UNIVERSIDAD DE SONORA

FACULTAD INTERDISCIPLINARIA DE CIENCIAS EXACTAS Y NATURALES

Programa de Licenciatura en Matemáticas

Numerical Solution of Ordinary and Delay Differential Equations by Neural Network Methods

T E S I S

Que para obtener el título de:

Licenciado en Matemáticas

Presenta:

Javier Calvo Quirrín

Director de tesis: M.C. Eddel Elí Ojeda Avilés

Hermosillo, Sonora, México, 23 de junio de 2025

SINODALES

Dr. Saúl Díaz-Infante Velasco
SECIHTI-Universidad de Sonora, Hermosillo, México

Dra. Carolina Espinoza Villalva
Universidad de Sonora, Hermosillo, México

M.C. Eddel Elí Ojeda Avilés
Universidad de Sonora, Hermosillo, México

Dr. Daniel Olmos Liceaga
Universidad de Sonora, Hermosillo, México

A mis padres . . .

Agradecimientos

En primer lugar, quiero agradecer profundamente mi familia, quienes siempre me han brindado su apoyo. En especial, quiero agradecer a mis padres, quiero agradecer su esfuerzo y dedicación a lo largo de toda mi vida. Gracias por siempre confiar en mi a lo largo de mis estudios, por los valores que me inculcaron y por estar presentes en todo momento. Este logro es tan mío como suyo.

Agradezco a mi director de tesis el M.C. Eddel Elí Ojeda Avilés y al Dr. Daniel Olmos Liceaga, gracias a ambos por su paciencia y orientación que me brindaron para la elaboración de este trabajo. Esta tesis no hubiera sido posible sin su dedicación constante.

También, quiero agradecer a mis sinodales, por el tiempo que dedicaron para revisar este trabajo. Valoro enormemente sus valiosas recomendaciones y comentarios que han enriquecido este trabajo.

Table of Contents

Chapter 1

Introduction

The study of Ordinary Differential Equations is of great interest. Many problems in science and engineering need this tool to answer important questions in their field. For example, in economics, the Solow-Swan model, which was developed independently by Robert Solow and Trevor Swan in 1956, aims to explain the long-term economic growth of a country by considering the accumulation of capital and labor growth. This economic model is characterized by a single ordinary differential equation. Another application of differential equations is in the neuroscience field, with the Hodgkin-Huxley equations. This mathematical model, developed by Alan Hodgkin and Andrew Huxley in 1952, describes how action potentials are initiated and propagated in neurons [18]. In chemical reaction dynamics, the Brusselator, developed by Ilya Prigogine and René Lefever in 1968, is a two-component mathematical model that utilizes differential equations for the study of autocatalytic reactions [14]. The most well-known example of the usage of these equations is the modeling of the oscillatory behavior coming from the Belousov–Zhabotinsky reaction.

In general, there are no mathematical techniques that solve such equations exactly. There are other methods, such as qualitative analysis, asymptotic analysis, or numerical approximations like Euler's method presented by Leonhard Euler in the 18th century or the Runge-Kutta methods, first presented by Carl Runge in 1895 and further developed by Wilhelm Kutta in 1901. The case of numerical approximations basically tells us that there are plenty of numerical schemes depending on the equation type. For example, when working with stiff differential equations, one may use methods based on backward differentiation formulas, which were first introduced in [11]. In [21], there was

a new proposal about numerical solutions for differential equations. Even though this approach is old, its real usage came in [25]. From here, people turned their eyes again to the methods of solving different types of differential equations using these computational models.
This thesis focuses on understanding the neural network approach for approximating differential equations. We build the essential foundations of the theory of ordinary differential equations and neural networks in the first two chapters of this thesis. This approach ensures that anyone, regardless of their prior knowledge in either of these fields, can follow and understand the method presented in this thesis. This work aims to guide the reader step by step through the concepts from the ground up.
Therefore, this thesis is structured as follows. Chapter 2 is devoted to a small overview of the theory of ordinary differential equations. Readers familiar with the fundamental theory of ordinary differential equations may choose to omit this chapter. We go over elemental topics like the definition of a differential equation and ways to classify them into various categories. We will then proceed to examine the idea of a solution of an ordinary differential equation and introduce some essential definitions involving the solutions of differential equations. Afterwards, we examine an important theorem that gives conditions that guarantee the existence and uniqueness of a solution of a differential equation, along with its proof, which is included in Appendix A of this thesis. Subsequently, we go over some basic theory of delay differential equations. In the last section of this chapter, we review two popular classical methods for approximating the solutions of differential equations.
In Chapter 3, we discuss the fundamental concepts that define the theory of neural networks. First, we examine the structure of neural networks and review the different elements that make up these computational models. Next, we will introduce the concept of activation function, focusing on their role in allowing neural networks to fit nonlinear data. We will also look at some commonly used examples of these functions. Then, we will study the process in which neural networks learn, starting with defining the concept of loss function, used to quantify the performance of a model. Afterwards, the algorithm of backpropagation and the way it is used to optimize a neural network is reviewed. After that, we inspect the gradient descent algorithm and examine its role in the optimization of neural networks. We also provide some examples of gradient descent-based algorithms, along with their corresponding pseudocode, used to minimize the loss function. To conclude this section, we will examine Physics-Informed Neural Networks and an impor-

tant Universal Approximation Theorem for neural networks.
In Chapter 4, the construction and implementation of the Physics-Informed Neural Network method for approximating ordinary differential equations and delay differential equations is presented. We define the respective loss function we seek to minimize to approximate each type of differential equation. After that, we start presenting our examples. Firstly, we show various types of ODEs, which include examples whose solution is known and examples where the solution cannot be found through analytical methods. Lastly, we present multiple examples of DDEs.
Finally, in Chapter 5, we will share the conclusions drawn from our work.

Chapter 2

An Overview of Ordinary Differential Equations

Differential equations are a mathematical tool that relates functions and their derivatives. The implementation of differential equations is essential in areas of applied mathematics because these equations can model the behavior of dynamic systems, and they can also be utilized to describe how quantities change over time. This is why differential equations are used across many fields, ranging from areas of science like physics and biology, and playing a crucial role in engineering. Differential equations allow us to mathematically model many kinds of phenomena, such as population growth, heat distribution, fluid dynamics, the spread of diseases, and so forth. In this chapter, we explore the theoretical foundations of differential equations, with a focus on Ordinary Differential Equations.

2.1 Basic Concepts

Definition of a Differential Equation

A differential equation is a type of equation whose unknown is not a number, but some other function that satisfies that equation. Let us begin by defining what a differential equation is.

Definition 2.1 (Differential Equation)**.** A differential equation **(DE)** is any equation that contains the derivatives of one or more dependent variables, with respect to one or more independent variables.

Even though they are both called equations, differential equations look nothing like algebraic equations. Let us look at a few examples to illustrate how differential equations look like:

$$\begin{aligned}\frac{dy}{dx} + 2y &= x,\\ \frac{dy}{dx} &= y \cdot \sin(x),\\ \frac{dy^2}{dx^2} - 3\frac{dy}{dx} + 2y, &= 0,\\ \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} &= 0,\\ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} &= 0.\end{aligned} \tag{2.1}$$

Although at first glance these equations can look quite daunting, they are one of the most useful resources in mathematics and are used in almost every academic discipline.

Notation for Differential Equations

We can write differential equations in different ways, among which are the following [32]:

- **Leibniz's Notation**

$$\frac{d^2y}{dx^2} - 3\frac{dy}{dx} + 2y = \sin(x).$$

- **Lagrange's Notation**

$$f''(x) - 3f'(x) + 2y = \sin(x).$$

- **D-Notation**

$$D_x^2 y - 3D_x y + 2y = \sin(x).$$

- **Newton's Notation**

$$\ddot{y} - 3\dot{y} + 2y - \sin(x).$$

Please note that each of these past four equations is equal to one another, and it is up to personal and stylistic preference which notation is used when writing.

2.2 Classification of Ordinary Differential Equations

In this section, we look into the classification of differential equations based on their order, type, and linearity. Understanding what kind of differential equation we are dealing with is crucial in order to be able to solve them, because different kinds of differential equations are solved with different methods.

Order of Differential Equations

Definition 2.2 (Order of a Differential Equation)**.** The order of a differential equation is equal to the highest derivative that appears in that equation.

In the example list (2.1), using the definition above, we can classify the first, second, and fourth examples as differential equations of order 1. While the third and fifth examples are both differential equations of order 2.

Types of Differential Equations

We now classify differential equations by their type. There are two types of differential equations, which are Ordinary Differential Equations and Partial Differential Equations.

Definition 2.3 (Type of a Differential Equation)**.** A differential equation is called an Ordinary Differential Equation **ODE** if said equation only has ordinary derivatives in it. Likewise, a differential equation is called a Partial Derivative Equation **PDE** if the equation has partial derivatives in it.

In the example list (2.1), we can see that examples 1-3 are Ordinary Differential Equations, and examples 4 and 5 are Partial Differential Equations.
In this thesis, we will only be working with Ordinary Differential Equations. Therefore, from this point onward, the word ordinary might be omitted from time to time when talking about differential equations.

Linearity of Differential Equations

The final classification category is linearity. In this case, we are working exclusively with ordinary differential equations.

Definition 2.4. (Linearity of a Differential Equation) A differential equation is said to be linear if it has the following form

$$a_0(x)y + a_1(x)\frac{dy}{dx} + a_2(x)\frac{d^2y}{dx^2} + \cdots + a_n(x)\frac{d^ny}{dx^n} = g(x),$$

where $a_0(x), a_1(x), a_2(x), \cdots, a_n(x), g(x)$ are functions of x, and y is the unknown function of the independent variable x.

From this definition, we can also derive two key properties of linear differential equations, which are as follows:

- The coefficients $a_0, a_1, \cdots, a_n$ only depend on the independent variable, in this case x.
- The degree of the dependent variable y and all of its derivatives are of first degree.

Let's look at the following examples and decide whether they are linear or not.

1. $\dfrac{d^2y}{dx^2} + 5\dfrac{dy}{dx} + 6y = 0.$
2. $\dfrac{d^2y}{dx^2} + y^2 = 0.$
3. $(1-y)\dfrac{dy}{dx} + 2y = e^x.$
4. $x^2\dfrac{d^2y}{dx^2} + x\dfrac{dy}{dx} - 2y = \sin(x).$

In example 1, we observe that it is linear, because y and all its derivatives are raised to the power 1, and also the coefficients are all integers. In example 2, we can quickly identify that it is nonlinear because of the y^2 term. In 3, we verify that it is nonlinear, this time because of the $(1-y)$ coefficient that is next to the first derivative of y, since coefficients only have to depend on the independent variable.
Finally, example 4 is also a linear differential equation. Someone may look at the x^2 coefficient and the sine of x, and argue that it cannot be linear since clearly those are not linear. However, as we stated previously, the coefficients accompanying the derivatives only need to depend on the independent variable, which x^2 follows, as well as $\sin(x)$. And since y and all of its derivatives are raised to the first power, we conclude that this differential equation is linear.

2.3 Solving Ordinary Differential Equations

In this section, we look at what it means to solve a differential equation. First, we examine the definition of a solution to a differential equation and what it means to be a solution. Secondly, we explore what initial value problems and boundary value problems are. Finally, we review some methods for solving differential equations.

2.3.1 The Solution of an ODE

To understand what a solution of a differential equation is, we first focus on an example of a second-order algebraic equation.
Let us consider the following equation

$$2x^2 - 4x - 6 = 0. \tag{2.2}$$

Using the quadratic formula, we obtain that the solutions of (2.2) are $x_1 = -1$ and $x_2 = 3$. When we say x_1 and x_2 are the solutions to the equation, we mean that x_1 and x_2 both satisfy the equation. In other words, if we plug either x_1 or x_2 into (2.2), the equality holds.
With that in mind, let us define what a solution of a differential equation is.

Definition 2.5 (Solution of an ODE)**.** A function $y = f(x)$ is said to be a solution of an n-th order differential equation on an interval I, if y has at least n continuous derivatives on I, which when substituted into the nth-order differential equation reduces the equation to an identity.

In mathematical symbols, this definition can be written as: the function $y = f(x)$ is a solution of a differential equation

$$F(x, y, y', \dots, y^{(n)}) = 0,$$

if

$$F[x, f(x), f'(x), \cdots, f^{(n)}] = 0 \qquad \forall x \in I.$$

As can be seen, the concept of a solution to a differential equation is similar to the concept of a solution to an algebraic equation, but instead of finding some real (or complex) number, we look for a function that satisfies the differential equation. Of course, these concepts are not identical, as differential equations have their particular aspects that we will be discussing later in this section.

Example 2.1. Consider the following differential equation:

$$x\frac{d^2y}{dx^2} + \frac{dy}{dx} = 0. \tag{2.3}$$

We will show that

a) $y_1(x) = \ln\left(\frac{1}{x}\right)$ is a solution 2.3.

b) $y_2(x) = x^2$ is not a solution of 2.3.

Firstly, we prove that y_1 is a solution of (2.3). To do that, we calculate the first and second derivatives of y_1, since it is a second-order differential equation:

$$y_1' = -\frac{1}{x} \qquad , \qquad y_1'' = \frac{1}{x^2}.$$

Now, let's substitute the derivatives back into the differential equation:

$$x \cdot y_1'' + y_1' = 0,$$
$$x \cdot \frac{1}{x^2} + \left(-\frac{1}{x}\right) = 0.$$

Simplifying this expression, we obtain:

$$\frac{1}{x} - \frac{1}{x} = 0,$$
$$0 = 0.$$

Since the equality holds even after we substituted the values of the derivatives into the differential equation, we say that y_1 is a solution of (2.3).
Now let's see that y_2 is not a solution of (2.3). Like before, let's find the first and second derivatives of y_2:

$$y_2' = 2x \qquad , \qquad y_2'' = 2.$$

Substituting these derivatives into the differential equation, we get:

$$x \cdot y_2'' + y_2',$$
$$x \cdot 2 + 2x,$$
$$2x + 2x,$$
$$4x \neq 0.$$

In this case, the equality does not hold after we substituted the values into (2.3), and with that, we have proven that y_2 is not a solution to (2.3), just as we desired.

