf{y}}$ is also a composite function. Therefore, we can apply chain rule to compute derivatives. In the back-propagation, we need to find the derivatives of the cost function in terms of the weights and biases.

For a single weight in l^{th} hidden layer, at the position (j, k), $j = 1, ..., n$ where n is the number of inputs, $k = 1, ..., d$ where d is the number of linear combinations within one layer, the derivative is:

$$\frac{\partial J}{\partial w_{jk}^{(l)}} = \frac{\partial J}{\partial h_j^{(l)}} \frac{\partial h_j^{(l)}}{\partial w_{jk}^{(l)}}. \tag{2.82}$$

The same goes for biases:

$$\frac{\partial J}{\partial b_j^{(l)}} = \frac{\partial J}{\partial h_j^{(l)}} \frac{\partial h_j^{(l)}}{\partial b_j^{(l)}}. \tag{2.83}$$

In short, we have covered feed-forward, the chain rule, and the derivative calculation for individual weights and bias. In the next section, we will discuss the back-propagation and its relationship to forward in fully-connected Multi-layer Perceptron (MLP).

Backpropagation Algorithm in Feedforward Neural Network

For convenience, the chosen fully-connected MLP is simple and straightforward to understand. All of gradients will be computed by using the above chain rule in this part. Before applying chain rule to back-propagation, the cost function J needs to be computed in the forward propagation.

As being presented in the feedforward section, the forward process shows how to compute the total cost J with respect to the weight matrices $\mathbf{W}$ and the bias matrices $\mathbf{b}$. The cost function is the sum of the loss $L(\hat{\mathbf{y}}, \mathbf{y})$ and an optional regularizer $\Omega(\theta)$.The function denotes to each layer k is $f(\mathbf{a}^{(k)})$, where $\mathbf{a}^{(k)}$ is an activation. Briefly we have this equation $f(\mathbf{a}^{(k)}) = f(\mathbf{W}^{(k)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k)})$ with $\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$. Following algorithm 1 is the pseudo-code for summarizing forward propagation step-by-step with l layers:

Algorithm 1 Forward Propagation

Require: $\boldsymbol{W}$, the weight matrix
Require: $\boldsymbol{b}$, the bias vector
Require: $\hat{\boldsymbol{y}}$, the target ouput
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, .., l$ **do**
    $\boldsymbol{a}^{(k)} = \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)} + \boldsymbol{b}^{(k)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)$

After getting the cost function J from the forward propagation, the task of backward propagation is to compute the gradients to find the optimal value for the cost function J. In other words, we have to find the values of weights $\mathbf{W}$ and biases $\mathbf{b}$ such that J is minimal:

$$\min_{W,b} J$$

In order to find satisfying values of $\mathbf{W}$ and $\mathbf{b}$, backward propagation computes the gradients of cost function J on weights ($\nabla_{\mathbf{W}^{(k)}} J$) and biases ($\nabla_{\mathbf{b}^{(k)}} J$) by the chain rule. Gradients will be computed from the output layer to the first hidden layer and used for updating parameters right after being computed. Back-propagation is presented in the algorithm 1.

Algorithm 2 Back-propagation and Update parameters

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{h})$
for $k = l, l-1, ..., 1$ **do**
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases:
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

  Update weights and biases with a positive learning rate α:
  $\boldsymbol{W}^{(k)} = \boldsymbol{W}^{(k)} - \alpha \nabla_{\boldsymbol{W}^{(k)}} J$
  $\boldsymbol{b}^{(k)} = \boldsymbol{b}^{(k)} - \alpha \nabla_{\boldsymbol{b}^{(k)}} J$

  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
end for

2.2.4 Problems of Feedforward Neural Network

Vanishing Gradient in Feedforward Neural Network

In section 2.2.1, we mention the depth of the feedforward neural network. It is believed that the deeper networks are, the better they can learn. However, in practice, very deep networks lead to many problems, one of which is vanishing gradient problem.

The vanishing gradient problem is occurred when training feedforward neural networks with backward propagation through too many layers. In the backward propagation step, the gradient of a layer is computed based on its following layer; gradient of cost function is calculated by chain rule:

$$\boldsymbol{g}_{\boldsymbol{W}} J(\boldsymbol{W}, \boldsymbol{b}) = \frac{\partial J}{\partial \boldsymbol{W}} = \frac{\partial J}{\partial \boldsymbol{h}^{(L)}} \frac{\partial \boldsymbol{h}^{(L)}}{\partial \boldsymbol{h}^{(L-1)}} ... \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{W}}, \tag{2.84}$$

and,

$$\boldsymbol{g}_{\boldsymbol{b}} J(\boldsymbol{W}, \boldsymbol{b}) = \frac{\partial J}{\partial \boldsymbol{b}} = \frac{\partial J}{\partial \boldsymbol{h}^{(L)}} \frac{\partial \boldsymbol{h}^{(L)}}{\partial \boldsymbol{h}^{(L-1)}} ... \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{b}}, \tag{2.85}$$

In some feedforward neural networks, "traditional" activation functions "normalize" the output of each layers, that makes gradient very small. In backpropagration step, we multiply very small values to compute the gradient, that makes gradients still so small. Thus, the weights and bias are slowly updated and the training time increases.

In contrast to vanishing, exploding gradient problem is the situation that the gradients are calculated by multiplying too large values.

Too overcome vanishing gradient problem, many methods were proposed. One of the newest ways to resolve this problem is Residual Neural Networks, or ResNets.

Residual Neural Network

The idea of Residual Neural Network was proposed by He, Zhang, Ren, and Sun in 2016. The residual network is organized into building blocks, which includes some layers; a building block connects to its following block by a shortcut connection. A building block is given by

$$\boldsymbol{y} = \boldsymbol{x} + g(\boldsymbol{x}; \theta), \tag{2.86}$$

where, $\boldsymbol{x}$, $\boldsymbol{y}$ are input and output; $\boldsymbol{\theta}$ is a parameter of the building block. The function $g(.)$ is called residual mapping; the operation $x + g(.)$ represent an element-wise addition and a shortcut connection (He et al., 2016).

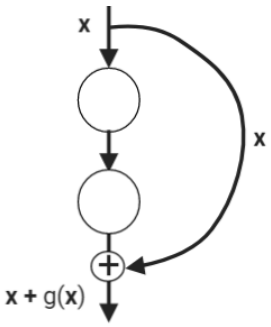

Figure 2.7: A building block in a residual network

Thus, the hidden state in a residual network is given by

$$\boldsymbol{z}^{(i+1)} = \boldsymbol{z}^{(i)} + g(\boldsymbol{z}^{(i)}; \theta^{(i)}) \tag{2.87}$$

where, $z^{(i)}$ is the hidden state at the i^{th} layer.

When stacking an infinitive numbers of layers in a residual network will lead to the neural ordinary differential equations, which we will discuss in the following chapter.

Chapter 3

Neural Ordinary Differential Equations Network(Neural ODEs-Net)

We use notations: d_x is the total derivative (gradient usually denoted by $\mathrm{d}(.)/\mathrm{d}x$ or ∇_x), ∂_x is the partial derivative (usually, $\partial(.)/\partial_x$), d is the differential, and $\dot{x} = \mathrm{d}x/\mathrm{d}t$.