2.3.2 Particular & General Solutions of ODEs

When we study the theory of integration in calculus with the help of the Fundamental Theorem of Calculus, we can solve problems like finding the antiderivative y of a function $f(x)$, which means finding a function y that satisfies $y'(x) = f(x)$. Looking at that expression and using what we have learned up to this point, we can claim that we have actually solved a simple differential equation without even realizing it.
With that in mind, let's recall some of the things that we learned in calculus about integration to understand the concept of an n-parameter family of solutions of differential equations.
Let's look at the following example

$$y' = e^x. \tag{2.4}$$

In (2.4), we use the Fundamental Theorem of Calculus (FTC) and integrate both sides of the equation to obtain the following function

$$y = e^x + c.$$

We consider this function, $y = f(x, c)$, the solution of the simple differential equation (2.4), with c being the constant of integration that can take any numerical value.
Now let's take a look at this next example

$$y'' = e^x. \tag{2.5}$$

This time we have the second derivative of y, integrating twice using once again the FTC and simple integration techniques, we can conclude that the solution of (2.5) is

$$y = e^x + c_1x + c_2.$$

Notice that this time the solution $y = f(x, c_1, c_2)$ has two constants c_1 and c_2, which again can take any numerical value.
Finally, let's take a look at the following equation

$$y''' = e^x. \tag{2.6}$$

This time we have the third derivative of y, so we are solving this equation the same way we solved (2.5), but now we are integrating three times to obtain

$$y = e^x + c_1x^2 + c_2x + c_3.$$

This time the solution of the differential equation $y = f(x, c_1, c_2, c_3)$ has three constants c_1, c_2, c_3 which like in the previous examples can take any numerical value.
With these past three examples in mind, we can come up with two conjectures involving the solutions of differential equations:

1. A differential equation has infinite solutions, as many as the number of values the constants $c_1, c_2, \ldots, c_n$ can take.

2. The number of constants in the solution depends on the order of the differential equation.

From this point onwards, we refer to the constants $c_1, c_2, \ldots, c_n$ as the differential equation solution's n-th parameters.

Definition 2.6 (n-parameter family of solutions of an ODE)**.** When solving the n-th order differential equation

$$F(x, y', y'', \cdots, y^{(n)}) = 0,$$

we seek a function involving $n + 1$ variables $x, c_1, c_2, \cdots, c_n$ with the form

$$y = f(x, c_1, c_2, \cdots, c_n),$$

and we call the function y the n-parameter family of solutions of the n-th order differential equation.

With this definition in mind, we can confirm both of our previously stated conjectures we made involving the solution of a differential equation. A single differential equation can possess an infinite number of solutions, which correspond to the unlimited number of choices the parameters of the solution can take. And when solving an n-th order differential equation, we obtain an n-th order family of solutions to that equation, that is, the solution will have as many parameters as the order of the differential equation.
Looking back at examples (2.4), (2.5), and (2.6), and their respective solutions, we can now conclude that these equations have a 1-parameter, 2-parameter, and 3-parameter family of solutions, respectively.
When we assign numerical values to the parameters of an n-parameter family of solutions, we obtain a particular solution of a differential equation.

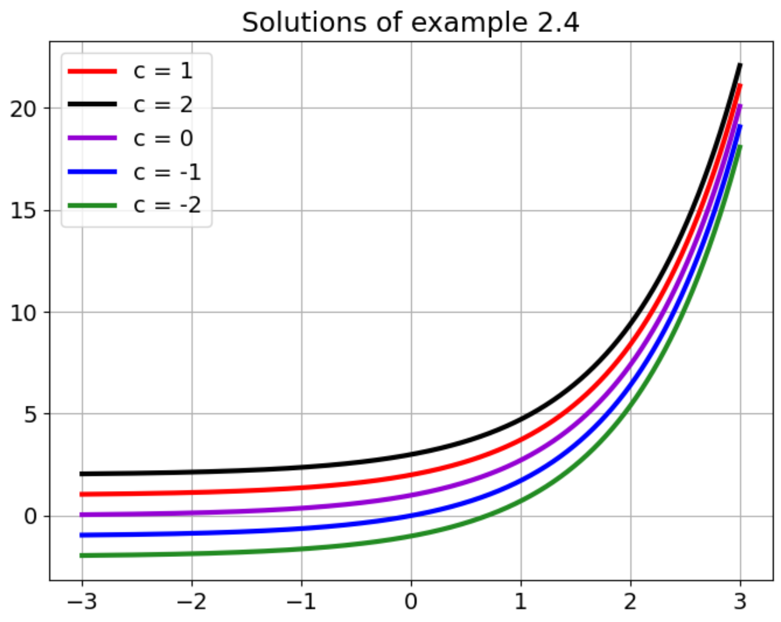


Figure 2.1: Particular solutions of example (2.4) for different values of c.

Definition 2.7 (Particular solution of an ODE)**.** A solution of a differential equation is called a particular solution if it does not contain any arbitrary parameters and satisfies the differential equation.

Now we define what the general solution of a differential equation is.

Definition 2.8 (General solution of an ODE)**.** An n-th parameter family of solutions that solves the n-th order differential equation is called the general solution of the differential equation if it contains every particular solution of the equation.

Let's take a look back at example (2.4) and its solution. We assign some numerical values to the parameter in the solution, and we obtain some particular solutions to that equation. In Figure 2.1, we substituted some values for the parameter of the solution of (2.4) and graphed those functions. Please note that each function plotted in Fig. 2.1 is a solution of (2.4), so that implies that all of them satisfy the differential equation.

Up until this point, we have learned that when solving a differential equation, we obtain what is called a general solution, and this general solution becomes a particular solution when we assign some numerical values to the parameters in the solution. Nonetheless, there exist some equations where the general solution does not describe all the solutions to a differential equation. These types of solutions that are not derived from the general solution are called singular solutions of a differential equation.

Definition 2.9 (Singular solution of an ODE)**.** A singular solution of a differential equation is a solution that cannot be obtained from the general solution of the equation.

2.3.3 Initial Value Problems

When solving differential equations in some applications, we are only interested in a specific solution that satisfies some additional conditions. The requirements we desire our particular solution to follow are called initial conditions. For example, let's say we want our solution to go through the point $(3, -1)$, then we write our initial condition as

$$y(3) = -1.$$

Definition 2.10 (Initial Conditions)**.** Initial conditions are a set of points that the solution of an n-th order differential equation and its first $n-1$ derivatives have to satisfy.

These initial conditions help us determine the values of the parameters from the general solution of a differential equation to obtain a particular solution that satisfies the equation, and also satisfies the given initial conditions.
When we pair an appropriate number of initial conditions with a differential equation, we obtain what is called an Initial Value Problem.

Definition 2.11 (Initial Value Problem)**.** An Initial Value Problem(IVP) is a differential equation along with some initial conditions.

Usually, an IVP has as many initial conditions as the order of the differential equation we want to solve. Based on this last statement, a typical IVP has the following form

$$\begin{array}{ll} \text{Solve:} & F(x, y, y', \ldots, y^{(n)}) = 0 \\ \text{Subject to:} & y(x_0) = y_0, \quad y'(x_0) = y_1, \ldots, \quad y^{(n-1)}(x_0) = y_{n-1}. \end{array} \tag{2.7}$$

When solving an IVP, two important questions involving the solution arise. Firstly, does this IVP have a solution? We would like to know if this initial-value problem actually has a solution before we work on it.
Secondly, if this solution exists, is it unique? This is an important question to consider because if this IVP models a physical system, we would like to know that this is a unique solution before we apply this solution to the system. Otherwise, if this IVP had multiple solutions, how would we know which one to use?
Fortunately, we have a theorem that gives conditions under which an initial value problem has a unique solution.

Theorem 2.1 (Existence and Uniqueness of Solutions [23])
Let E be an open subset of $\mathbb{R}^n$ containing x_0 and assume $f \in C^1(E)$. Then there exists an $a > 0$ such that the initial value problem

$$\begin{aligned} \dot{x} &= f(x) \\ x(0) &= x_0 \end{aligned}$$

has a unique solution $x(t)$ on the interval $[-a, a]$.

Proof. The proof of this theorem can be found in Appendix A. □

2.3.4 Some Analytical Methods to solve ODEs

We look at a few ways we can solve differential equations. We only focus on 2 methods to find the solutions of differential equations, but there are a lot more methods that can be studied. Sometimes, differential equations cannot be solved through analytical methods; therefore, we need the help of numerical methods to approximate the solutions.

Separable Equations

Let's start with the easiest method, that is, solution by separable equations. We first define what it means for a differential equation to be separable [34].

Definition 2.12 (Separable Equation)**.** A first-order differential equation in the form

$$\frac{dy}{dx} = g(x)h(y)$$

is said to be a separable equation.

Let's see how to solve a separable equation. First, let's consider an equation like in the definition.

$$\frac{dy}{dx} = g(x)h(y).$$

Assuming $h(y) = 0\,\forall y$ and dividing each side by $h(y)$ we get

$$\frac{1}{h(y)}\frac{dy}{dx} = g(x).$$

For convenience lets denote $\frac{1}{h(y)}$ as $p(y)$

$$p(y)\frac{dy}{dx} = g(x).$$

Let's suppose $y = \phi(x)$ is a solution of the equation; therefore, it must be true that $p(\phi(x))\phi'(x) = g(x)$. Consequently, integrating both sides with respect to x to obtain

$$\int p(\phi(x))\phi'(x)dx = \int g(x)dx.$$

Notice that $dy = \phi'(x)dx$, so this simply becomes

$$\int p(y)dy = \int g(x)dx.$$

Let's take a look at an example.

Example 2.2. Solve

$$\frac{dx}{dt} = x^2\cos(t).$$

Separating the x and t variables we get

$$\int \frac{1}{x^2}dx = \int \cos(t)dt.$$

By integrating both sides, we obtain

$$-\frac{1}{x} = \sin(t) + c.$$

Solving for x, we get the solutions

$$x = -\frac{1}{\sin(t) + c}.$$

Linear Equations

The next method we examine is for linear differential equations. For this method to work, we need to have a first-order differential equation in the following form

$$\frac{dy}{dx} + p(x)y = g(x), \tag{2.8}$$

where both $p(x)$ and $g(x)$ are continuous functions.
Now that we have our linear equation in this form, we identify $p(x)$ and find the integrating factor $\mu(x) = e^{\int p(x)dx}$.
Then we multiply both sides of 2.8 by the integrating factor [32]

$$e^{\int p(x)dx}\frac{dy}{dx} + e^{\int p(x)dx}p(x)y = e^{\int p(x)dx}g(x).$$

Notice how the left side of this equation is the derivative of the product between $e^{\int p(x)dx}$ and y, therefore

$$\frac{d}{dx}\left[e^{\int p(x)dx}y\right] = e^{\int p(x)dx}g(x).$$

Finally, we simply integrate both sides and solve for y.
Let's look at an example.

Example 2.3. Solve

$$\frac{dy}{dx} - 3y = 0.$$

As we can see, this equation is already in the form 2.8, so $p(x) = -3$. To find the integrating factor you simply solve $e^{\int -3dx}$, which clearly is e^{-3x}. Now we multiply both sides of the differential equation by e^{-3x}

$$e^{-3x}\frac{dy}{dx} - 3e^{-3x}y = e^{-3x} \cdot 0 = e^{-3x}.$$

Notice how the left side of this last equation can be written as

$$\frac{d}{dx}\left[e^{-3x}y\right] = 0.$$

Integrating both sides of this equation

$$\int \frac{d}{dx}\left[e^{-3x}y\right]dx = \int 0dx.$$

We obtain that

$$e^{-3x}y = c,$$

and solving for y yields

$$y = ce^{3x}.$$

2.4 Systems of Differential Equations

Up until this point, we have only worked with single differential equations. In this section, we learn about systems of differential equations that involve two or more differential equations. The main reason for studying these systems is that many of the "real world" situations we are interested in can be modeled using systems of differential equations. One of these problems is modeling the population of two species, predators and prey. With the help of a system of differential equations, we can make a prediction about the population of these two species that changes with time, given certain biological parameters. Let's define what a system of first-order differential equations is [34].

Definition 2.13 (System of n first order differential equations)**.** The system of n equations

$$\begin{aligned} \frac{dy_1}{dt} &= g_1(y_1, y_2, \cdots, y_n, t), \\ \frac{dy_2}{dt} &= g_2(y_1, y_2, \cdots, y_n, t), \\ &\vdots \\ \frac{dy_n}{dt} &= g_n(y_1, y_2, \cdots, y_n, t), \end{aligned} \tag{2.9}$$

where $g_1, \cdots, g_n$ are functions of $y_1, y_2, \cdots, y_n, t$, defined on a common set S is called a system of n first order differential equations.

The solution of a system of differential equations is not a single function, but rather a set of functions.

Definition 2.14 (Solution of a system of differential equations)**.** A solution of the system of n first order differential equations is a set of functions $y_1(t), y_2(t), y_3(t), \cdots, y_n(t)$, each defined on a common interval $I \subseteq S$, satisfying all the equations in the system.

If the functions $g_1, g_2, \cdots, g_n$ in definition 2.13 are all linear, we refer to it simply as a linear system. In this particular case, the system will have the following form:

$$\begin{aligned}
\frac{dy_1}{dt} &= a_{11}(t)y_1 + a_{12}(t)y_2 + \cdots + a_{1n}(t)y_n + f_1(t),\\
\frac{dy_2}{dt} &= a_{21}(t)y_1 + a_{22}(t)y_2 + \cdots + a_{2n}(t)y_n + f_2(t),\\
&\vdots\\
\frac{dy_n}{dt} &= a_{n1}(t)y_1 + a_{n2}(t)y_2 + \cdots + a_{nn}(t)y_n + f_n(t).
\end{aligned} \tag{2.10}$$

For simplicity, lets assume that the coefficients a_{ij} and the functions f_i with $i = j = 1, 2, \cdots, n$ are all continuous on a common interval I.
Another way of expressing the linear system 2.10 is by using matrices. The first matrix has the derivatives of the n differential equations in the system.

$$\mathbf{Y} = \begin{pmatrix} y_1(t)\\ y_2(t)\\ y_3(t)\\ \vdots\\ y_n(t) \end{pmatrix}.$$

The next matrix will have the coefficients a_{ij}, where $i, j \in 1, 2, 3, \cdots, n$

$$\mathbf{A(t)} = \begin{pmatrix} a_{11}(t) & a_{12}(t) & \cdots & a_{1n}(t)\\ a_{21}(t) & a_{22}(t) & \cdots & a_{2n}(t)\\ a_{31}(t) & a_{32}(t) & \cdots & a_{3n}(t)\\ \vdots & & \ddots & \\ a_{n1}(t) & a_{n2}(t) & \cdots & a_{nn}(t) \end{pmatrix}.$$

The last matrix will have the functions f_i, where $i \in 1, 2, 3, \cdots, n$

$$\mathbf{F(t)} = \begin{pmatrix} f_1(t)\\ f_2(t)\\ f_3(t)\\ \vdots\\ f_n(t) \end{pmatrix}.$$

With the help of these 3 matrices, the linear first-order differential equations system 2.10 can be written as

$$\begin{pmatrix} \dot{y}_1(t) \\ \dot{y}_2(t) \\ \dot{y}_3(t) \\ \vdots \\ \dot{y}_n(t) \end{pmatrix} = \begin{pmatrix} a_{11}(t) & a_{12}(t) & \cdots & a_{1n}(t) \\ a_{21}(t) & a_{22}(t) & \cdots & a_{2n}(t) \\ a_{31}(t) & a_{32}(t) & \cdots & a_{3n}(t) \\ \vdots & & \ddots & \\ a_{n1}(t) & a_{n2}(t) & \cdots & a_{nn}(t) \end{pmatrix} \begin{pmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \\ \vdots \\ y_n(t) \end{pmatrix} + \begin{pmatrix} f_1(t) \\ f_2(t) \\ f_3(t) \\ \vdots \\ f_n(t) \end{pmatrix},$$

or just simply

$$\dot{\mathbf{Y}} = \mathbf{AY} + \mathbf{F}.$$

We say a system is homogeneous if $\mathbf{F} \equiv 0$, this is

$$\dot{\mathbf{Y}} = \mathbf{AY}. \tag{2.11}$$

In this thesis, we only focus on systems of two linear homogeneous differential equations with constant coefficients, like the following

$$\begin{aligned} \dot{y}_1 &= ay_1 + by_2, \\ \dot{y}_2 &= cy_1 + dy_2. \end{aligned} \tag{2.12}$$

We call a system like 2.12 a coupled system because we need the information of y_2 in order to find y_1, and vice versa, we need the information of y_1 in order to find y_2.
To solve 2.12, let's first look at an example.