3.1 Introduction

As mentioned in the background chapter, the hidden state in a residual network can be written as the flowing formulation

$$\mathbf{z_{t+1}} = \mathbf{z}_t + g(\mathbf{z}_t, \theta_t), \tag{3.1}$$

where $t \in \{0...T\}$ and $\mathbf{z}_t \in \mathbb{R}^d$ is the hidden state at layer t. This equation is rewritten as

$$\frac{\mathbf{z}_{t+1} - \mathbf{z}_t}{(t+1) - t} = g(\mathbf{z}_t, \theta_t). \tag{3.2}$$

If we add more layers until the number of layers goes to infinite, then the left side of Equation 3.2 is the derivative of hidden state respect to t. Equation (3.2) can be expressed as an ordinary differential equation with the initial condition is the input $\mathbf{x}$, then we get the following IVP

$$\begin{aligned} \frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} &= g(\mathbf{z}(t), \theta(t)), \quad t \in [0, T], \\ \mathbf{z}(0) &= \mathbf{x}. \end{aligned} \tag{3.3}$$

Chen et al. (2018) proposes a new family of neural network called Neural Ordinary Differential Equations Network (Neural ODEs-Net). The main idea is to use a neural network of form $f(\mathbf{z}(t), t, \theta)$ to replace g in the IVP (3.3). Then, the new IVP will be

$$\begin{aligned} \frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} &= f(\mathbf{z}(t), t, \theta), \quad t \in [0, T], \\ \mathbf{z}(0) &= \mathbf{x}. \end{aligned} \tag{3.4}$$

This IVP starts from the input $\mathbf{z}(0)$, passes through an ordinary differential equations solver to get the outcome $\mathbf{z}(T)$ at some time T. Within this chapter, all parameters θ are fixed over time and not dependent on t. This is slightly different from the Resnet version shown through Equation (3.3) when the parameter $\theta(t)$ depends on t. The main difference between Resnet and Neural ODE is that Resnet works with discrete depth and Neural ODEs work with continuous depth. Using Neural ODEs has some advantages when we no need to store any intermediate quantities of the forward for computing gradients. The next section explains the learning process of this type of neural network.

3.2 Learning Process of Neural ODEs-Net

Like the neural networks that existed before, we need to learn parameters θ of the dynamics $f(\mathbf{z}(t), t, \theta)$, in which θ is independents on t . This learning process can be compassed by using the Adjoint Sensitive

Method (Pontryagin LS, 1962) and treating ODE solvers as black boxes. The two following parts detail both the forward and backward propagation.

3.2.1 Continuous Forward Propagation

As mentioned in the IVP (3.4), $\mathbf{z}(t)$ follows the differential equation $\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} = f(\mathbf{z}(t), t, \theta)$, where θ are the parameters. Theoretically, we can find the value of $\mathbf{z}(T)$ at the last time T with the initial value $\mathbf{z}(0)$.

$$\mathbf{z}(T) = \mathbf{z}(0) + \int_0^T f(\mathbf{z}(t), t, \theta)\mathrm{d}t. \tag{3.5}$$

The forward propagation that returns the outcome $\mathbf{z}(T)$ can be completely resolved by using ODE solvers without the need for computing integral. We treat the ODE solver as a back box, in which inputs include the initial value, the derivative, and the range of time $[0, T]$.

3.2.2 Continuous Backward Propagation

The following formula is the loss function $L(.)$ of Neural ODEs

$$L(\mathbf{z}(T)) = L\left(\mathbf{z}(0) + \int_0^T f(\mathbf{z}(t), t, \theta)\mathrm{d}t\right). \tag{3.6}$$

During backpropagation phrase, we need to find $\frac{\mathrm{d}L}{\mathrm{d}\theta}$ to update θ for optimizing the loss function L. However, using backpropagation with chain rule requires storing a huge number of intermediate quantities from forward. This makes the computational graph too large to hold in memory. This problem can be solved efficiently by a well-known technique called The Adjoint Sensitive method.

The Adjoint Sensitive Method

Consider the constrained optimization problem

$$\min_{\theta} L(\mathbf{z}(T)) \tag{3.7}$$

subject to

$$\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} - f(\mathbf{z}(t), t, \theta) = \mathbf{0}, \tag{3.8}$$

$$\mathbf{z}(0) - \mathbf{x} = \mathbf{0}. \tag{3.9}$$

For convenience, we define two functions $h(\dot{\mathbf{z}}(t), \mathbf{z}(t), t, \theta) = \frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} - f(\mathbf{z}(t), t, \theta) = \mathbf{0}$ and $g(\mathbf{z}(0)) = \mathbf{z}(0) - \mathbf{x} = \mathbf{0}$. We also use an auxiliary function called the Lagrangian to convert the above problem to the unconstrained optimization problem by using $\lambda(t)$ and μ of Lagrange multipliers

$$\mathcal{L} \triangleq L(\mathbf{z}(T)) + \int_0^T \lambda(t) h(\dot{\mathbf{z}}(t), \mathbf{z}(t), t, \theta)\mathrm{d}t + \mu g(\mathbf{z}(0)), \tag{3.10}$$

where $\lambda(t)$ is a function of time, and μ is a vector associating with the initial conditions.

Because $g(\mathbf{z}(0))$ and $h(\dot{\mathbf{z}}(t), \mathbf{z}(t), t, \theta)$ are everywhere zero by construction, we may choose λ and μ freely, $L(\mathbf{z}(T)) = \mathcal{L}$, and we have

$$\mathrm{d}_\theta L = \mathrm{d}_\theta \mathcal{L}. \tag{3.11}$$

The adjoint sensitive method is a well-known technique reach to $\mathrm{d}_\theta \mathcal{L}$ without computing directly $\mathrm{d}_\theta \mathbf{z}$ during the backpropagation phrase by choosing $\lambda(t)$ and μ, because computing $\mathrm{d}_\theta \mathbf{z}$ is difficult in most cases. Nextly, how to apply the adjoint method sensitive method will be explained in details.

From Lagrange multipliers 3.10, we have

$$\mathcal{L} = L(\mathbf{z}(T)) + \int_0^T \lambda(t)\big(\dot{\mathbf{z}}(t) - f\big)\mathrm{d}t + \mu g(\mathbf{z}(0)) \tag{3.12}$$

$$= L(\mathbf{z}(T)) + \lambda(t)\mathbf{z}(t)\big|_0^T - \int_0^T \mathbf{z}(t)\dot{\lambda}(t)\ \mathrm{d}t - \int_0^T \lambda(t) f \mathrm{d}t + \mu g(\mathbf{z}(0)) \tag{3.13}$$

$$= L(\mathbf{z}(T)) + \lambda(T)\mathbf{z}(T) - \lambda(0)\mathbf{z}(0) - \int_0^T \big(\mathbf{z}(t)\dot{\lambda}(t) + \lambda(t) f\big)\mathrm{d}t + \mu g(\mathbf{z}(0)). \tag{3.14}$$

With $\mathrm{d}_\theta f = \partial_\theta f + \partial_{\mathbf{z}} f \mathrm{d}_\theta \mathbf{z}$, $\mathrm{d}_\theta \mathbf{z}(0) = \mathbf{0}$ and $\mathrm{d}_\theta g = \mathrm{d}_\theta \mathbf{z}(0) + \mathrm{d}_\theta \mathbf{x} = \mathbf{0}$, we get

$$\mathrm{d}_\theta \mathcal{L} = \partial_{\mathbf{z}(T)} L \mathrm{d}_\theta \mathbf{z}(T) + \lambda(T)\mathrm{d}_\theta \mathbf{z}(T) - \lambda(0)\mathrm{d}_\theta \mathbf{z}(0) - \int_0^T \big(\dot{\lambda}(t)\mathrm{d}_\theta \mathbf{z} + \lambda(t)\mathrm{d}_\theta f\big)\ \mathrm{d}t + \mu \mathrm{d}_\theta g \tag{3.15}$$

$$= \big(\partial_{\mathbf{z}(T)} L + \lambda(T)\big)\mathrm{d}_\theta \mathbf{z}(T) - \int_0^T \big(\dot{\lambda}(t) + \lambda(t)\partial_{\mathbf{z}} f\big)\mathrm{d}_\theta \mathbf{z}\ \mathrm{d}t - \int_0^T \lambda(t)\partial_\theta f\ \mathrm{d}t. \tag{3.16}$$

To avoid computing $\mathrm{d}_\theta \mathbf{z}(T)$ and $\mathrm{d}_\theta \mathbf{z}$, we choose $\lambda(t)$ such that $\dot{\lambda}(t) = -\lambda(t)\partial_{\mathbf{z}} f$ with initial value $\lambda(T) = \partial_{\mathbf{z}(T)} L$ and $\lambda(t)$ is called *adjoint state*. Then, Equation 3.16 of the derivativeof the Lagrangian with respect to θ simplifies to

$$\mathrm{d}_\theta \mathcal{L} = -\int_0^T \lambda(t)\partial_\theta f\ \mathrm{d}t. \tag{3.17}$$

In summary, the derivative of the loss function with respect to θ can be computed by solving

$$\begin{bmatrix} \mathbf{z}(0) - \mathbf{z}(T) \\ \lambda(0) - \lambda(T) \\ d_\theta L \end{bmatrix} = \int_T^0 \begin{bmatrix} f \\ -\lambda(t)\partial_z f \\ -\lambda(t)\partial_\theta f \end{bmatrix} \mathrm{d}t, \tag{3.18}$$

in which $\lambda(t)$ is a function of time with the initial value $\lambda(T) = \partial_{\mathbf{z}(t)} L$. Automatic differentiation can efficiently evaluate the vector-Jacobian products $-\lambda(t)\partial_z f$ and $-\lambda(t)\partial_\theta f$. We use an abbreviation numerical ODE solver to solve 3.18. Algorithm 3 gives a summary of the full adjoint sensitive method for the backpropagation of Neural ODEs-Net. ODESolver receives four elements as inputs: functions of derivatives, initial values, an end time value, and a start time value. Outcomes are the last values backward at time $t = 0$. Another more modern approach demonstrated for the Adjoint Sensitive method proved by Chen et al. (2018) is presented in the Appendix A.

Algorithm 3 The full Adjoint Sensitive Method for Neural ODEs

1: $\mathbf{z}(T) = \mathrm{ODESolver}(f, \mathbf{z}(0), 0, T)$.
2: Compute $\lambda(T) = \partial_{\mathbf{z}(T)} L$.
3: Use the abbreviation ODESolver

$$\begin{bmatrix} \mathbf{z}(0) \\ \lambda(0) \\ d_\theta L(\mathbf{z}(T)) \end{bmatrix} = \mathrm{ODESolver}\left(\begin{bmatrix} f \\ -\lambda(t)\partial_z f \\ -\lambda(t)\partial_\theta f \end{bmatrix}, \begin{bmatrix} \mathbf{z}(T) \\ \lambda(T) \\ \mathbf{0} \end{bmatrix}, T, 0\right).$$

3.3 Implementation Neural ODEs-Net for Supervised Learning

We explain the training of neural ODEs for supervised learning, in particular for classification and regression.

3.3.1 Choosing $f(\mathbf{z}(t), t, \theta)$ function

This section explains how to choose and design f function. We must emphasize that f can be parameterized by any function, any neural network with an activation function is not necessarily ReLU. The existence and uniqueness theorem always guarantees that there exists only one solution to a first-order differential equation that satisfies a given initial condition. Following the defined architecture in Chen et al. (2018), we choose f is a Convolutional Neural Network (CNN) or a Multilayer Perceptrons (MLPs) with weights not depending on time.

3.3.2 Architecture

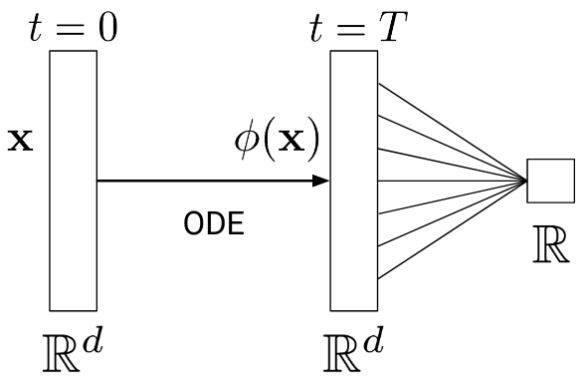

Figure 3.1: Diagram of a Neural ODEs-Net architecture followed by a linear layer Dupont et al. (2019)

For classification and regression, our learning function goes from $\mathbb{R}^d$ to $\mathbb{R}$. We follow an example in (Lin and Jegelka, 2018) for ResNets, and create a simple model architecture as Figure 3.1 with an ODEs layer, followed by a linear layer. The ODEs map the input $\mathbf{x} \in \mathbb{R}^d$ to a set of features in $\mathbb{R}^d$. Then, a linear function will map this set of feature from $\mathbb{R}^d$ to $\mathbb{R}$.

3.3.3 Benefits of Using Neural ODEs-Net

By using the Adjoint Sensitive method and treating the ODE solver as a black box, Neural ODE-Net has low memory cost and scales linearly with problem size. Defining and evaluating models using ODE solvers allows us to leverage computing power with memory efficiency.

Memory Benefits

With using the Adjoint Sensitive method in the backpropagtion phrase, there is no need to storing any intermediate quantities of the forward backpropagation (Chen et al., 2018) We only need the last outcome $\mathbf{z}(T)$ at time $t = T$ during the forward phase. Therefore, we can train our models with *constant memory cost*, $\mathcal{O}(1)$. There are reversible versions of ResNets (Gomez et al., 2017; Haber and Ruthotto, 2017; Chang et al., 2017) giving the same constant memory advantage. However, their architectures are restricted by partitioning the hidden units. Neural ODEs-Net does not have these restrictions.

Computation Benefits

The history of the development of efficient and accurate ODE solvers has been more than 120 years (Runge, 1895; Kutta, 1901; Hairer, 1987) with many methods such as the Euler method, Runge–Kutta methods, Taylor methods, etc. Modern ODE solvers provide assurances of the approximation errors growth, track the level of error and quickly adjust their evaluation strategy to accomplish the required level of accuracy. All of these allow the evaluating cost to scale with the problem complexity.

Chapter 4

Extensions of Neural ODEs-Net

4.1 Properties of Neural ODEs-Net

This section elucidates some special properties of Neural ODEs. They might be reasons for limitations in using Neural ODEs. Each following subsection is constructed with a property at the beginning and its disadvantages in the end.

4.1.1 Trajectories in Neural ODEs-Net cannot intersect

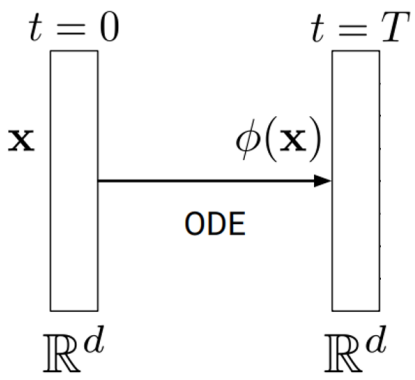

Figure 4.1: Diagram of a simple model architecture with an ODE layer

For convenience, we derive definitions for both flows and trajectories of ODE in this part.

ODE flows

A flow $\phi_t : \mathbb{R}^d \to \mathbb{R}^d$ is a hidden state at time t of neural ODEs with initial condition $\mathbf{x}$. This means that $\phi_t(\mathbf{x}) = \mathbf{z}(t)$. We are only concerned with the final time T, so we define the features of the ODE as $\phi(\mathbf{x}) = \phi_T(\mathbf{x})$. $\phi(\mathbf{x})$ is also called *the feature mapping*. Figure 4.1 is a simple model architecture with an ODE layer and a flow $\phi(x)$ within.

ODE trajectories

An important thing needed to emphasize is that ODE trajectories do not intersect each other. The following is a proof for this.

Proposition 4.1.1. *Let $\mathbf{z}_1(t)$ and $\mathbf{z}_2(t)$ be two trajectories of ODE 3.4 with two different initial conditions, $\mathbf{z}_1(0) \neq \mathbf{z}_2(0)$. Then $\mathbf{z}_1(t) \neq \mathbf{z}_2(t)$ for all $t \in (0, T]$. This implies that ODE trajectories do not intersect each other.*

Proof. Suppose that there exists at least a value $\tilde{t} \in (0, T]$ such that $\mathbf{z}_1(\tilde{t}) = \mathbf{z}_2(\tilde{t})$. We define a new IVP with the initial condition $\mathbf{z}(\tilde{t}) = \mathbf{z}_1(\tilde{t}) = \mathbf{z}_2(\tilde{t})$ and solve it backwards to $t = 0$. As the proof of existence and uniqueness of Solution in the background section, the solution at $t = 0$ is unique. This means that $\mathbf{z}_1(0) = \mathbf{z}_2(0)$. However, $\mathbf{z}_1(0)$ cannot equal to $\mathbf{z}_2(0)$, so our supposition is false. Therefore, $\mathbf{z}_1(t) \neq \mathbf{z}_2(t)$ for all $t \in (0, T]$, this implies ODE trajectories cannot intersect. □

Disadvantages: If a function has the intersecting trajectories, then Neural ODEs cannot represent it. In section 4.2, we will prove this problem for a class of functions. It is clear that the original Neural ODEs is not universal approximation.