Example 2.4. Verify that

$$\mathbf{Y_1} = \begin{pmatrix} 1 \\ -1 \end{pmatrix} e^{-2t} \quad \text{and} \quad \mathbf{Y_2} = \begin{pmatrix} 3 \\ 5 \end{pmatrix} e^{6t}$$

are solutions of the system

$$\dot{\mathbf{Y}} = \begin{pmatrix} 1 & 3 \\ 5 & 3 \end{pmatrix} \mathbf{Y}. \tag{2.13}$$

First lets get the derivatives of both $\mathbf{Y_1}$ and $\mathbf{Y_2}$

$$\dot{\mathbf{Y}}_\mathbf{1} = \begin{pmatrix} -2e^{-2t} \\ 2e^{-2t} \end{pmatrix} \quad \text{and} \quad \dot{\mathbf{Y}}_\mathbf{2} = \begin{pmatrix} 18e^{6t} \\ 30e^{6t} \end{pmatrix}.$$

Now lets find $\mathbf{AY_1}$ and $\mathbf{AY_2}$

$$\mathbf{AY_1} = \begin{pmatrix} 1 & 3 \\ 5 & 3 \end{pmatrix} \begin{pmatrix} e^{-2t} \\ -e^{-2t} \end{pmatrix} = \begin{pmatrix} e^{-2t} - 3e^{-2t} \\ 5e^{-2t} - 3e^{-2t} \end{pmatrix} = \begin{pmatrix} -2e^{-2t} \\ 2e^{-2t} \end{pmatrix} = \dot{\mathbf{Y}}_\mathbf{1},$$

and

$$\mathbf{AY_2} = \begin{pmatrix} 1 & 3 \\ 5 & 3 \end{pmatrix} \begin{pmatrix} 3e^{6t} \\ 5e^{6t} \end{pmatrix} = \begin{pmatrix} 3e^{6t} + 15e^{6t} \\ 15e^{6t} + 15e^{6t} \end{pmatrix} = \begin{pmatrix} 18e^{6t} \\ 30e^{6t} \end{pmatrix} = \dot{\mathbf{Y}}_\mathbf{2}.$$

Therefore $\mathbf{Y_1}$ and $\mathbf{Y_2}$ are solutions of the system.

It is also the case that if we multiply by any constant c a solution of a system, it will remain a solution of the system. Also, if we add two solutions of a system, the resulting vector will also be a solution. Let's state the following theorem [6].

Theorem 2.2
Let $\mathbf{Y_1}$ and $\mathbf{Y_2}$ be two solutions of the system $\dot{\mathbf{Y}} = \mathbf{AY}$. Then:

(a) $c\mathbf{Y_1}$ is a solution, for any $c \in R$.

(b) $\mathbf{Y_1} + \mathbf{Y_2}$ is a solution.

Proof.

(a) If $\mathbf{Y_1}$ is a solution of $\dot{\mathbf{Y}} = \mathbf{AY}$, then

$$\frac{d}{dt} c\mathbf{Y_1} = c\frac{d\mathbf{Y_1}}{dt} = c\mathbf{AY_1} = \mathbf{A}(c\mathbf{Y_1}).$$

Therefore, $c\mathbf{Y_1}$ is also a solution of $\dot{\mathbf{Y}} = \mathbf{AY}$.

(b) If $\mathbf{Y_1}$ and $\mathbf{Y_2}$ are solutions of $\dot{\mathbf{Y}} = \mathbf{AY}$ then

$$\frac{d}{dt}(\mathbf{Y_1} + \mathbf{Y_2}) = \frac{d\mathbf{Y_1}}{dt} + \frac{d\mathbf{Y_2}}{dt} = \mathbf{AY_1} + \mathbf{AY_2} = \mathbf{A}(\mathbf{Y_1} + \mathbf{Y_2}).$$

Hence, $\mathbf{Y_1} + \mathbf{Y_2}$ is also a solution of the system.

□

An immediate corollary of this theorem is that any linear combination of solutions of 2.11 is also a solution of the system.

Corollary 2.2.1
If $\mathbf{Y_1}, \mathbf{Y_2}, \ldots, \mathbf{Y_n}$ *are solutions of the homogeneous system* $\dot{\mathbf{Y}} = \mathbf{AY}$*, then* $c_1\mathbf{Y_1} + c_2\mathbf{Y_2} + \cdots + c_n\mathbf{Y_n}$ *is again a solution of the system for any choice of constants* $c_1, c_2, \ldots, c_n$.

Proof.
Immediately follows from Theorem 2.2. □

With this corollary, we can define what the general solution of a homogeneous system is.

Definition 2.15 (General Solution of Homogeneous Systems)**.** Let $\mathbf{Y_1}, \mathbf{Y_2}, \ldots, \mathbf{Y_n}$ be a set of linearly independent solutions of the system 2.11. We define the general solution $\mathbf{Y}$ of the system as

$$\mathbf{Y} = c_1\mathbf{Y_1} + c_2\mathbf{Y_2} + \cdots + c_n\mathbf{Y_n},$$

where $c_1, c_2, \ldots, c_n$ are arbitrary constants.

Looking back at example 2.13, we know that both $\mathbf{Y_1} = \begin{pmatrix} 1 \\ -1 \end{pmatrix} e^{-2t}$ and $\mathbf{Y_2} = \begin{pmatrix} 3 \\ 5 \end{pmatrix} e^{6t}$ are solutions of the system. Then, the general solution of 2.13 is

$$\mathbf{Y} = c_1\mathbf{Y_1} + c_2\mathbf{Y_2} = c_1 \begin{pmatrix} 1 \\ -1 \end{pmatrix} e^{-2t} + c_2 \begin{pmatrix} 3 \\ 5 \end{pmatrix} e^{6t}.$$

This general solution has the form

$$\mathbf{Y}_i = \begin{pmatrix} k_1 \\ k_2 \end{pmatrix} e^{\lambda_i t} \qquad i = 1, 2$$

where k_1, k_2, λ_1 and λ_2 are constants.
One problem that comes to mind is whether we can always find a solution to 2.11 with the form

$$\mathbf{Y} = \mathbf{K}e^{\lambda t},$$

where $\mathbf{K}$ is a vector of n constants.
Lets suppose $\mathbf{Y} = \mathbf{K}e^{\lambda t}$ is a solution to 2.11, then

$$\lambda\mathbf{K}e^{\lambda t} = \mathbf{AK}e^{\lambda t}.$$

We divide each side by $e^{\lambda t}$ and we get

$$\mathbf{AK} = \lambda\mathbf{K}.$$

Which anyone with a basic knowledge of linear algebra identifies as the eigenvalue problem. This is in order to find the solutions of 2.11, we must find the eigenvalues and eigenvectors of the matrix $\mathbf{A}$. The way we can find the eigenvalues of a matrix is by using the characteristic equation of $\mathbf{A}$

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0.$$

For the two-equation system 2.12, the characteristic equation of the matrix of coefficients is

$$\det(\mathbf{A}-\lambda\mathbf{I}) = \begin{vmatrix} a-\lambda & b \\ c & d-\lambda \end{vmatrix} = (a-\lambda)(d-\lambda)-bc = \lambda^2-(a+d)\lambda+ad-bc.$$

When calculating the eigenvalues of the matrix $\mathbf{A}$, we are basically solving an n-th degree algebraic equation. So when finding the eigenvalues of a matrix, we have three options involving these values:

1. Real Eigenvalues,
2. Complex Eigenvalues,
3. Repeated Eigenvalues.

Let's look at an example of an initial-value problem where the matrix of coefficients $\mathbf{A}$ has real, distinct eigenvalues.

Example 2.5. Solve

$$\begin{aligned} \dot{x}_1 &= x_1 + 2x_2, \\ \dot{x}_2 &= 3x_1 + 2x_2, \end{aligned}$$

with

$$x_1(0) = 0, x_2(0) = -4.$$

First, let's transform the equations into matrix form

$$\dot{\mathbf{X}} = \begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \text{with} \quad \mathbf{X}(0) = \begin{pmatrix} 0 \\ -4 \end{pmatrix}.$$

The next step we do is to calculate the eigenvalues of the matrix with the help of the characteristic equation

$$\lambda^2 - (1+2)\lambda + (1)(2) - (2)(3) = 0,$$

$$\lambda^2 - 3\lambda - 4 = 0,$$

$$(\lambda + 1)(\lambda - 4) = 0.$$

Therefore the eigenvalues are $\lambda_1 = -1$ and $\lambda_2 = 4$. Next, we find the eigenvector corresponding to each eigenvalue. Lets start with λ_1 :

$$(\mathbf{A} - \lambda_1 \mathbf{I})\boldsymbol{\alpha} = \mathbf{0},$$

$$(\mathbf{A} + \mathbf{I})\boldsymbol{\alpha} = \mathbf{0},$$

$$\begin{pmatrix} 2 & 2 \\ 3 & 3 \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Solving for α_1 and α_2 we get

$$\alpha_1 = -\alpha_2.$$

If we take $\alpha_2 = 1$ the eigenvector associated to λ_1 is

$$\begin{pmatrix} -1 \\ 1 \end{pmatrix}.$$

Now, let's find the eigenvector associated to λ_2:

$$(\mathbf{A} - \lambda_2 \mathbf{I})\boldsymbol{\alpha} = \mathbf{0},$$

$$(\mathbf{A} - 4\mathbf{I})\boldsymbol{\alpha} = \mathbf{0},$$

$$\begin{pmatrix} -3 & 2 \\ 3 & -2 \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Solving for α_1 and α_2

$$\alpha_1 = \frac{2}{3}\alpha_2.$$

If we take $\alpha_2 = 3$ the eigenvector associated to λ_2 is

$$\begin{pmatrix} 2 \\ 3 \end{pmatrix}.$$

Then the general solution of the system is

$$\mathbf{X} = c_1 e^{-t}\begin{pmatrix}-1\\1\end{pmatrix} + c_2 e^{4t}\begin{pmatrix}2\\3\end{pmatrix}.$$

Now we use the given initial condition to find c_1 and c_2

$$\begin{pmatrix}0\\-4\end{pmatrix} = \mathbf{X}(0) = c_1\begin{pmatrix}-1\\1\end{pmatrix} + c_2\begin{pmatrix}2\\3\end{pmatrix}.$$

In equation form, this is

$$\begin{aligned}-c_1 + 2c_2 &= 0,\\ c_1 + 3c_2 &= -4.\end{aligned}$$

Solving for both c_1 and c_2

$$c_1 = -\frac{8}{5}, c_2 = -\frac{4}{5}.$$

Therefore, the solution of the initial-value problem is

$$\mathbf{X} = -\frac{8}{5}e^{-t}\begin{pmatrix}-1\\1\end{pmatrix} - \frac{4}{5}e^{4t}\begin{pmatrix}2\\3\end{pmatrix}.$$

Rewriting Equations as Systems

We can rewrite an n-th order differential equation into an equivalent system of differential equations of first order. One of the main reasons to do this is that it is often easier to work with a bunch of first-order differential equations than a single differential equation of order n. We say the n-th order differential equation and the system of equations are equivalent because they both share the same set of solutions. Please note that the reverse is not always possible; we cannot always write a system of n-th differential first order differential equations into a single n-th order differential equation.
The main idea is to define n functions and "chain" the derivatives and make a change of variables. Let's take a look and see how it's done [6].
Suppose we are given the n-th order linear differential equation

$$a_n(t)\frac{d^n y}{dt^n} + a_{n-1}(t)\frac{d^{n-1}y}{dt^{n-1}} + \cdots + a_0(t)y = 0. \tag{2.14}$$

We define the following functions

$$\begin{aligned} x_1 &= y, \\ x_2 &= \frac{dy}{dt}, \\ x_3 &= \frac{d^2y}{dt^2}, \\ &\vdots \\ x_n &= \frac{d^{n-1}y}{dt^{n-1}}. \end{aligned}$$

Now we take the derivative with respect to t on all the last n equations on both sides of them. Then

$$\begin{aligned} \frac{dx_1}{dt} &= \frac{dy}{dt}, \\ \frac{dx_2}{dt} &= \frac{d^2y}{dt^2}, \\ \frac{dx_3}{dt} &= \frac{d^3y}{dt^3}, \\ &\vdots \\ \frac{dx_n}{dt} &= \frac{d^ny}{dt^n}. \end{aligned}$$

Notice how the derivative of x_1 is really x_2, and the derivative of x_2 is x_3, and so forth. Also observe how the derivative of x_n is the n-th derivative of y, this is the highest order derivative in 2.14. Therefore, if we isolate the $\frac{d^ny}{dy^n}$ term in 2.14, we can determine the value of the derivative of x_n. With this in mind

$$\begin{aligned} \frac{dx_1}{dt} &= x_2, \\ \frac{dx_2}{dt} &= x_3, \\ \frac{dx_3}{dt} &= x_4, \\ &\vdots \\ \frac{dx_n}{dt} &= -\frac{a_{n-1}x_n + a_{n-2}x_{n-1} + \cdots + a_1x_2 + a_0x_1}{a_n}. \end{aligned}$$

And with this, we have transformed the n-th order differential equation 2.14 into a system of first-order differential equations just as we desired.
Let's take a look at an example to make this clear.