4.1.2 Neural ODEs-Net describes a Homeomorphism

Dupont et al. (2019) showed a brief proof that the feature mapping ϕ_t is a homeomorphism. The following is the standard form of the Gronwall inequality which is used in this proof.

Theorem 4.1.2 (The Gronwall inequality). *Let $U \subset \mathbb{R}^d$ be an open set. Let $f : [t_1, t_2] \to \mathbb{R}^d$ be a continuous function and let $\mathbf{z}_1, \mathbf{z}_2 : [t_1, t_2] \to \mathbb{R}^d$ satisfy the IVPs*

$$\frac{d\mathbf{z}_1(t)}{dt} = f(\mathbf{z}_1(t), t), \quad \mathbf{z}_1(t_1) = \mathbf{x}_1, \tag{4.1}$$

$$\frac{d\mathbf{z}_2(t)}{dt} = f(\mathbf{z}_2(t), t), \quad \mathbf{z}_2(t_1) = \mathbf{x}_2. \tag{4.2}$$

Also assume there is a constant $C \geq 0$ so that

$$||\mathbf{f}(\mathbf{z}_2(t), t) - \mathbf{f}(\mathbf{z}_1(t), t)|| \leq C||\mathbf{z}_2(t) - \mathbf{z}_1(t)||. \tag{4.3}$$

Then for $t \in [t_1, t_2]$

$$||\mathbf{z}_2(t) - \mathbf{z}_1(t)|| \leq e^{C|t-t_1|}||\mathbf{x}_2 - \mathbf{x}_1||. \tag{4.4}$$

A full proof of the Gronwall inequality can be found in. (Howard, 1998)

Theorem 4.1.3. *For all $t \in [0, T]$, the feature mapping $\phi_t : \mathbb{R}^d \to \mathbb{R}^d$ is a homeomorphism.*

Proof. A homeomorphism function is a continuous bijection that has a continuous inverse function. To prove ϕ_t is a homeomorphism, we need to show that ϕ_t satisfies both three following properties

(1) ϕ_t is continuous:
Consider two initial conditions of the ODE, $\mathbf{z}_1(0) = \mathbf{x}$ and $\mathbf{z}_2(0) = \mathbf{x} + \delta$ with δ is the difference between $\mathbf{z}_1(0)$ and $\mathbf{z}_2(0)$. Using Gronwall inequality, we have

$$||\mathbf{z}_2(t) - \mathbf{z}_1(t)|| \leq e^{Ct}||\mathbf{z}_1(0) - \mathbf{z}_2(0)|| = e^{Ct}||\delta||, \tag{4.5}$$

with C is a constant, $C \geq 0$. Rewriting in terms of $\phi_t(\mathbf{x})$, we get

$$||\phi_t(\mathbf{x} + \delta) - \phi_t(\mathbf{x})|| \leq e^{Ct}||\delta||. \tag{4.6}$$

If setting $\delta \to 0$, then $\phi_t(\mathbf{x})$ is continuous in $\mathbf{x}$ for all $t \in [0, T]$.

(2) ϕ_t is a bijection:
To prove ϕ_t is a bijection,we need to show that ϕ_t is both onto and one-to-one. Because $\phi_t : \mathbb{R}^d \to \mathbb{R}^d$, it is onto itself. We have to prove that $\phi_t(\mathbf{x}_1) \neq \phi_t(\mathbf{x}_2)$ for each $\mathbf{x}_1 \neq \mathbf{x}_2$.

Suppose there exists initial conditions $\mathbf{x}_1 \neq \mathbf{x}_2$ such that $\phi_t(\mathbf{x}_1) = \phi_t(\mathbf{x}_2)$. We also define an IVP system starting from $\phi_t(\mathbf{x}_1)$ backwards to time $t = 0$. Since satisfying the existence and uniqueness condition, its solution is unique, so $\mathbf{x}_1 = \mathbf{x}_2$, leading our supposition is false. This means that $\phi_t(\mathbf{x}_1) \neq \phi_t(\mathbf{x}_2)$ for each $\mathbf{x}_1 \neq \mathbf{x}_2$, then ϕ_t is one-to-one.

(3)ϕ_t^{-1} is continuous:
The inverse of ϕ_t is ϕ_{-t}, then $\phi_t^{-1} = \phi_{-t}$. To prove ϕ_{-t} is continuous, we also define an IVP backwards in time and use the Gronwall inequality as the proof in part (a). We have

$$||\phi_{-t}(\mathbf{x} + \delta) - \phi_{-t}(\mathbf{x})|| \leq e^{-Ct}||\delta|| = \frac{1}{e^{Ct}}||\delta||, \tag{4.7}$$

with C is a constant, $C \geq 0$. Let setting $\delta \to 0$, then ϕ_{-t} is continuous.

In summary, ϕ_t satisfies both three above properties, so ϕ_t is a homeomorphism. □

Disadvantages: Because θ_t is a homeomorphism, this leads to a chain of disadvantages that are proved in studies of homeomorphisms. The features of Neural ODEs preserve the topology of the input space. This means that Neural ODEs can only deform the input space and cannot tear a connected region apart (Dupont et al., 2019). For example, it is not only difficult to learn a good approximation but also numerically expensive to solve for a binary classification problem where not existing a hyperplane between two labeled input sets.

4.2 Functions Neural ODEs-Net cannot Represent

4.2.1 Not increasing Functions in One-dimensional Space

In this section, we will demonstrate that there is no ODE for representing a non-increasing function $h : \mathbb{R} \to \mathbb{R}$. Therefore, Neural ODEs cannot learn the class of this function.

Proposition 4.2.1. *Neural ODEs-Net cannot represent a not increasing function* $h : \mathbb{R} \to \mathbb{R}$.

Proof. There exists $x_0, y_0 \in \mathbb{R}$ such that $x_0 > y_0$ and $h(x_0) < h(y_0)$. We will prove that two trajectories corresponding to $x_0 \mapsto h(x_0)$ and $y_0 \mapsto h(y_0)$ must cross each other.

Suppose there is an $\mathbf{g}$ such that two trajectories $z_1(t)$ and $z_2(t)$ with $t \in [0, T]$ where

$$\begin{cases} z_1(0) = x_0 & z_1(T) = h(x_0) \\ z_2(0) = y_0 & z_2(T) = h(y_0) \end{cases} \tag{4.8}$$

We define $z(t) = z_1(t) - z_2(t)$. Since $z_1(t)$ and $z_2(t)$ are solutions of the IVP, they are continuous [ref-Coddington]. Because both $z_1(t)$ and $z_2(t)$ are continuous, so $z(t)$ is also continuous. We have $z(0) = x_0 - y_0 > 0$ and $z(T) = h(x_0) - h(y_0) < 0$, so $z(0).z(T) < 0$. By the Intermediate Value Theorem, there exists some $\tilde{t} \in [0, T]$ where $z(\tilde{t}) = 0$, i.e where $z_1(\tilde{t}) = z_2(\tilde{t})$. Therefore, $z_1(t)$ and $z_2(t)$ intersect. However, section 4.1.1 proved that trajectories in Neural ODEs-Net cannot intersect. Thus, Neural ODEs-Net cannot represent non-increasing functions. □

4.2.2 Classes of Functions in d-dimensional Space

Proposition 4.2.2. *Neural ODEs-Net cannot represent* $g(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}$

$$\begin{cases} g(\mathbf{x}) = -1 & \text{if} \quad ||\mathbf{x}|| < r_1 \\ g(\mathbf{x}) = 1 & \text{if} \quad r_1 \leq ||\mathbf{x}|| \leq r_2 \end{cases} \tag{4.9}$$

in which $0 < r_1 < r_2$.

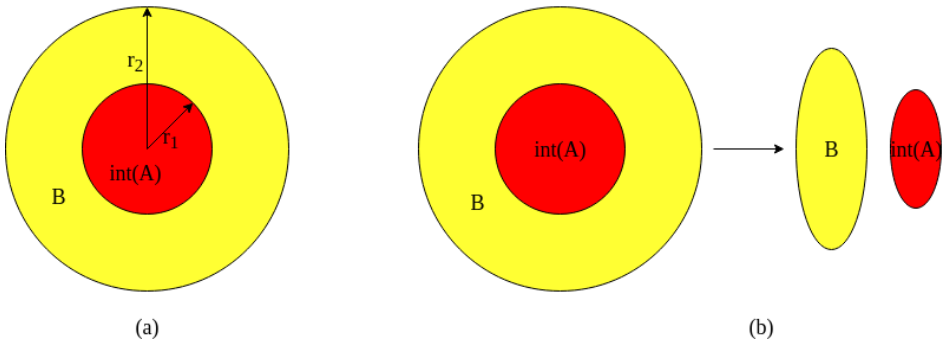

Figure 4.2: (a) Diagram of $g(\mathbf{x})$ in 2-dimentional space. (b) An example of the feature mapping $\phi(\mathbf{x})$ from input data to features.

Proof. Let define a disk $A = \{\mathbf{x} \in \mathbb{R}^d : ||\mathbf{x}|| \leq r_1\}$ where $g(\mathbf{x}) = -1$ with boundary $\partial A = \{\mathbf{x} \in \mathbb{R}^d : ||\mathbf{x}|| = r_1\}$ and interior $\text{int}(A) = \{\mathbf{x} \in \mathbb{R}^d : ||\mathbf{x}|| < r_1\}$, and let $B = \{\mathbf{x} \in \mathbb{R}^d : r_1 \leq ||\mathbf{x}|| \leq r_2\}$ denotes the annulus where $g(\mathbf{x}) = -1$. To map points in $\text{int}(A)$ to -1 and points in B to $+1$ by using Neural ODEs-Net, the linear layer as describing in 3.3.2 must map the features of $\phi(\text{int}(A))$ to -1 and the features of $\phi(B)$ to $+1$. This means that $\phi(\text{int}(A))$ and $\phi(B)$ must be linearly separable. We will prove that it is impossible with ϕ is a homeomorphism.

We have $\partial A \subset B$, so all points in ∂A map to $+1$ and all points in $\text{int}(A)$ map to -1. So $\phi(\text{int}(A))$ and $\phi(B)$ are linearly separable if and only if $\phi(\text{int}(A))$ and $\phi(\partial A)$ are linearly separable.

Through a homeomorphism, points on boundary will map to points on boundary and point in the interior to points in the interior (Armstrong, 1979). This means $\phi(\text{int}(A)) = \text{int}(\phi(A))$ and $\phi(\partial A) = \partial(\phi(A))$ through the feature transformation ϕ. Therefore, we need to show that it is

impossible for $\text{int}(\phi(A))$ and $\partial(\phi(A))$ separate linearly.

Suppose there exists a hyperplane such that all points of $\text{int}(\phi(A))$ and $\partial(\phi(A))$ are located on opposite sides of this hyperplane. This means that there is a linear function $G(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ and a constant C such that $G(\mathbf{x}) > C$ for all $\mathbf{x} \in \partial(\phi(A))$ and $G(\mathbf{x}) < C$ for all $\mathbf{x} \in \text{int}(\phi(A))$. Because A is a connected subset and ϕ is a homeomorphism, so $\phi(A)$ is also a connected subset. Thus, each point $\mathbf{x} \in \text{int}(\phi(A))$ can be rewritten as a combination of points in the boundary $\partial(\phi(A))$. So $\mathbf{x} = \alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2$ with $\mathbf{x}_1, \mathbf{x}_2 \in \partial(\phi(A))$ and $0 < \alpha < 1$. We have

$$\begin{aligned} G(\mathbf{x}) &= \mathbf{w}^T\mathbf{x} \\ &= \mathbf{w}^T(\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2) \\ &= \alpha\mathbf{w}^T\mathbf{x}_1 + (1-\alpha)\mathbf{w}^T\mathbf{x}_2 \\ &\geq \alpha C + (1-\alpha)C \\ &= C. \end{aligned}$$

Since $G(\mathbf{x}) = C$, all points of both $\text{int}(\phi(A))$ and $\partial(\phi(A))$ are in the same side of the hyperplane. Therefore, $\phi(\text{int}(A))$ and $\phi(B)$ are not linearly separable. This implies that Neural ODEs-Net cannot represent $g(\mathbf{x})$. □

In summary, Neural ODEs-Net cannot represent the two above functions. It is so clear that Neural ODEs-Net is not a universal approximator. This is a serious problem because any paucity of performance can remove the need for using Neural ODEs-Net. The next sections show methods to improve Neural ODEs-Net.

4.3 Neural ODEs-Net with Extra Dimensions

The previous section explained that Neural ODEs-Net is not universal approximation, it cannot always represent every function. To address these limitations of Neural ODEs-Net, a simple solution is to lift the model to a higher dimensional space from $\mathbb{R}^d$ to $\mathbb{R}^{d+p}$. This approach is presented in (Dupont et al., 2019) by defining a function $\mathbf{u}(t) \in \mathbb{R}^p$ in the additional part of the space. It achieves better generalization and lower losses than the pure Neural ODEs-Net.

$$\frac{d}{dt}\begin{bmatrix}\mathbf{z}(t)\\ \mathbf{u}(t)\end{bmatrix} = f\left(\begin{bmatrix}\mathbf{z}(t)\\ \mathbf{u}(t)\end{bmatrix}, t\right), \qquad \begin{bmatrix}\mathbf{z}(0)\\ \mathbf{u}(0)\end{bmatrix} = \begin{bmatrix}\mathbf{x}_0\\ \mathbf{0}\end{bmatrix} \tag{4.10}$$

The data point $\mathbf{x}_0$ is concatenated with a vector of q zeros and the ODE is solved on the augmented space $\mathbb{R}^{d+p}$. The trajectories may not intersect in this augmented space when p is large enough.

4.4 Neural ODEs-Net with Evolutionary Parameters

In the original Neural ODEs-Net (Chen et al., 2018), the weight θ is independent on time t. We have

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), \theta, t), \tag{4.11}$$

with θ is fixed over time, the model is easier to train, but it leads to the lack of flexibility. Limiting the number parameters might be the cause of decreasing the approximation capacity. To solve this problem, a natural way is to allow evolution of the neural network parameters. We has form

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), \theta(t), t), \tag{4.12}$$

where, $\theta(t)$ depends on time t. Similarly to $\mathbf{z}(t)$, we use another Neural ODEs-Net to model the evolution of $\theta(t)$. With initial condition $\theta(0) = \theta_0$, we assume

$$\frac{\mathrm{d}\theta(t)}{\mathrm{d}t} = g(\theta(t), \omega, t), \tag{4.13}$$

with ω is a parameter to characterize the weight network.
We can model simply the evolution of both weights $\theta(t)$ and activation $\mathbf{z}(t)$ with a coupled system of ODEs by the following formulation

$$\begin{cases} \mathbf{z}(T) = \mathbf{z}(0) + \int_0^T f(\mathbf{z}(t), \theta(t), t)dt, & \mathbf{z}(0) = \mathbf{x} \quad \text{“Activation network”} \\ \theta(t) = \theta(0) + \int_0^t g(\theta(t), \omega, t)dt, & \theta(0) = \theta_0 \quad \text{“Weight network”} \end{cases} \tag{4.14}$$

in which, $\mathbf{x}$ is the input, ω is fixed over time. In this new version of Neural ODEs-Net, we will use a system of coupled ODEs: one ODE for the activations evolving in time and one for the model parameters. We will fix a value for ω instead of fixing parameter θ like the original version in (Chen et al., 2018). If $g = 0$, then it is exactly the original Neural ODEs-Net with fixed weights.

Zhang et al. (2019) also investigates a formulation that is sightly general than the formulation defined by 4.14. An auxiliary dynamic system is defined for $w(t)$ which is used to compute $\theta(t)$.