Example 2.6. Rewrite the following 2^{nd} order differential equation into a system of first-order differential equations

$$2y'' - 5y' + y = 0. \tag{2.15}$$

First, let's define the following functions

$$\begin{aligned} x_1(t) &= y(t), \\ x_2(t) &= y'(t). \end{aligned}$$

Differentiating both equations, we get

$$\begin{aligned} x_1' &= y' = x_2, \\ x_2' &= y''. \end{aligned}$$

Notice how y'' is really the original equation 2.15. Then the 2^{nd} order differential equation has been rewritten as a differential equation system

$$\begin{aligned} x_1' &= x_2, \\ x_2' &= \frac{5}{2}x_2 - \frac{1}{2}x_1. \end{aligned}$$

2.5 Delay Differential Equations

Delay differential equations (DDEs) are a type of differential equation in which the derivative of the unknown function at the current time depends on the value of the function at previous times. In contrast with ODEs, which only take into consideration the present state of the system when calculating the current value, DDEs incorporate the history of the system into the calculation of the value at any given time. The reasoning for the study of DDEs is that many processes in biology, chemistry, engineering, etc., involve some sort of natural delay. For instance, animals must first digest their food; only then do their bodies show a response. Therefore, focusing only on ODEs and ignoring these intrinsic delays within a system can lead to inaccurate predictions and a misunderstanding of the system's behavior.

The general form of an initial value problem of a delay differential equation is expressed by:

$$\begin{aligned} &y'(t) = f(t, y(t-\tau_1), \dots, y(t-\tau_n)), \\ &y(t) = \phi(t),\ t \leq t_0, \end{aligned} \tag{2.16}$$

where $\tau_1, \dots, \tau_n$ are the delays (or lags) of the system. It is important to note that these delays are always non-negative, and can be classified into three categories [5]:

- constant delay : τ_i is a constant value,$\tau_i = k$,
- time dependent delay : τ_i is a function of t, $\tau_i = \tau_i(t)$,
- state dependent delay : τ_i is a function of t and y, $\tau_i = \tau_i(t, y(t))$.

Unlike ordinary differential equation initial value problems which only need an initial condition for t_0, when solving delay differential equation initial value problems an initial condition function, that depends on t, that spans the entire interval $[-\tau, t_0]$ is needed. We denote this initial condition function as $\phi(t)$ in IVP 2.16.
In this thesis, we only work with DDEs with a single constant delay; this is

$$\begin{aligned} &y'(t) = f(t, y(t), y(t-\tau)), \\ &y(t) = \phi(t),\ t \in [-\tau, t_0]. \end{aligned} \tag{2.17}$$

2.6 Classical Numerical Methods for Ordinary Differential Equations

It is not always the case that we can find an exact solution to a differential equation. In fact, most differential equations cannot be solved analytically. For these kinds of equations, it is a common practice to approximate the solutions with some computational help. When used in industry, differential equations are usually solved numerically as modeling of real-world phenomena leads to equations that are too complicated to solve analytically. In this section, we look into some ways we can obtain the solution of a differential equation with the help of computer algorithms.
The methods we look at in this section are called finite difference methods.

For these kinds of methods, we discretize the solution interval we are interested in. The reason for this is that a computer clearly cannot approximate a function on an entire continuous interval $[a, b]$ since that would require approximating the solution at an infinite number of points.
What we do instead is determine numerical approximations at discrete times $t_0 < t_1 < t_2 < \cdots < t_n$ in the interval we're interested in. To do this we divide the interval $[a, b]$ into n small segments of constant length

$$h = \frac{b-a}{n} = t_{i+1} - t_i \quad \forall i = 0, 1, 2, \ldots, n.$$

We call h the step-size we will use to approximate the solution to our differential equation. With the help of the step-size, we can build a set of equally spaced discrete times of $[a, b]$, this is $a = t_0 < t_1 < t_2 < \cdots < t_n = b$.

Euler's Method

Euler's method is the simplest and easiest approximation method for solving initial value problems. The objective is to approximate the solution of the first-order initial value problem [7]

$$y' = f(t, y) \quad , \quad y(t_0) = y_0. \tag{2.18}$$

Let's suppose that $y(t) \in C^2[a, b]$ is the unique solution of 2.18. With the help of Taylor's Theorem, for each $i = 0, 1, 2, \ldots, n$ we have

$$y(t_{i+1}) = y(t_i) + (t_{i+1} - t_i)y'(t_i) + \frac{(t_{i+1} - t_i)^2}{2} y''(\zeta_i),$$

for some number $\zeta_i \in (t_i, t_{i+1})$. Since $h = t_{i+1} - t_i$, then

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2} y''(\zeta_i).$$

Due to $y(t)$ being the solution of 2.18 it satisfies the differential equation, therefore

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2} y''(\zeta_i).$$

Euler's method constructs $y_i \approx y(t_i) \quad \forall i = 1, 2, \cdots, n$, by deleting the remainder term. Finally, Euler's method is given by

$$y_{i+1} = y_i + hf(t_i, y_i) \qquad \forall i = 0, 1, 2, \ldots, n-1. \tag{2.19}$$

Runge-Kutta Methods

The next family of methods we look at is the Runge-Kutta methods. These kinds of methods are designed to imitate the Taylor series method without requiring analytic differentiation of the original differential equation.
The Runge-Kutta methods are a generalization of the Euler method, but this time, instead of only calculating the slope at a single point, we use a weighted average of slopes in the interval $t_i \leq t \leq t_{i+1}$. This is

$$y_{i+1} = y_i + h \sum_{n=1}^{m} w_n k_n.$$

Here w_i are constants used to weigh the slope, and each k_i is the function $f(t, y)$ evaluated at a point. The number m is called the order of the Runge-Kutta method.
The most widely used method of the Runge-Kutta family is the Runge-Kutta method of order 4, or simply the RK4 method. But there exist other methods like the Runge-Kutta methods of order 2 and order 3. The formulas for the "classical" RK4 method are the following:

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \qquad \forall i = 0, 1, 2, \ldots, n,$$

where

$$\begin{aligned}
k_1 &= f(t_i, y_i),\\
k_2 &= f\left(t_i + \frac{h}{2}, y_i + h\frac{k_1}{2}\right),\\
k_3 &= f\left(t_i + \frac{h}{2}, y_i + h\frac{k_2}{2}\right),\\
k_4 &= f(t_i + h, y_i + hk_3).
\end{aligned}$$

We skipped the derivation of the order 4 Runge-Kutta method as it is quite tedious and out of the scope of this thesis, but it can be found in Ralston and Rabinowitz [26]. In the formulas above, y_{i+1} is the approximation of the solution y at t_{i+1}, this is $y(t_{i+1})$, which is determined by the present value y_i plus the weighted average of k_1, k_2, k_3 and k_4, where

- k_1 is the slope at the beginning of the interval, using y,

- k_2 is the slope at the midpoint of the interval, using y and k_1,
- k_3 is the slope at the midpoint of the interval, using y and k_2,
- k_4 is the slope at the end of the interval, using y and k_3.

Example 2.7. Use Euler's and RK4 methods with $h = 0.1$ to approximate the following IVP at the indicated value

$$y' = 2x - 3y + 1, \quad y(1) = 5. \tag{2.20}$$

To measure how effective each of the methods are, let's use the absolute error, which is defined by

$$\text{absolute error} = |\text{exact value} - \text{approximate value}|.$$

It is not always the case that we know the exact value, but because it is an IVP that has an exact solution that can be found by analytical methods, let's use it to compare how our numerical methods hold.

x_n	**Exact Value**	**Euler**	**RK4**	**Absolute Error Euler**	**Absolute Error RK4**
1	5.0000	5.0000	5.0000	0.0000	0.0000
1.1	3.9723	3.8000	3.9724	0.1723	0.0001
1.2	3.2283	2.9800	3.2284	0.2483	0.0001
1.3	2.6944	2.4260	2.6945	0.2684	0.0001
1.4	2.3161	2.0582	2.3162	0.2579	0.0001
1.5	2.0532	1.8207	2.0533	0.2325	0.0001

Table 2.1: Approximated values and absolute errors of IVP 2.20 with Euler and RK4 methods.

As can be observed in Table 2.1 and Fig. 2.2, the results are overwhelming. Even with a relatively "big" step-size of $h = 0.1$ and with a precision of four decimal points, the absolute errors of the RK4 method are extremely small compared to the absolute errors of Euler's method. This means the RK4 approximations are very close to the real value; therefore, the Runge-Kutta method of order 4 is a very good way to approximate the solution of an initial value problem.

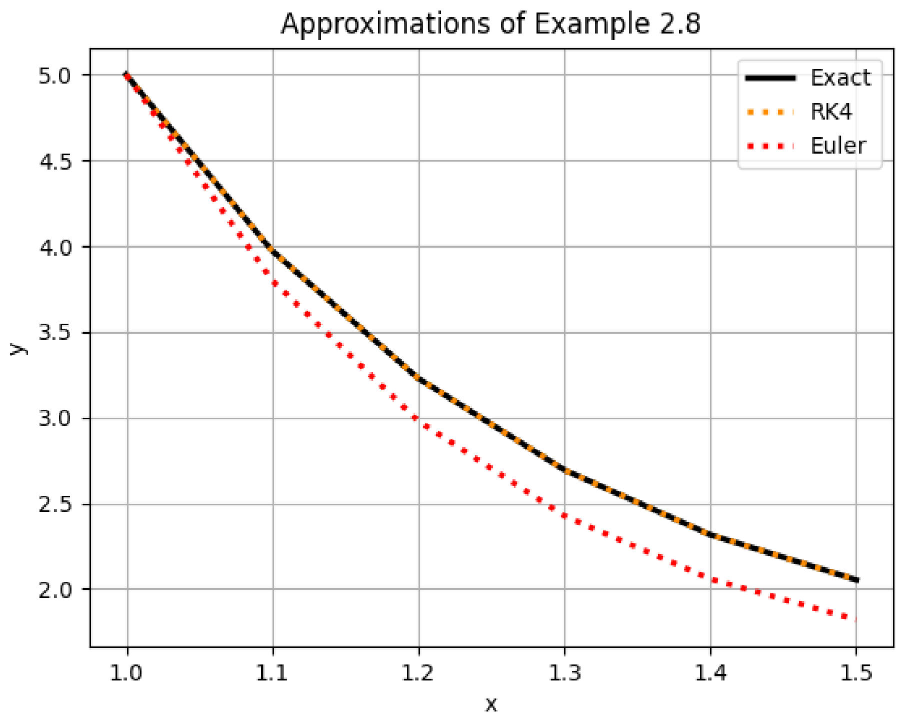


Figure 2.2: Plot of the exact solution and approximations with Euler and RK4 methods of IVP 2.20.

2.7 Applications of Ordinary Differential Equations

As we have stated previously, ordinary differential equations are useful when we want to model real-life phenomena in many academic fields such as physics, chemistry, biology, etc. In this section, we look into a couple of ways differential equations are used to help understand certain problems that arise from disciplines other than mathematics.

Lotka-Volterra Equations

The Lotka-Volterra equations are a pair of first-order differential equations used to describe the changes of a biological system in which two species interact, the predator and the prey. The populations of the two species at any time are given by positive parameters α, β, γ and δ, which have their own biological meaning. The model is formed with these two equations:

$$\frac{dx}{dt} = \alpha x - \beta xy,$$
$$\frac{dy}{dt} = -\gamma y + \delta xy,$$

where:

- x is the population of the prey,
- y is the population of the predator,
- t represents time,
- α represents the maximum prey per capita growth rate,
- β represents the effect of the presence of the predators on the prey death rate,
- γ represents the predator's per capita death rate,
- δ represents the effect of the presence of prey on the predator's growth rate.

If we know what the populations of the two species are at a certain time, let's say $t = 0$, and the values of the parameters, then we can use our model to find out how the number of prey and predators will behave over time. We look at an example to see how this model works.

Example 2.8. Let's suppose we have an environment consisting of only two species, rabbits (prey) and foxes (predator), and that their initial population is 40 and 15, respectively. Let's assume the parameters $\alpha, \beta, \gamma, \delta$ of our environment are: 1.0, 0.1, 1.0, and 0.05, respectively.
Using the information given in the last statement and the Lotka-Volterra equations, we can form a model that describes how the population of our two species will behave as time changes. Let's denote with the variable R the number of rabbits and F the number of foxes. Then, the Lotka-Volterra equations for our example are:

$$\begin{aligned} \frac{dR}{dt} &= 1.0R - 0.1RF \quad R(0) = 40, \\ \frac{dF}{dt} &= -1.0F + 0.05RF \quad F(0) = 15. \end{aligned} \tag{2.21}$$

Since it is a fairly simple system, we can use a basic method like Euler's Method to solve our system. With a step-size of $h = 0.001$, we obtain a good solution.

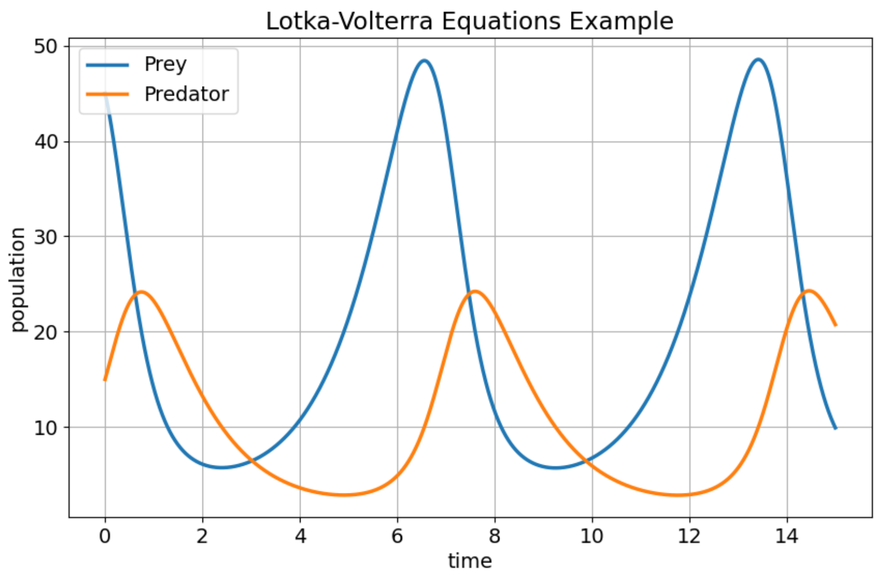


Figure 2.3: Population of rabbits and foxes of 2.21 with Euler's Method.