In particular, we have a constrained optimization problem

$$\min_{p, w_0} \mathcal{J}(\mathbf{z}(T)) = \frac{1}{N} \sum_{i=1}^{N} l(\mathbf{z}(T); x_i, y_i) + R(w_0, p) \tag{4.15}$$

subject to

$$\begin{aligned} \frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} &= f(\mathbf{z}(t), \theta(t), t), \qquad \mathbf{z}(0) = \mathbf{z}_0 \qquad \text{”Activation ODE”}, \\ \frac{\mathrm{d}w(t)}{\mathrm{d}t} &= g(w(t), p, t), \qquad w(0) = w_0 \qquad \text{”Evolution ODE”}, \\ \theta(t) &= \int_0^t K(t-\tau) w(\tau) \mathrm{d}\tau, \end{aligned}$$

with (x_i, y_i) is the i^{th} training sample and its label, R is a regularization operator and K is a time convolution kernel. To perform backpropagation for this formulation, we need first to form the Lagrangian operator as the following:

$$\begin{aligned} \mathcal{L} = \mathcal{J}(\mathbf{z}(T)) &+ \int_0^T \alpha(t) \left(\frac{\mathrm{d}z}{\mathrm{d}t} - f(\mathbf{z}(t), \theta(t)) \right) \mathrm{d}t + \int_0^T \beta(t) \left(\frac{\partial w}{\partial t} - g(w(t), p, t) \right) \mathrm{d}t \\ &+ \int_0^T \gamma(t) \left(\theta(t) - \int_0^t K(t-\tau) w(\tau) \mathrm{d}\tau \right) \mathrm{d}t, \end{aligned}$$

in which $\alpha(t)$, $\beta(t)$ and $\gamma(t)$ are the adjoint variables. Then, we also use the adjoint sensitive methods to find derivatives in the backward phrase. The derivations of $\mathcal{L}$ with respect to z, θ, w, w_0 and p as below:

$$\frac{\partial \mathcal{J}(\mathbf{z}_T)}{\partial \mathbf{z}_T} + \alpha_T = 0, \quad -\frac{\partial \alpha}{\partial t} - \left(\frac{\partial f}{\partial \mathbf{z}} \right)^T \alpha(t) = 0; \qquad (\partial \mathcal{L}_\mathbf{z}) \tag{4.16}$$

$$-\left(\frac{\partial f}{\partial \theta} \right)^T \alpha(t) + \gamma(t) = 0; \qquad (\partial \mathcal{L}_\theta) \tag{4.17}$$

$$-\frac{\partial \beta(t)}{\partial t} - \left(\frac{\partial q}{\partial w} \right)^T \beta(t) - \int_t^T K^T(\tau - t) \gamma(\tau) \mathrm{d}\tau = 0, \beta(T) = 0; \qquad (\partial \mathcal{L}_w) \tag{4.18}$$

$$-\beta(0) + \frac{\partial R}{\partial w_0} = g_{w_0}; \qquad (\partial \mathcal{L}_{w_0}) \tag{4.19}$$

$$\frac{\partial R}{\partial p} - \int_0^T \left(\frac{\partial q}{\partial p} \right)^T \beta(t) \mathrm{d}(t) = g_p; \qquad (\partial \mathcal{L}_p), \tag{4.20}$$

in which $(.)^T$ is the transpose.

In the forward phrase, given w_0 and p, we compute $w(t)$, then $\theta(t)$ can be computed. By using $\theta(t)$, we can find the activation $\mathbf{z}(t)$.

In the backward phrase, we solve the first adjoint equation for $\alpha(t)$ with the initial condition $\alpha_T = -\frac{\partial \mathcal{J}(\mathbf{z}_T)}{\partial \mathbf{z}_T}$, in which $\mathbf{z}_T$ is computed from the continuous forward propagation. Then, we use the computed $\alpha(t)$ to find the second adjoint variable $\gamma(t)$. Using the computed $\gamma(t)$, we can solve the third adjoint equation to get values of $\beta(t)$. By plugging $\beta(t)$ in Equation 4.19 and 4.20, we can compute the gradients g_p and g_{w_0}.

Chapter 5

Experimental Results

We apply Neural ODEs for image classification, with using the CIFAR-10 dataset. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. The dataset is divided into ten classes. The classes are mutual exclusive which means there is no overlap among ten classes. There are 50000 training images and 10000 test images, in which contains 195 train batches and 78 test batches. They will be trained with Stochastic Gradient Descent algorithm.

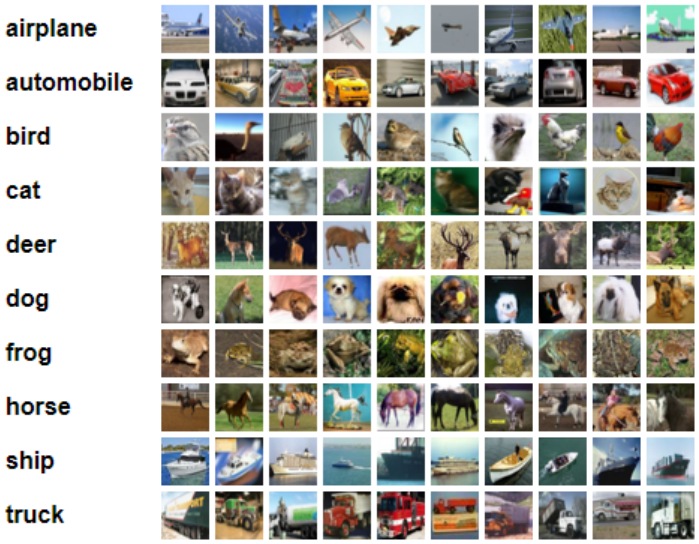

Figure 5.1: Ten classes in CIFAR-10 dataset and ten images from each.

Source: https://www.cs.toronto.edu/ kriz/cifar.html

We perform experiments on CIFAR-10 dataset using three models: the original Neural ODEs-Net, Neural ODEs-Net (NODEs) with Extra Dimensions and Neural ODEs-Net with Evolutionary Parameters, in which we use convolutional architectures for $f(\mathbf{h}(t), t, \theta)$. The input $\mathbf{x}$ is an image, which has dimensionality of $32 \times 32 \times 3$. In NODEs models, we perform experiments with extra dimension $p = 1$ and $p = 5$, that makes the input images have dimensionality of $32 \times 32 \times (3 + d)$.

As mentioned in above section, NODEs with extra dimensions is an extension of neural ODEs, in which we solve ordinary differential equations in $\mathbb{R}^{d+p}$ instead of $\mathbb{R}^d$.

Experimental results for models are shown in figure 5.2. As we can see, two extensions of Neural ODEs models obtain lower loss and train faster than original Neural ODEs model. For the ANODEs with extra dimensions, the higher the value of p is, the less time we need to get a certain loss. It is also expected that the higher the value of p, the better model we received. However, in figure 5.3, it is shown that the very high value of p causes over-fitting problems.

In figure 5.2(a), it is easy to see that five extra dimensions model achieve the best training loss, but in figure 5.2(b) and figure 5.3(c), validation losses increase rapidly, that means this model may be over-fitting. While the neural ODEs model's training losses are quite unstable, that they increase

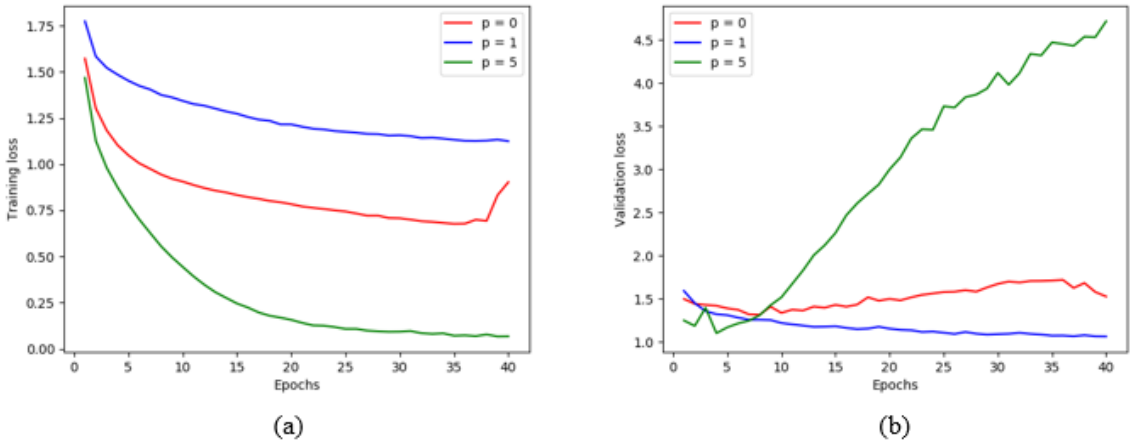

Figure 5.2: Training loss and validation loss for original model and augmented models on CIFAR-10 dataset. (a) Training losses (b) Validation losses. Note that p indicates the numbers of augmented dimensions, so $p = 0$ indicates the original neural ODEs-Net model.

suddenly from epoch 38.

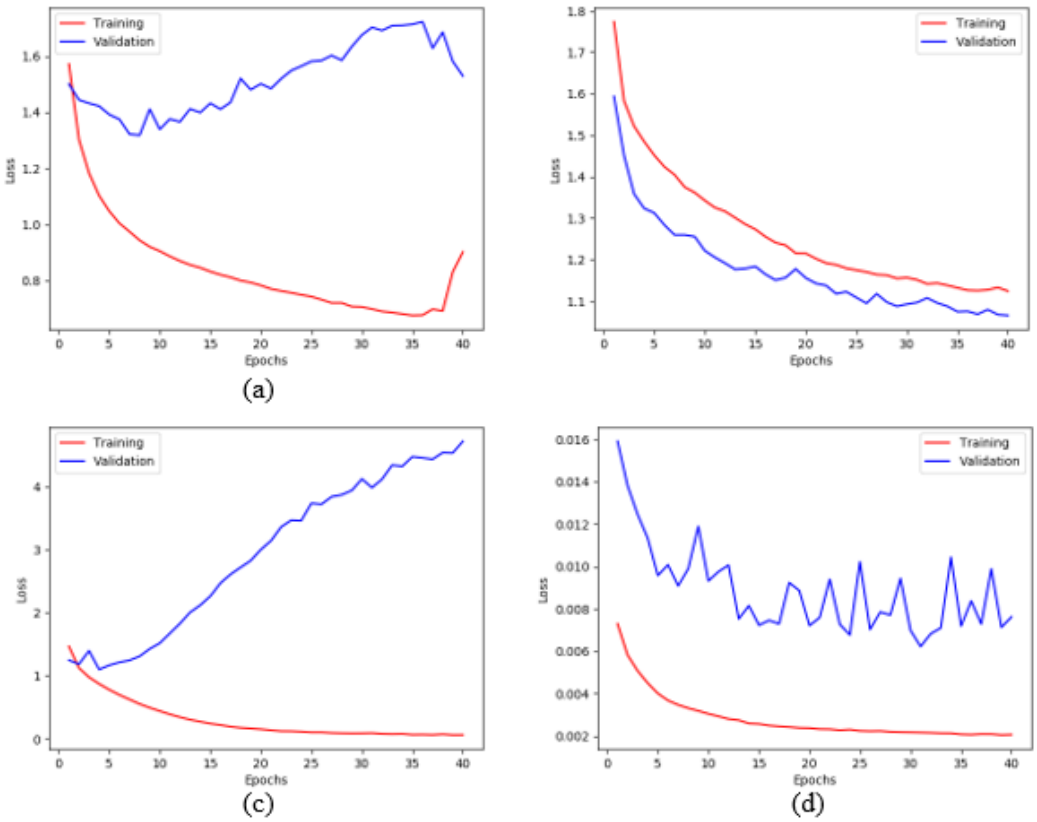

Figure 5.3: Training loss and validation loss for original model and augmented models on CIFAR-10 dataset. (a) Neural ODEs Model (b) NODEs with Evolutionary Parameters Model.

Another extension of neural ODEs-Net is NODEs with evolutionary parameters, in which parameter $\theta(t)$ depends on time variable t. Figure 5.4 shows the experimental results for comparing training loss between the original neural ODEs-Net and NODEs-Net with evolutionary parameters. The model of NODEs with evolutionary parameters gives better training losses compared to the original model.

Table 5.1 compares test accuracy among original neural ODEs model and its extension models, which

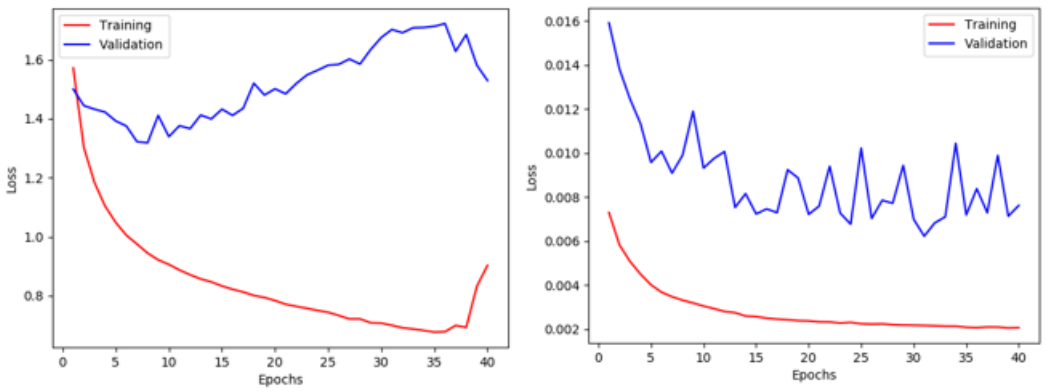

Figure 5.4: Training loss and validation loss for original model and augmented models with evolutionary parameters on CIFAR-10 dataset. (a) Training losses, (b) Training losses of NODEs with evolutionary model, (c) Validation losses, (d) Validation losses of NODEs with evolutionary model.

	Min	**Max**	**Average**
Original Neural ODEs-Net	42.81%	65.63%	51.76%
NODEs-Net with Extra Dimensions $p = 1$	56.25%	69.92%	62.27%
NODEs-Net with Extra Dimensions $p = 5$	40.63%	71.88%	55.45%
NODEs-Net with Evolutionary Parameters	76.92%	77.45%	77.30%

Table 5.1: Test accuracy for CIFAR-10 dataset.

includes minimum, maximum and average accuracy. We perform testing through test sets and receive the table of result.

The NODEs with augmented dimensions model and the NODEs with evolutionary parameters model achieve higher accuracy than the original model; especially, the NODEs with one extra dimension and the NODEs with evolutionary parameters gives very high results. The NODEs with five extra dimensions gives high accuracy, but it is an over-fitting model, that leads to a skew-learn situation. The NODEs with evolutionary parameters performs the best accuracy, and the other extension models also achieve quite good result that can compare to other networks as feedforwarn neural networks or residual neural networks.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

In summary, Neural ODEs-Net was a breakthrough in the development of deep learning. There are three main things we presented in this thesis.

Firstly, we presented about Neural ODEs-Net. In which, we identify clearly and definitely its architecture, its learning process, and how to apply it for a supervised learning problem, after recalling some background about ODE and a general neural network. Problems in neural networks led the

Secondly, properties of Neural ODEs-Net were pointed out clearly that led to the strengths and weaknesses in using it. Defining and implementing neural networks using ODE solvers brought benefits in effective memory and took advantage of adaptive computing when ODE solvers were developed more than 120 years.

Thirdly, two extensions of Neural ODEs-Net that improved both performance and representation ability of Neural ODEs-Net were presented in this thesis. One of them was to lift the original model to a higher dimensional space. The other was about the evolutionary of parameters.

Lastly, our experiments showed that the extensions models learn better then the original neural ODEs-Net model and achieve higher accuracy. However, it is also bring many problems, which we can improve models to solve in the future.

6.2 Future Works

Neural ODEs-Net still has many limitations that need to be improved to be widely applied in practice. In the future we wish to improve the following two things:

The training time: During the training, we found that Neural ODEs-Net has a very long training time compared to Resnets. This is a huge obstacle to bringing Neural ODEs-Net to practical problems, although the low memory usage is an outstanding advantage of Neural ODEs-Net. Finlay et al. (2020) showed it to be quite possible. However, we still need to improve Neural ODE-Net such that it is competitive to other existing models.