Newton's Law of Cooling

In physics, Newton's law of cooling is used to describe the rate at which a body loses heat through radiation. This rate of heat loss is directly proportional to the difference in the temperatures between the body and the ambient temperature [8]. Newton's law of cooling is given by the following first-order differential equation:

$$\frac{dT}{dt} = k(T - T_m), \tag{2.22}$$

where:

- T is the temperature of the object,
- k is a constant of proportionality,

- T_m is the ambient temperature,
- t is time.

The greater the difference between the temperature of the object and its surroundings, the faster the body temperature changes. We look at an example to see how this law is used.

Example 2.9. Suppose we are trying to cool down a batch of cooking oil to store it. The oil was heated to a temperature of 180°C and it cooled down to 160°C in 8 minutes. How long will it take to cool down to a temperature of 30°C with a surrounding temperature of 20°C?
First of all, for simplicity purposes, let's suppose the ambient temperature stays constant. Using Newton's law of cooling and the provided data, we arrive at the following initial-value problem

$$\frac{dT}{dt} = k(T-20), \quad T(0) = 180. \tag{2.23}$$

The differential equation 2.23 is clearly linear and separable, so we can separate the variables and get

$$\frac{dT}{T-20} = kdt.$$

Integrating both sides we obtain $\ln|T-20| = kt + c$, which implies $T = 20 + ce^{kt}$. At $t = 0$ we have that $T = 180$, so $180 = 20 + c$ gives that $c = 160$. Therefore we have $T = 20 + 160e^{kt}$. Using the fact that $T(8) = 160$ to find the value of k, $160 = 20 + 160e^{8k}$, when solving for k yields a value of $k \approx -0.016691$. Thus, the solution of 2.23 is

$$T(t) = 20 + 160e^{-0.016691t}. \tag{2.24}$$

To find how long it will take for the oil to cool down to 30°C we simply use the general solution 2.24 when $T = 30$

$$30 = 20 + 160e^{-0.016619t}.$$

And solving for t we find out that it will take approximately 167 minutes for the oil to cool down to a temperature of 30°C with an ambient temperature of 20°C.

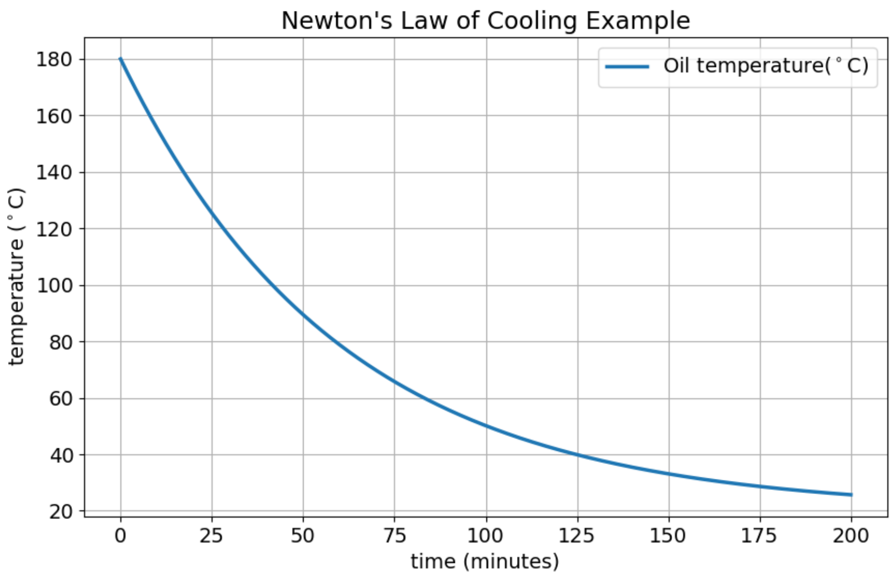


Figure 2.4: Solution of 2.23.

Chapter 3

Introduction to Neural Networks

In this chapter, we provide a brief introduction to the theory of neural networks. We start by examining the elements that make up a neural network, and after that, we explore the different types of neural networks and how they are classified based on their structure. Then, we look into the role that activation functions play in neural networks and how they are used to match complex data. Next, we go in-depth into the learning part of neural networks. We go over how these models are optimized with the use of a loss function that quantifies the performance of the model. After that, we explore the role the backpropagation algorithm has in the training of a neural network. Then, we go over the methods in which the loss function is minimized with the help of gradient descent and automatic differentiation. And finally, we examine the theory behind Physics-Informed Neural Networks, while exploring an important theorem for function approximation using neural networks.

3.1 Overview

A neural network is a model of artificial intelligence that is based on the organizational structure of the human brain. Neural networks are effective when trying to predict the outcome of an event when they have been trained with a large database of examples. These networks are adaptive, which means that they can learn from their mistakes and improve the model continuously.

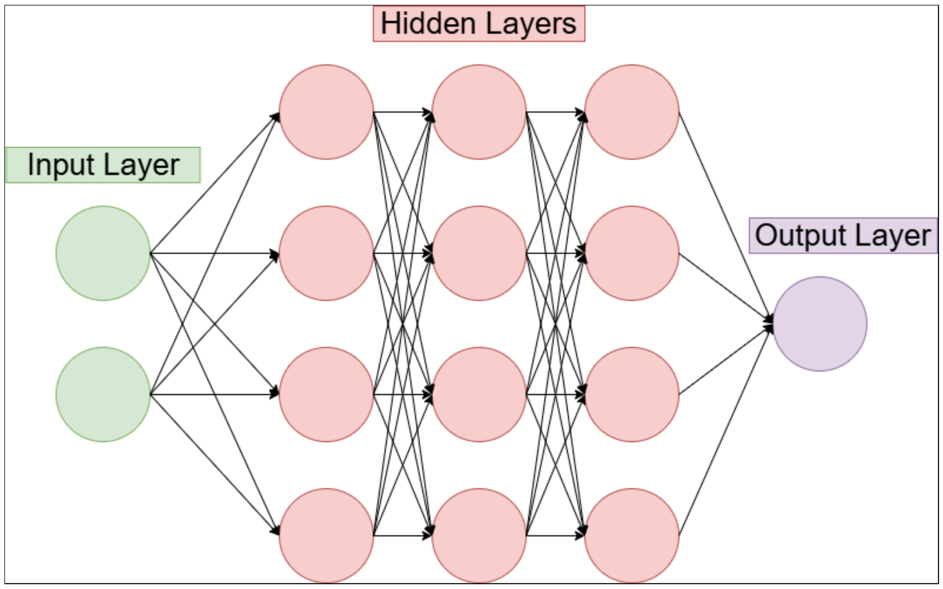


Figure 3.1: Structure of a neural network

3.2 Structure of a Neural Network

In its simplest definition, a neural network is just a directed graph that can process information, and like all graphs, a neural network is also made up of nodes and edges. The nodes of a neural network are the ones responsible for the processing of the information. These nodes are all neatly organized in what we call a layer of a neural network. We can then classify the layers of a neural network into three basic layers: the input layer, the hidden layers, and the output layer. Each of these layers has a finite number of nodes, which we will start referring to as artificial neurons.
Inside every artificial neuron of the network, we have what we call an activation function, which we will be discussing in a later section.
Every node inside each hidden layer of a neural network is connected to every node in the previous and following layer. The links between nodes are called connections or edges. These connections are unidirectional, which means that information can move only in a single direction. In Fig. 3.1 we can see the usual structure a neural network has [17, 28].

3.2.1 Neural Networks and the Brain

As mentioned previously, an artificial neural network is loosely modeled after the human brain. In this section, we seek to view the comparison between these computational models and the human brain. Firstly, without going into much detail, let's look at how a human neuron is made up, and then we discuss how the neurons in our brain communicate and work together in order for us to do anything with our body.
If we look at the structure of a single neuron of the human brain, broadly speaking, it consists of only three parts: the cell body, the axon, and the dendrites. The cell body contains the nucleus of the neuron, which is responsible for controlling the activity of the cell. The axon, which looks like a long stem, is in charge of sending messages from the cell. And lastly, the dendrites, which look like branches from a tree, are the ones responsible for receiving messages from other cells [2].
Our human brain consists of approximately 10^{11} neurons, and they communicate with each other via electrical signals [16]. Each one of our neurons can receive a vast number of signals from other neurons. All of these signals eventually reach the nucleus of the neuron, where they come together, and if the signal coming from the dendrites is strong enough, then the signal is transmitted forward to other neurons through the axon.

3.2.2 Layers of a Neural Network

As stated earlier, neural networks are formed by layers of connected nodes. Generally speaking, every one of these layers can be classified within three categories, which are the following:

- Input layer: The input layer is the one responsible for processing the real-world data we feed the neural network. Once the input layer has processed, analyzed, and categorized the data, it is passed on to the next layer. It can have one or multiple nodes, based on the specific problem we are solving. (Green in Fig. 3.1)

- Hidden layers: The hidden layers take their input from the input layer or other hidden layers. Every hidden layer takes the data from the previous layer, where it is then processed further and passed into the next layer. A neural network can have as many hidden layers as we desire(or as many as our computational power can handle). (Red in Fig. 3.1)

- Output layer: The output layer gives the final result from all the processing by the neural network. It can have single or multiple nodes, depending on the problem we are dealing with. (Purple in Fig. 3.1)

3.3 Types of Neural Networks

Artificial neural networks can be categorized into two major types: shallow neural networks and deep neural networks. The biggest difference in these two types of networks is in the number of hidden layers they have. Shallow neural networks are usually way simpler than deep neural networks, so choosing the appropriate type of network for a given task is crucial to save time and resources.

3.3.1 Shallow Neural Networks

Shallow neural networks(SNNs) are known for their fairly simple architecture. This type of network consists of the same three layers we previously discussed: an input layer, a hidden layer, and an output layer. But, for it to be categorized as a shallow neural network, it must only have one hidden layer. Due to their sheer simplicity, this type of network is known for its low complexity and limited learning capacity. Also, thanks to their basic structure, they require fewer computational resources than a deep neural network.
Even though these networks are somewhat basic, they still have their uses in some cases. For example, they might be used on problems like binary classification. We look into two examples of shallow neural networks: the single-layer perceptron and the single-layer neural network.

Single-Layer Perceptrons

A perceptron is a type of artificial neuron that takes several binary inputs $x_1, x_2, \dots, x_n$ to produce a single binary output.
The binary inputs are then multiplied by real numbers $w_1, w_2, \dots, w_n$ called weights, which are used to express the importance the respective input should have in the final output. The perceptron's output is determined by the sum

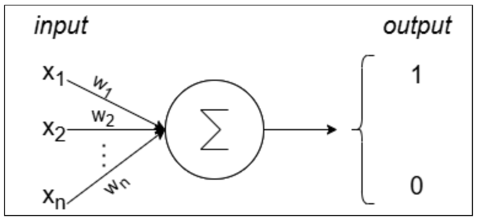


Figure 3.2: Structure of a perceptron

$\sum_{k=1}^{n} w_k x_k$ and a threshold value. If this sum is greater than the threshold value, then the perceptron's output is 1, and if the sum is less than the threshold value, then the perceptron's output is 0. In mathematical symbols, a perceptron's behavior can be written in the following way:

$$\text{output} = \begin{cases} 1 & \text{if } \sum_{k=1}^{n} w_k x_k > \text{threshold value}, \\ 0 & \text{if } \sum_{k=1}^{n} w_k x_k \leq \text{threshold value}. \end{cases}$$

A single-layer perceptron is a network consisting of a single perceptron in a single layer. Since this is one of the simplest setups you can build, it does not have many practical uses. Thanks to the binary output of the single-layer perception, they are capable of making decisions based on the values we assign as inputs and weights [22].

Single-Layer Neural Networks

Instead of being made of perceptrons, a single-layer neural network is made of artificial sigmoid neurons. Sigmoid neurons are also capable of receiving inputs $x_1, x_2, \cdots, x_n$, but unlike perceptrons, these inputs can take any numerical value between 0 and 1. Also similar to the perceptron, a sigmoid neuron has weights $w_1, w_2, \cdots, w_n$ for every input, as well as a bias, which is equivalent to the threshold value from the perceptron.
A key difference from the perceptron is that a sigmoid neuron can output any real number between 0 and 1, instead of a binary output. To get the output value of our neuron, we multiply every input and its corresponding weight and add them, then we add the bias to this sum. To obtain the output value, we use a sigmoid function, which is defined by

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Mathematically, the output of a sigmoid neuron is

$$\text{output} = \sigma\left(\sum_{k=1}^{n} w_k x_k + b\right).$$

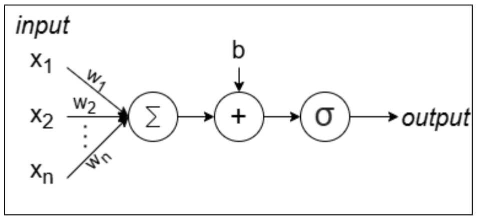


Figure 3.3: Structure of a sigmoid neuron

A single-layer neural network is the simplest artificial neural network model. It consists of sigmoid neurons organized in the same three layers: input, hidden, and output layers. Its layers can have as many sigmoid neurons as necessary, but it can only have one hidden layer.

3.3.2 Deep Neural Networks

Deep neural networks (DNNs) are another category of artificial neural networks. They get their name from the fact that these networks can be quite deep, as they have multiple hidden layers stacked one after the other between the input and output layers. A deep neural network may have thousands of layers, each with hundreds of neurons making computations. Therefore, the deeper the network, the more computational resources it needs to be implemented. Thanks to their more complex architecture, however, DNNs have a higher learning capacity than SNNs. This higher learning capacity makes them especially useful for applications like image and speech recognition, natural language processing, and computer vision [31].