The representation ability: The original Neural ODEs-Net cannot represent some functions such as the non-increasing function which has intersected trajectories. Thus, Neural ODEs-Net is not a universal approximator. A new promising result which is proved that it is a universal approximator was introduced by Kidger et al. (2020) with providing additional theoretical results. This is the continuous analogue of an RNN that subsumes apparently-similar ODE models in which the vector field depends directly upon continuous data.

Appendix A

The Adjoint Sensitive Method

A.1 A Proof of the Adjoint State

We define a function $\lambda(t)$ called *adjoint state* by:

$$\lambda(t) = \frac{dL}{d\mathbf{z}(t)} \tag{A.1}$$

We will prove that the differential equation of the above function respect to t is

$$\frac{d\lambda(t)}{dt} = -\lambda(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}} dt \tag{A.2}$$

With a continuous hidden state, let define

$$\mathbf{z}(t+\varepsilon) = \mathbf{z}(t) + \int_t^{t+\varepsilon} f(\mathbf{z}(t), t, \theta)dt = T_\varepsilon(\mathbf{z}(t), t), \tag{A.3}$$

in which ε is an change in time. After applying chain rule, the results will be

$$\frac{dL}{d\mathbf{z}(t)} = \frac{dL}{d\mathbf{z}(t+\varepsilon)}\frac{d\mathbf{z}(t+\varepsilon)}{d\mathbf{z}(t)} \qquad \text{or equivalently,} \qquad \lambda(t) = \lambda(t+\varepsilon)\frac{\partial T_\varepsilon(\mathbf{z}(t), t)}{\partial \mathbf{z}(t)} \tag{A.4}$$

Using Taylor series around $\mathbf{z}(t)$, we have:

$$\mathbf{z}(t+\varepsilon) = T_\varepsilon(\mathbf{z}(t), t) = \mathbf{z}(t) + \varepsilon f(\mathbf{z}(t), t, \theta) + O(\varepsilon^2) \tag{A.5}$$

Therefore,

$$\begin{aligned}
\frac{d\lambda(t)}{dt} &= \lim_{\varepsilon \to 0^+} \frac{\lambda(t+\varepsilon) - \lambda(t)}{\varepsilon} && \text{(A.6)}\\
&= \lim_{\varepsilon \to 0^+} \frac{\lambda(t+\varepsilon) - \lambda(t+\varepsilon)\frac{\partial T_\varepsilon(\mathbf{z}(t), t)}{\partial \mathbf{z}(t)}}{\varepsilon} && \text{(A.7)}\\
&= \lim_{\varepsilon \to 0^+} \frac{\lambda(t+\varepsilon) - \lambda(t+\varepsilon)\frac{\partial}{\partial \mathbf{z}(t)}\big(\mathbf{z}(t) + \varepsilon f(\mathbf{z}(t), t, \theta) + O(\varepsilon^2)\big)}{\varepsilon} && \text{(A.8)}\\
&= \lim_{\varepsilon \to 0^+} \frac{\lambda(t+\varepsilon) - \lambda(t+\varepsilon)\big[1 + \varepsilon\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + O(\varepsilon^2)\big]}{\varepsilon} && \text{(A.9)}\\
&= \lim_{\varepsilon \to 0^+} -\lambda(t+\varepsilon)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + O(\varepsilon) && \text{(A.10)}\\
&= -\lambda(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)}, && \text{(A.11)}
\end{aligned}$$

A.2 Computing Gradients for Backpropagation

In this section, we will explain how $\frac{dL}{d\mathbf{z}(t)}$ and $\frac{dL}{d\theta}$ look like.

$$\frac{dL}{d\mathbf{z}(t_0)} = \lambda(t_0) \tag{A.12}$$

$$= \lambda(t_1) + \int_{t_1}^{t_0} \frac{d\lambda(t)}{dt} dt \tag{A.13}$$

$$= \lambda(t_1) - \int_{t_1}^{t_0} \lambda(t) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} \tag{A.14}$$

Note that parameters θ independent on t, this means θ is fixed over the time, so

$$\frac{\partial \theta(t)}{\partial t} = \mathbf{0} \quad and \quad \frac{dt(t)}{dt} = \mathbf{1} \tag{A.15}$$

Similarly, we define $\lambda_\theta(t) = \frac{dL}{d\theta(t)}$ with setting $\lambda_\theta(t_1) = 0$, then prove that

$$\frac{dL}{d\theta} = \lambda_\theta(t_0) \tag{A.16}$$

$$= \lambda_\theta(t_1) - \int_{t_1}^{t_0} \lambda(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt \tag{A.17}$$

$$= -\int_{t_1}^{t_0} \lambda(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt \tag{A.18}$$

Bibliography

W. Ang and Y. Park. *Ordinary differential equations: methods and applications*. Universal-Publishers, 2008.

T. Archibald, C. Fraser, and I. Grattan-Guinness. The history of differential equations, 1670–1950. *Oberwolfach Reports*, 1(4):2729–2794, 2005.

M. Armstrong. *Basic Topology*. McGraw-Hill Book Company, 1979. ISBN 9780070840904. URL https://books.google.com.vn/books?id=x0PvAAAAMAAJ.

R. L. Burden and D. J. Faires. Numerical analysis. 1985.

B. Chang, L. Meng, E. Haber, L. Ruthotto, D. Begert, and E. Holtham. Reversible architectures for arbitrarily deep residual neural networks. *arXiv preprint arXiv:1709.03698*, 2017.

R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud. Neural ordinary differential equations. *Advances in Neural Information Processing Systems*, 2018.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

E. Dupont, A. Doucet, and Y. W. Teh. Augmented neural odes. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 3140–3150. Curran Associates, Inc., 2019. URL http://papers.nips.cc/paper/8577-augmented-neural-odes.pdf.

C. Finlay, J.-H. Jacobsen, L. Nurbekyan, and A. M. Oberman. How to train your neural ode: the world of jacobian and kinetic regularization, 2020.

A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse. The reversible residual network: Backpropagation without storing activations. In *Advances in neural information processing systems*, pages 2214–2224, 2017.

D. F. Griffiths and D. J. Higham. *Numerical methods for ordinary differential equations: initial value problems*. Springer Science & Business Media, 2010.

E. Haber and L. Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1): 014004, 2017.

E. Hairer. Sp n rsett, and g. wanner. solving ordinary differential equations i. nonsti problems, 1987.

E. Hairer, S. Norsett, and G. Wanner. Solving ordinary, differential equations i, nonstiff problems/e. hairer, sp norsett, g. wanner, with 135 figures, vol.: 1. Technical report, 2Ed. Springer-Verlag, 2000, 2000.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

K. Hornik, M. Stinchcombe, H. White, et al. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

K. Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, 3(5):551–560, 1990.

R. Howard. The gronwall inequality. 1998.

P. Kidger, J. Morrill, J. Foster, and T. Lyons. Neural controlled differential equations for irregular time series, 2020.

E. Kreyszig. Advanced engineering mathematics, 10th eddition, 2009.

W. Kutta. Beitrag zur naherungsweisen integration totaler differentialgleichungen. *Z. Math. Phys.*, 46:435–453, 1901.

Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, pages 21–28. CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988.

M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

H. Lin and S. Jegelka. Resnet with one-neuron hidden layers is a universal approximator. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6169–6178. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/7855-resnet-with-one-neuron-hidden-layers-is-a-universal-approximator.pdf.

Y. Lu, A. Zhong, Q. Li, and B. Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. In *International Conference on Machine Learning*, pages 3276–3285. PMLR, 2018.

R. Mattheij and J. Molenaar. *Ordinary differential equations in theory and practice*. SIAM, 2002.

T. M. Mitchell. *The discipline of machine learning*, volume 9. Carnegie Mellon University, School of Computer Science, Machine Learning . . . , 2006.

B. A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural networks*, 6(5):1212–1228, 1995.

B. V. G. R. Pontryagin LS, Mishchenko E. The mathematical theory of optimal processes. 1962.

C. Runge. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2): 167–178, 1895.

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

G. F. Simmons. *Differential equations with applications and historical notes*. CRC Press, 2016.

T. Zhang, Z. Yao, A. Gholami, K. Keutzer, J. Gonzalez, G. Biros, and M. Mahoney. Anodev2: A coupled neural ode evolution framework. *arXiv preprint arXiv:1906.04596*, 2019.