Feed-Forward Neural Networks

Feed-forward neural networks are the quintessential model of deep neural networks. In this type of network, information is always fed forward, from the input layer through the hidden layers to the output layer. Each layer receives an input from the previous layer, processes it, and outputs it to

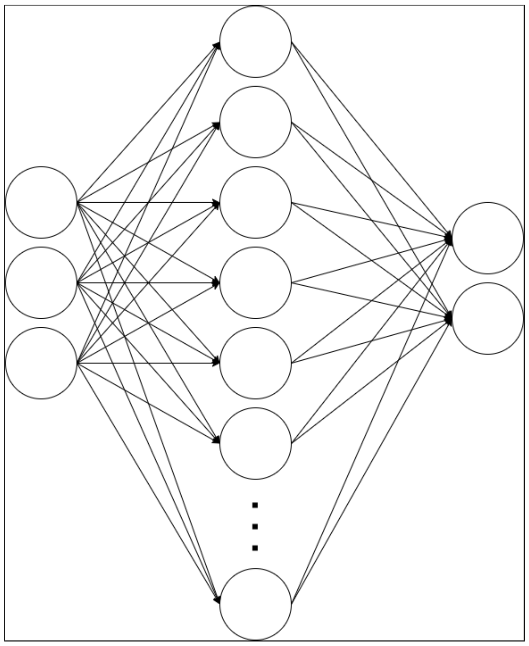


Figure 3.4: Structure of a Single Layer Neural Network

the next layer. There are no feedback connections in the network through which the model's information can be fed back into itself. So, there are no connections from a neuron to itself, nor from a neuron to a neuron in a previous layer [15].
Feed-forward neural networks are used in a variety of tasks, like pattern and image recognition. Another application of this kind of network is in the medical field, where diseases can be detected by analyzing patients' data.

3.4 Activation Functions

As we stated in the last section, to get the output of a single neuron, we have to pass the weighted sum through a function we called the sigmoid function. Now we discuss why using these kinds of functions is important when working with neural networks.
The use of nonlinear activation functions, like the logistic function or the hyperbolic tangent, allows neural networks to introduce nonlinearity to the model. If we were only to use linear activation functions, our network would,

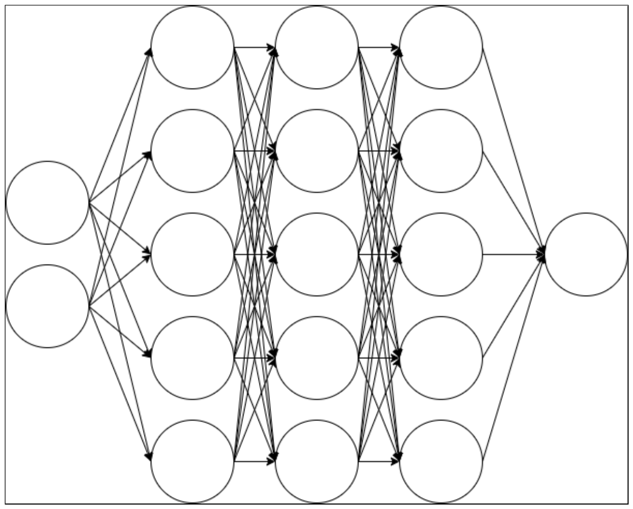
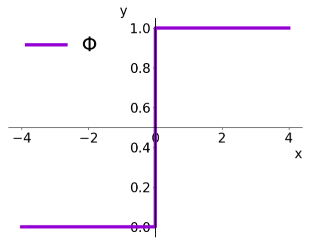
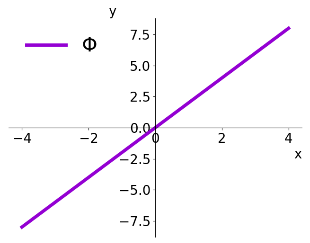
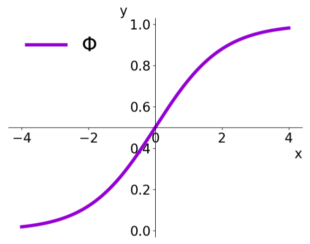
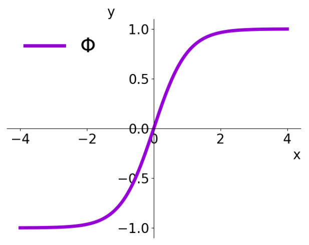
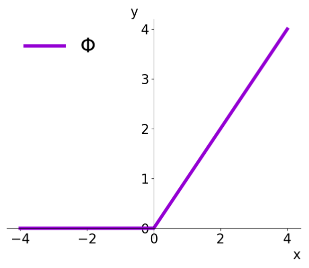
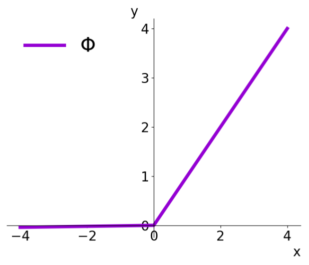


Figure 3.5: Structure of a Feed-Forward Neural Network

at most, be able to perform simple linear regression regardless of how many layers it has. Because most of the real-world data is nonlinear, it is necessary to use nonlinear activation functions to model complex patterns and relationships in the data [20].
It is important to note that the activation function doesn't necessarily need to be the same for every layer of a neural network; in a few cases, some activation functions cannot be used in the final layer.
On Table 3.4, a few examples of activation functions are shown [13, 29].

Activation Functions		
Name	Expression	Graph
Binary Step	$\Phi(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$	
Linear	$\Phi(x) = ax \quad a \in \mathbb{R}$	
Sigmoid	$\Phi(x) = \dfrac{1}{1 + e^{-x}}$	

Name	Expression	Graph
Hyperbolic Tangent	$\Phi(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$	
Rectified Linear Unit (ReLU)	$\Phi(x) = \max(0, x)$	
Leaky ReLU	$\Phi(x) = \begin{cases} x & x \geq 0 \\ 0.01x & x < 0 \end{cases}$	

3.5 Training of Neural Networks

3.5.1 Optimizing a Neural Network

Loss Function

We first look into the loss function and the importance it has when optimizing a neural network. The loss function is one of the most important aspects of

a neural network as it enables us to optimize the model and fit it to the training data.
To properly train a neural network, we first need to define a way to measure how our model is performing. To achieve this, we build a function that compares the neural network's output with the actual value it is supposed to be. The output of a neural network depends on the parameters of the network and the input data we feed it. Because we cannot change the input data, which usually comes from real-world data, the only option left to improve the model is by modifying and adjusting the parameters (weights and biases) of the network.
We define a loss function that depends on the n parameters of the neural network and yields a real number that describes how the model is behaving. We use this function to quantify the difference between the output of our model and the actual truth value. We define this loss function as follows:

$$\mathcal{L}(\omega, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2, \tag{3.1}$$

where N is the number of training inputs, y_i are the true values and $\hat{y_i}$ are the output values of the network. This type of loss function is also often called the mean squared error (MSE) loss function. There are other ways of computing this error; an example is the mean absolute error (MAE), which, instead of adding quadratic terms, adds the absolute value of the difference between the true value and the predicted value. For the remainder of this thesis, we only use the MSE formula for computing the loss function.
Clearly, $\mathcal{L} \geq 0$ for any choice of parameters the network takes. Note that $\mathcal{L}$ becomes small when a significant number of $\hat{y_i}$'s are close to their y_i counterpart, that is, the value of the loss function becomes smaller as our model becomes better at predicting the true value. On the other hand, $\mathcal{L}$ becomes large when a considerable number of $\hat{y_i}$'s are far from their y_i counterparts. Therefore, we are interested in finding a set of weights and biases that minimize the loss function to improve our model.

Backpropagation

Now we look at the backpropagation algorithm used to minimize the loss function.
In single-variable calculus, we use derivatives to find the instantaneous rate of change of a function at any point. In other words, the derivative of a

single variable function at a given point represents how much the output of the function changes in response to changes in the input. When working in multivariable calculus with functions of more than one variable, we use partial derivatives to measure how the output reacts to changes in one of the function's input variables.
When working with an n variable loss function, we need to calculate the partial derivative of the loss function with respect to every parameter to determine how each parameter changes the output of the function. To perform these calculations, we use the generalized version of the chain rule for derivatives.

Theorem 3.1 (Multivariable Chain Rule)
Let $u = f(x_1, x_2, \ldots, x_n)$ be a differentiable function of n variables, also for $i \in \{1, \ldots, n\}$ let $x_i = (t_1, t_2, \ldots, t_m)$ be a differentiable function of m variables. Then for every $j \in \{1, \ldots, m\}$

$$\frac{\partial u}{\partial t_j} = \frac{\partial u}{\partial x_1}\frac{\partial x_1}{\partial t_j} + \frac{\partial u}{\partial x_2}\frac{\partial x_2}{\partial t_j} + \cdots + \frac{\partial u}{\partial x_n}\frac{\partial x_n}{\partial t_j}.$$

Proof. Can be found in [30]. □

With the help of the chain rule, we can calculate the impact any given parameter has on the output of the network. Let's look at an example to see how it works.
Suppose we have a neural network consisting of a single input neuron, a single hidden neuron, and a single output neuron. Let x be the input of the network and y the true value. Let's denote as ω_1 the weight of the hidden layer neuron, and ω_2 the weight of the output layer neuron. Similarly, b_1 and b_2 as the biases associated with the neurons of the hidden and output layers, respectively. Also, let σ be any activation function.
First, we need to perform the forward pass, propagating the input data through the network. We must determine the weighted sum and apply the activation function at every layer.
For the hidden layer:

$$h_1 = \omega_1 \cdot x + b_1,$$

We now apply the activation function and get the output of the hidden layer

$$z_1 = \sigma(h_1).$$

For the output layer, we have

$$h_2 = \omega_2 \cdot z_1 + b_2.$$

Finally, we apply σ to get the final output of the network

$$z_2 = \sigma(h_2).$$

Explicitly, the output of the network is

$$\hat{y} = \sigma(\omega_2 \cdot z_1 + b_2) = \sigma(\omega_2 \cdot \sigma(h_1) + b_2) = \sigma(\omega_2 \cdot \sigma(\omega_1 \cdot x + b_1) + b_2).$$

We compute the loss function using the MSE formula

$$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2.$$

To get the influence of each parameter on the output, we calculate the partial derivative of the loss function with respect to each parameter using the chain rule.
For ω_1:

$$\frac{\partial \mathcal{L}}{\partial \omega_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \omega_2}.$$

Computing both partial derivatives yields

$$\frac{\partial \mathcal{L}}{\partial \omega_2} = -(y - \hat{y}) \cdot \sigma'(h_2) z_1.$$

For ω_1:

$$\frac{\partial \mathcal{L}}{\partial \omega_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial \omega_1}.$$

Computing the three derivatives yields

$$\frac{\partial \mathcal{L}}{\partial \omega_1} = -(y - \hat{y}) \cdot \sigma'(h_2)\omega_2 \cdot \sigma'(h_1)x.$$

Similarly for b_1 and b_2:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial b_2} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_2} = -(y - \hat{y}) \cdot \sigma'(h_2), \\
\frac{\partial \mathcal{L}}{\partial b_1} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial b_1} = -(y - \hat{y}) \cdot \sigma'(h_2)\omega_2 \cdot \sigma'(h_1).
\end{aligned}$$

We have determined how each parameter impacts the output of the network. Now, we need to modify these parameters to improve our model.

Gradient Descent and Optimizers

We look into the algorithm of gradient descent used to optimize neural networks. The loss function depends on the n parameters of the neural network, so our main objective is to optimize our network by finding a set of parameters that yields the smallest loss values. That is, we are interested in finding a local (or global) minimum of the loss function.
The vector containing all partial derivatives of a function is called the gradient of a function [3].

Definition 3.1 (Gradient of a function)**.** Let $f : \mathbb{R}^n \to \mathbb{R}$ differentiable at $a \in \mathbb{R}^n$. The gradient $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ of f at a is an n-dimensional vector defined as

$$\nabla f(a) = \left(\frac{\partial f}{\partial x_1}(a), \frac{\partial f}{\partial x_2}(a), \cdots, \frac{\partial f}{x_n}(a)\right).$$

The gradient has a very helpful property for optimizing the function. The gradient vector is the direction in which the function increases most rapidly; in other words, the gradient is the direction of steepest ascent of the function. But, if we take the negative of the gradient, we go in the direction in which the function decreases most rapidly; this is, $-\nabla f$ indicates the direction of steepest descent of the function [10].
Every partial derivative has already been computed thanks to the backpropagation algorithm; thus, we already have the gradient of the loss function. We need to decide how much we advance towards this direction. To get the distance we move, we define a new hyperparameter called the learning rate, $\eta > 0$. The learning rate scales the gradient, controlling how much we advance.

$$\text{step-size} = \eta \cdot \nabla \mathcal{L}.$$

As we are minimizing the function, we need to subtract the step-size from the current parameters to get the new updated parameters of the network

$$\omega_{t+1} = \omega_t - \eta \nabla \mathcal{L},$$

where ω_t represents the weights at the current time. The choice of an appropriate value of η is crucial for efficient training of the network, as choosing a high learning rate might take bigger steps and overshoot the minimum (exploding gradients). Choosing a small learning rate can make the model take tiny steps close to 0 and take a lot of time to converge to a minimum (vanishing gradients).

We iteratively run the gradient descent until we reach the limit of iterations we desired, or until the step-size is smaller than an $\epsilon > 0$.

Algorithm 1 Gradient Descent

```
Initialize $\omega_0$ arbitrarily
Choose $\eta > 0$
Choose $N > 0$
for $k = 0, 1, 2, \ldots, N$ do
    $g_k \leftarrow \nabla\mathcal{L}(\omega_k)$
    $\omega_{k+1} \leftarrow \omega_k - \eta g_k$
end for
```

Algorithm 1 shows the pseudocode for the gradient descent algorithm, the foundational algorithm of neural network optimization. It is simple to understand and implement, but it can be computationally expensive to use when working with large datasets, as the gradient is calculated using the entire dataset. We look into a few optimization algorithms based on gradient descent. These new algorithms are variants of vanilla gradient descent, making it faster and more efficient. We go over the development of optimizers through the years, culminating in the optimization algorithm we used for this thesis, the Adam optimizer.

Stochastic Gradient Descent The algorithm of stochastic gradient descent (SGD) is a simplification of the gradient descent algorithm. In this approach, the gradient is approximated by a subset or mini-batch of the data at every iteration, making it computationally cheaper and faster to run than vanilla gradient descent. At every epoch, the mini-batch of data is chosen randomly (the reason for calling it stochastic). The main advantage of SGD is that it is effective to use when working with large datasets. The noise produced by the stochastic nature of SGD can be beneficial or detrimental to the convergence to a minimum. This noise can help the algorithm escape local minima of the loss function; thus, it is possible to find an even better minimum that optimizes the model even further. But the noise can also make the algorithm oscillate around the minimum instead of directly converging to it [24].
The pseudocode for stochastic gradient descent can be seen in Algorithm 2. As the gradient is computed at a subset of the entire dataset, it allows the

Algorithm 2 Stochastic Gradient Descent (SGD) [9]

```
Initialize $\omega_0$ arbitrarily
Choose $\eta > 0$
Choose $N > 0$
for $k = 0, 1, 2, \ldots, N$ do
    Draw $\xi_k \subset D$
    $g_k \leftarrow \nabla\mathcal{L}(\xi_k)$
    $\omega_{k+1} \leftarrow \omega_k - \eta g_k$
end for
```

algorithm to run more iterations within a fixed time frame, as compared with standard gradient descent.

SGD with Momentum Optimization algorithms with momentum recognize that consistent movement in a particular direction over a medium to long term is beneficial for convergence, and this is done by minimizing the impact of local distortions in the loss function. We utilize momentum by setting a new hyperparameter $\beta \in (0, 1)$ (sometimes called friction parameter) which represents a fraction of the parameters in the previous step; hence, β represents the influence of past gradients in the current parameter update. If we set $\beta = 0$, we have the standard SGD algorithm. We call it the friction parameter because small β's act as a brake for the optimization process. Adding momentum to SGD helps the algorithm with the noise, smoothing out oscillations caused by the stochastic part of SGD. Also, with momentum, the algorithm can skip over local minima or navigate through flat regions, which helps improve the model.
Pseudocode for SGD with Momentum is presented in Algorithm 3. Adding a momentum term to standard SGD enhances the algorithm, as this term can help accelerate the learning process by taking into consideration consistent gradient directions.

AdaGrad The AdaGrad algorithm, short for adaptive gradient, is an optimization algorithm that adjusts the learning rate for each parameter rather than having a fixed learning rate. For each parameter, AdaGrad keeps track of the gradient value at every step, using this to scale the learning rate on a per-parameter basis. Thanks to storing the squared gradients over time, the AdaGrad algorithm updates infrequent parameters with more updates than

Algorithm 3 SGD with Momentum [9]

```
Initialize $\omega_0$ arbitrarily
Choose $\eta > 0$,$\beta > 0$
Choose $N > 0$
$v_0 \leftarrow 0$
for $k = 0, 1, 2, \ldots, N$ do
    Draw $\xi_k \subset \omega_k$
    $g_k \leftarrow \nabla\mathcal{L}(\xi_k)$
    $v_{k+1} \leftarrow \beta v_k + \eta g_k$
    $\omega_{k+1} \leftarrow \omega_k - v_{k+1}$
end for
```

frequent parameters, which makes it especially useful when working with sparse data. The way AdaGrad computes the learning rate and updates each parameter is shown next. Let $S_{i,k}$ be the accumulated squared gradient of the i-th parameter at iteration k. For every step, each S_i is updated as follows

$$S_{i,k} = S_{i,k} + (g_{i,k})^2,$$

where $g_{i,k}$ is the value of the gradient of the i-th parameter at step k. Then each parameter is updated

$$\omega_{i,k+1} = \omega_{i,k} - \frac{\eta}{\sqrt{S_{i,k}} + \epsilon} g_{i,k}.$$

$\omega_{i,t}$ indicates the i-th weight at time t, and ϵ is a small value, such as 10^{-8}, to avoid division by zero. From now on, we will use vector notation for simplicity, where all the operations are performed on an element-wise basis. Thus, the above equations will now be written as:

$$\boldsymbol{S_k} = \boldsymbol{S_k} + \boldsymbol{g_k^2},$$
$$\boldsymbol{\omega_{k+1}} = \boldsymbol{\omega_k} - \frac{\eta}{\sqrt{\boldsymbol{S_k}} + \epsilon} \boldsymbol{g_k}.$$

The pseudocode for the AdaGrad optimizer is found in Algorithm 4. Thanks to the per-parameter basis the learning rate takes, the optimizer is capable of fine-tuning each parameter based on its historical gradients. A major disadvantage of AdaGrad is the accumulation of square gradients in the denominator. This causes the learning rate to become smaller to the point

Algorithm 4 AdaGrad [1]

```
Initialize $\omega_0$ arbitrarily
Choose $\eta > 0$,$\beta > 0$
Choose $N > 0$
Set $\epsilon = 10^{-8}$
$\boldsymbol{S_0} \leftarrow 0$
for $k = 0, 1, 2, \ldots, N$ do
    Draw $\xi_k \subset \omega_k$
    $\boldsymbol{g_k} \leftarrow \nabla\mathcal{L}(\xi_k)$
    $\boldsymbol{S_k} \leftarrow \boldsymbol{S_k} + \boldsymbol{g_k}^2$
    $\boldsymbol{\omega_{k+1}} \leftarrow \boldsymbol{\omega_k} - \frac{\eta}{\sqrt{\boldsymbol{S_k}} + \epsilon}\boldsymbol{g_k}$
end for
```

that it becomes infinitesimally small, preventing the algorithm from further optimization.

RMSProp Now we look at the Root Mean Square Propagation optimization algorithm, or RMSProp for short. Like AdaGrad, it computes an adaptive learning rate for each parameter; however, the calculation is done differently. RMSProp addresses one of AdaGrad's main problems, which is the diminishing values for the learning rate caused by the addition of the squared gradients throughout the run time of the algorithm. Instead, the RMSProp algorithm uses an exponentially decaying average of squared gradients. Doing this prevents the learning rate from shrinking excessively by reducing the impact of past gradients. We introduce a decay factor $\rho \in (0, 1)$ that weighs the past gradients to weaken their impact. We multiply ρ by the aggregate of the squared gradients, and then we add $(1 - \rho)$ times the squared gradient at the current epoch. For each parameter i, we have

$$\boldsymbol{S_k} = \rho\boldsymbol{S_k} + (1 - \rho)\boldsymbol{g_k}^2.$$

Then all the parameters are updated

$$\boldsymbol{\omega_{k+1}} = \boldsymbol{\omega_k} - \frac{\eta}{\sqrt{\boldsymbol{S_k}} + \epsilon}\boldsymbol{g_k}.$$

RMSProp's pseudocode is shown in Algorithm 5. Similar to AdaGrad, RMSProp features a dynamic learning rate that varies for each parameter. On

Algorithm 5 RMSProp [1]

```
Initialize $\omega_0$ arbitrarily
Choose $\eta > 0$
Choose $\rho \in (0, 1)$
Choose $N > 0$
Set $\epsilon = 10^{-8}$
$\boldsymbol{S_0} \leftarrow 0$
for $k = 0, 1, 2, \ldots, N$ do
    Draw $\xi_k \subset \omega_k$
    $\boldsymbol{g_k} \leftarrow \nabla\mathcal{L}(\xi_k)$
    $\boldsymbol{S_k} \leftarrow \rho\boldsymbol{S_k} + (1-\rho)\boldsymbol{g_k^2}$
    $\boldsymbol{\omega_{k+1}} \leftarrow \boldsymbol{\omega_k} - \frac{\eta}{\sqrt{\boldsymbol{S_k}} + \epsilon}\boldsymbol{g_k}$
end for
```

the other hand, RMSProp addresses AdaGrad's main disadvantage by updating the parameters based on the exponentially decaying average of the squared gradients, instead of simply accumulating them.

Adam The Adaptive Moment Estimation optimizer (Adam for short) is one of the most commonly used optimizers, combining the foundation of RMSProp and momentum-based optimization. Instead of applying the current gradients, Adam optimizer applies momentum like in the SGD optimizer with momentum, and similar to RMSProp, it applies an adaptive learning rate to each parameter based on its past gradients. The Adam optimizer stores the exponentially decaying average of squared gradients (like RMSProp), as well as an exponentially decaying average of past gradients (similar to momentum). Two new decay rates β_1 and β_2 are introduced, then the first momentum (the mean) and the second momentum (the uncentered variance) of the gradients are estimated as follows:

$$\boldsymbol{m_k} = \beta_1\boldsymbol{m_{k-1}} + (1-\beta_1)\boldsymbol{g_k},$$
$$\boldsymbol{v_k} = \beta_2\boldsymbol{v_{k-1}} + (1-\beta_2)\boldsymbol{g_k^2}.$$

As both m_k and v_k are initialized at zero, during the early steps, they are biased towards zero. To counteract this, a bias correction is performed on the first and second moments:

$$\hat{\boldsymbol{m}}_{\boldsymbol{k}} = \frac{\boldsymbol{m}_{\boldsymbol{k}}}{(1-\beta_1^k)},$$
$$\hat{\boldsymbol{v}}_{\boldsymbol{k}} = \frac{\boldsymbol{v}_{\boldsymbol{k}}}{(1-\beta_2^k)},$$

with β_1^t and β_2^t being the decay rates β_1 and β_2 to the power k. Then the parameters are updated

$$\boldsymbol{\omega}_{\boldsymbol{k}} = \boldsymbol{\omega}_{\boldsymbol{k-1}} - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_{\boldsymbol{k}}}+\epsilon}\hat{\boldsymbol{m}}_{\boldsymbol{k}}.$$

Algorithm 6 Adam [19]

```
Initialize $\omega$ arbitrarily
Choose $\eta > 0$
Choose $\beta_1, \beta_2 \in [0, 1)$
Choose $N > 0$
Set $\epsilon = 10^{-8}$
$\boldsymbol{m}_{\boldsymbol{0}} \leftarrow 0$
$\boldsymbol{v}_{\boldsymbol{0}} \leftarrow 0$
for $k = 0, 1, 2, \ldots, N$ do
    Draw $\xi_k \subset \omega_k$
    $\boldsymbol{g}_{\boldsymbol{k}} \leftarrow \nabla\mathcal{L}(\xi_k)$
    $\boldsymbol{m}_{\boldsymbol{k}} \leftarrow \beta_1\boldsymbol{m}_{\boldsymbol{k-1}} + (1-\beta_1)\boldsymbol{g}_{\boldsymbol{k}}$
    $\boldsymbol{v}_{\boldsymbol{k}} \leftarrow \beta_2\boldsymbol{v}_{\boldsymbol{k-1}} + (1-\beta_2)\boldsymbol{g}_{\boldsymbol{k}}^2$
    $\hat{\boldsymbol{m}}_{\boldsymbol{k}} \leftarrow \frac{\boldsymbol{m}_{\boldsymbol{k}}}{(1-\beta_1^k)}$
    $\hat{\boldsymbol{v}}_{\boldsymbol{k}} \leftarrow \frac{\boldsymbol{v}_{\boldsymbol{k}}}{(1-\beta_2^k)}$
    $\boldsymbol{\omega}_{\boldsymbol{k}} \leftarrow \boldsymbol{\omega}_{\boldsymbol{k-1}} - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_{\boldsymbol{k}}}+\epsilon}\hat{\boldsymbol{m}}_{\boldsymbol{k}}$
end for
```

The pseudocode for the Adam optimizer is presented in Algorithm 6. The author's default settings are $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$. The Adam optimizer combines the advantages of both AdaGrad and RMSProp algorithms, while also requiring little memory and being straightforward to implement. This makes Adam a robust choice for a wide range of optimization problems in machine learning.

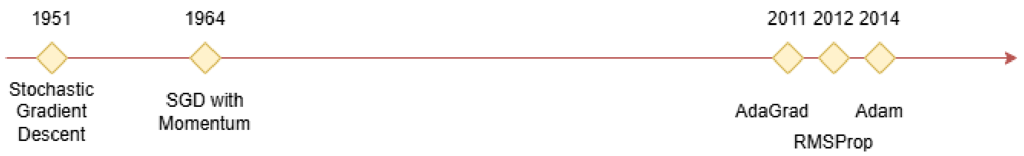


Figure 3.6: Optimizer development timeline [33]

The development of optimization theory is shown in Fig. 3.6. It starts from simpler algorithms like SGD and SGD with Momentum algorithms, and ends with more recent and more complex algorithms like the Adam optimizer.

3.5.2 Automatic Differentiation

We look into automatic differentiation (sometimes called autodiff), which is an algorithm used to compute the partial derivatives of a function. Computation of derivatives in computers can be divided into four categories: a) manually calculating the derivatives and coding them directly, b) numerical differentiation, c) symbolic differentiation, and d) automatic differentiation. Manually computing the derivatives and coding them is time-consuming and inefficient, while using numerical differentiation can be inaccurate due to rounding and truncation errors. Using symbolic differentiation can lead to complex expressions and inefficient code, which becomes the ”expression swell” problem [4]. Thus, the autodiff algorithm was developed to numerically evaluate derivatives of functions. The main idea behind autodiff is to break down the function into a sequence of arithmetic operations and elementary functions, then apply basic derivative rules (like the power rule, product rule, etc) to compute the full derivative of the function with the help of the chain rule.
We break down the evaluation of a function with the help of computational graphs. A computational graph is a directed graph that represents the process by which a mathematical expression is computed. The nodes of the graph represent the operations (addition, multiplication, etc) or elementary function evaluation (cosine, logarithm, etc), and the edges of the graph represent the flow of the data between the operations. The evaluation of any function is the composition of a finite number of elementary operations with known derivatives. Suppose we have a function $f : \mathbb{R}^n \to \mathbb{R}$, to construct the computational graph of f we use intermediate variables v_i such that:

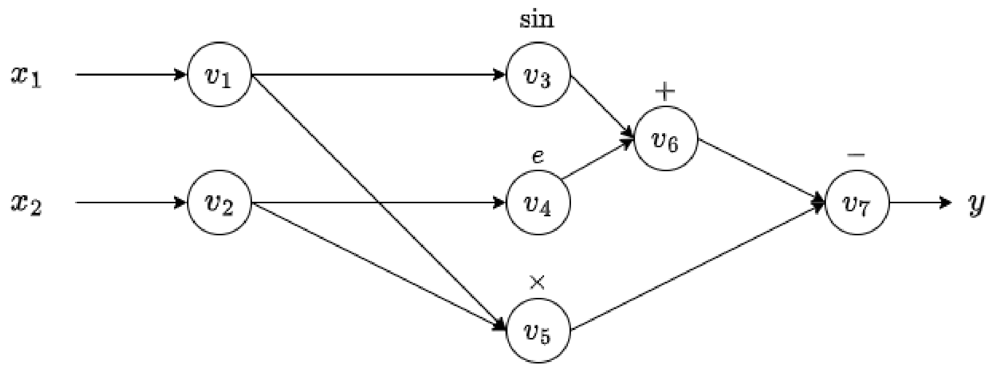


Figure 3.7: Computational graph of the function 3.2.

- $v_i = x_i, i = 1, \ldots, n$ are the input values;
- $v_i, i = n, \ldots, k$ are the intermediate variables;
- $y_{=}v_{k+1}$ is the output value.

Figure 3.7 shows the computational graph of the two-variable function:

$$f(x_1, x_2) = \sin x_1 + e^{x_2} - x_1 x_2. \tag{3.2}$$

Table 3.2 shows the evaluation trace of computing f at $\boldsymbol{x} = (-3, 5)$.

Forward Mode We now look into the autodiff method of computing partial derivatives in forward mode. To compute the derivative of function 3.2 with respect to x_1 we define a partial derivative for each v_i

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1} \quad i = 1, \ldots, 7.$$

By applying the chain rule to each v_i, we generate the derivative forward trace of f with respect to x_1 at $\boldsymbol{x}$. Notice that by calculating $\dot{v}_7$, by definition of y we also calculate the partial derivative of $y = f(x_1, x_2)$ with respect to x_1 (Table 3.3 shows these computations). To calculate the derivative of f with respect to x_2 we would have to do another forward pass. Thus, if we had a function $g : \mathbb{R}^n \to \mathbb{R}$ and we wanted to calculate the gradient of g at a given $x \in \mathbb{R}^n$, then we would have to perform n forward passes of autodiff to fully compute the gradient. This is highly inefficient, as for more complex functions in large n means more time consumed and more computational cost to calculate the gradient.

Table 3.2: Evaluation trace of 3.2 at $\boldsymbol{x} = (-3, 5)$.

Evaluation trace	
$v_1 = x_1$	$= -3$
$v_2 = x_2$	$= 5$
$v_3 = \sin v_1$	$= \sin(-3) = -0.141$
$v_4 = e^{v_2}$	$= e^5 = 148.4131$
$v_5 = v_1 \times v_2$	$= (-3)(5) = -15$
$v_6 = v_3 + v_4$	$= -0.1411 + 148.4131 = 148.272$
$v_7 = v_6 - v_5$	$= 148.272 - (-15) = 163.272$
$y = v_7$	$= 163.272$

Table 3.3: Forward autodiff trace of 3.2 at $\boldsymbol{x} = (-3, 5)$.

Evaluation trace		**Forward AD trace**	
$v_1 = x_1$	$= -3$	$\dot{v}_1$	$= 1$
$v_2 = x_2$	$= 5$	$\dot{v}_2$	$= 0$
$v_3 = \sin v_1$	$= -0.141$	$\dot{v}_3 = \cos(v_1)\dot{v}_1$	$= -0.9899$
$v_4 = e^{v_2}$	$= 148.4131$	$\dot{v}_4 = e^{v_2}\dot{v}_2$	$= 0$
$v_5 = v_1 \times v_2$	$= -15$	$\dot{v}_5 = \dot{v}_1 v_2 + \dot{v}_2 v_1$	$= 5$
$v_6 = v_3 + v_4$	$= 148.272$	$\dot{v}_6 = \dot{v}_3 + \dot{v}_4$	$= -0.9899$
$v_7 = v_6 - v_5$	$= 163.272$	$\dot{v}_7 = \dot{v}_6 - \dot{v}_5$	$= -5.9899$
$y = v_7$	$= 163.272$	$\frac{\partial y}{\partial x_1} = \dot{v}_7$	$= -5.9899$

Reverse AD trace	
$\bar{v}_7 = \frac{\partial y}{\partial v_7} = \frac{\partial}{\partial v_7} v_7$	$= 1$
$\bar{v}_6 = \frac{\partial y}{\partial v_6} = \frac{\partial y}{\partial v_7}\frac{\partial v_7}{\partial v_6} = \bar{v}_7 \times 1$	$= 1$
$\bar{v}_5 = \frac{\partial y}{\partial v_5} = \frac{\partial y}{\partial v_7}\frac{\partial v_7}{\partial v_5} = \bar{v}_7 \times (-1)$	$= -1$
$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_6}\frac{\partial v_6}{\partial v_4} = \bar{v}_6 \times 1$	$= 1$
$\bar{v}_3 = \frac{\partial y}{\partial v_3} = \frac{\partial y}{\partial v_6}\frac{\partial v_6}{\partial v_3} = \bar{v}_6 \times 1$	$= 1$
$\bar{v}_2 = \frac{\partial y}{\partial v_2} = \frac{\partial y}{\partial v_4}\frac{\partial v_4}{\partial v_2} + \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_2} = \bar{v}_4 e^{v_2} + \bar{v}_5 v_1$	$= 151.4131$
$\bar{v}_1 = \frac{\partial y}{\partial v_1} = \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_1} + \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_1} = \bar{v}_3 \cos v_1 + \bar{v}_5 v_2$	$= -5.9899$

Table 3.4: Reverse autodiff trace of 3.2 at $\boldsymbol{x} = (-3, 5)$

Reverse Mode Now we look into the reverse mode of calculating derivatives. In this mode, the derivatives are computed backwards starting from the output of the function. We define adjoint variables for each auxiliary v_i as follows

$$\bar{v}_i = \frac{\partial y}{\partial v_i}.$$

If the i-th node has j successors, then the chain rule gives us

$$\bar{v}_i = \sum_{j \in \{\text{next } i\}} \bar{v}_j \frac{\partial v_j}{\partial v_i}.$$

We calculate each $\bar{v}_i$ propagating iteratively in reverse, beginning from the function's output. Returning to our example, Table 3.4 shows the computation of the autodiff algorithm in reverse mode.

Notice the rows highlighted in blue in Table 3.4, by definition $\bar{v}_2$ and $\bar{v}_1$ are the partial derivatives of f with respect to x_2 and x_1, respectively. Thus, we have computed the full gradient of f with a single pass of reverse mode autodiff. The disadvantage of reverse mode is its storage requirement, as the amount of storage needed grows proportionally to the number of operations in the function (in the worst case scenario) [4].

3.6 Physics-Informed Neural Networks

Physics-Informed Neural Networks (PINNs) are a type of neural network that is governed by physical laws described by differential equations. Unlike traditional neural networks that require input data to learn, PINNs incorporate the knowledge from the physical system into the learning process. This is done by directly incorporating the equations governing the system into the loss function of the network.
In the following notation, t is the input of our network; therefore, the whole network is a function of t. Also, N_n^j denotes the j-th neuron in the n-th layer of the neural network.

$$\begin{aligned}
&N_0^1(t) = t,\\
&N_m^j(t) = \sigma\left(\sum_{k=1}^{h_{m-1}} \omega_m^{(j,k)} N_{m-1}^{(k)}(t) + b_m^{(j)}\right),\\
&N_{F+1}(t) = \sum_{k=1}^{h_F} \omega_{F+1}^{(j,k)} N_F^{(k)}(t) + b_{F+1}^{(j)},\\
&\text{Output:}\left\{N_{F+1}^{(j)}(t)\right\}_{j=1}^{h_{F+1}}.
\end{aligned}$$

Where:

- F is the number of hidden layers in the network,
- L_m is the m-th hidden layer,
- h_m is the number of neurons in L_m,
- $N_0^1(t)$ is the neuron in the input layer of the network,
- $N_m^j(t)$ denotes the j-th neuron in L_m,

- $\omega_m^{(j,k)}$ is the weight of the link connecting the k-th neuron in L_{m-1} to the j-th neuron in L_m,
- b_m^j is the bias associated to $N_m^j(t)$,
- σ is any activation function.

Note that the output is a collection of n nodes in the output layer. Also, observe how when we go from the last of the hidden layers we have in our network (hidden layer number F) to the layer $F+1$ (the output layer), we do not apply the activation function σ and we only take the weighted sum of that neuron.
The Universal Approximation Theorem is a fundamental result in the field of neural networks. In broad terms, it states that any continuous function on a compact subset can be approximated by a feedforward neural network containing a single hidden layer with a finite number of neurons using a sigmoidal activation function (sigmoid, hyperbolic tangent, etc.).

Theorem 3.2
Let σ be any continuous sigmoidal function. Then, finite sums of the form

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j)$$

are dense in $C(I_n)$(n-dimensional unit cube). In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $G(x)$ of the above form, for which

$$|G(x) - f(x)| < \epsilon \qquad \forall x \in I_n.$$

Proof. The proof for this theorem can be found in Cybenko [12]. □

In the theorem above, N is the number of neurons in the hidden layer of the neural network, α_j are the output weights for the j-th neuron, y_j^T are the input weights for the j-th neuron, x is the input of the network and θ_j are the biases of the j-th neuron.
This means neural networks are not restricted to linear relationships; they are capable of approximating non-linear complex relationships between inputs and outputs. Another consequence of this theorem is that a single-layer neural network is enough to approximate any continuous function. This means that given any continuous function, there exists a finite number of

neurons that can approximate the function. Although Theorem 3.2 shows the existence of a neural network that can approximate any continuous function, it does not provide an algorithm to find the weights and biases that approximate the function. This theorem demonstrates that a neural network can be used as a universal function approximator.

Chapter 4

Solving ODEs and DDEs Using Neural Networks

In this chapter, we go over the inner mechanisms of the PINN method for solving ODEs and DDEs. We start by constructing the loss function used to approximate ODEs. Next, we construct the loss function for DDEs, along with its small differences compared to the base ODE method. To conclude this chapter, we will look at different examples of ODEs and DDEs, and explain the different network configurations used to achieve our results.

4.1 How does it work

In the Physical-Informed Neural Network method of solving differential equations, the neural network is used to approximate a function. The idea is that the function we approximate is the solution of the differential equation, given an equation and a set of initial conditions. The input of the network is the independent variable of the equation, such as time t, while its output u is the approximation of the solution.
Suppose that we are interested in approximating the ordinary differential equation:

$$\begin{aligned} F(x, y, y', y'', \dots, y^{(n)}) &= 0 \\ y(x_0) = y_0, y'(x_0) &= y_1, \dots, y^{(n-1)}(x_0) = y_{n-1}. \end{aligned} \tag{4.1}$$

The main idea behind PINN is to incorporate the physics that governs the system into the training of the neural network. To use the PINN method, we need a differential equation and a set of appropriate initial conditions. For the training of the network, we build a loss function consisting of two parts: the data loss and the physics loss.

For the physics loss part of the total loss function, we require that our approximation satisfies the differential equation for all t in a given interval. Because it is impossible to work with a continuous interval, we sample a finite number of points N_f within the desired interval; this set of points is called collocation points. These points can be chosen at random through a uniform distribution, or the points can be evenly distributed across the interval. The latter has been chosen for this thesis. Selecting an appropriate number of points N_f is important to achieve a good result. Choosing too few points may lead to a "crooked" approximation, while choosing too many points might increase the program's execution time or memory usage.

Suppose that the network's input is a set of N_f points in the domain; we denote this input by t, while the output of the network is given by u. Then, the output of the network is a function of the parameters θ and input values t; this is $u(\theta, t)$. Using the autograd algorithm, we can compute the partial derivatives of $u(\theta, t)$ with respect to t. If $u(\theta, t)$ is a solution of the differential equation 4.1, then it must follow that

$$F\left(t, u(\theta,t), \frac{\partial u(\theta,t)}{\partial t}, \frac{\partial^2 u(\theta,t)}{\partial t^2}, \ldots, \frac{\partial^n u(\theta,t)}{\partial t^n}\right) = 0.$$

Thus, we need to push $F\left(t, u(\theta,t), \frac{\partial u(\theta,t)}{\partial t}, \frac{\partial^2 u(\theta,t)}{\partial t^2}, \ldots, \frac{\partial^n u(\theta,t)}{\partial t^n}\right)$ toward zero. We do this by measuring the distance between the true value, in this case 0, and the predicted value. This is done by using the Mean Squared Error between both quantities, specifically

$$\mathcal{L}_{ODE} = \frac{1}{N_f}\sum_{i=1}^{N_f}\left(F\left(t_i, u(\theta,t_i), \frac{\partial u(\theta,t_i)}{\partial t}, \frac{\partial^2 u(\theta,t_i)}{\partial t^2}, \ldots, \frac{\partial^n u(\theta,t_i)}{\partial t^n}\right) - 0\right)^2,$$

or simply

$$\mathcal{L}_{ODE} = \frac{1}{N_f}\sum_{i=1}^{N_f}\left(F\left(t_i, u(\theta,t_i), \frac{\partial u(\theta,t_i)}{\partial t}, \frac{\partial^2 u(\theta,t_i)}{\partial t^2}, \ldots, \frac{\partial^n u(\theta,t_i)}{\partial t^n}\right)\right)^2.$$

In the data loss part, we measure the discrepancy between the network's output and the true values at specific points, such as the initial conditions.

This term guarantees that the network will fit the initial conditions of the differential equation. Let N_{IC} be the number of initial conditions of the differential equation; then the data loss term is written as

$$\mathcal{L}_{IC} = \frac{1}{N_{IC}} \sum_{i=1}^{N_{IC}} (u_{true}(t_i) - u_{approx}(t_i))^2,$$

or simply the Mean Squared Error between the predicted value given by the network and the true value, given by the initial conditions of the differential equation.
The total loss function of the neural network consists of a weighted sum of these two terms

$$\mathcal{L}_{total} = \omega_{IC}\mathcal{L}_{IC} + \omega_{ODE}\mathcal{L}_{ODE},$$

where ω_{IC} and ω_{ODE} are weighting terms. Therefore, by minimizing $\mathcal{L}_{total}$ with a gradient descent-based optimizing algorithm, we are, in fact, solving the differential equation 4.1.

4.2 Implementation

We discuss the implementation of the PINN method in Python using the Pytorch library. As the implementation of ODEs and DDEs is slightly different, we look at them separately.

4.2.1 Ordinary Differential Equations

When implementing for ordinary differential equations, the construction of the loss function previously described is followed directly. The details of the network employed vary from example to example, so they will be presented thoroughly for each of our examples.

4.2.2 Delay Differential Equations

The implementation of delay differential equations differs from the implementation for ODEs, so we discuss it in detail.

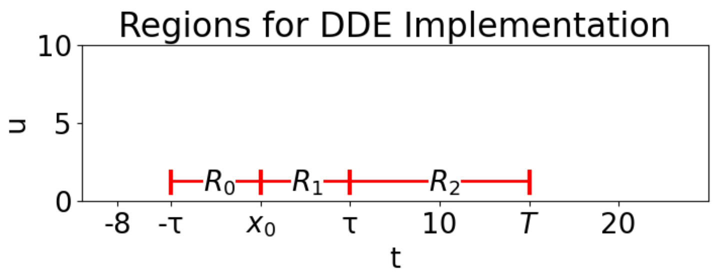


Figure 4.1: The three regions we divide the $[-\tau, T]$ domain.

Suppose we are interested in solving the delay differential equation with a single constant delay

$$\begin{aligned} &y'(t) = f(t, y(t), y(t-\tau)). \\ &y(t) = \phi(t), \ t \in [-\tau, x_0]. \end{aligned} \tag{4.2}$$

on the interval $[x_0, T]$, with $\tau > 0$ and $T > \tau$.
We split the $[-\tau, T]$ input interval into three subintervals: $[-\tau, x_0]$, $[x_0, \tau]$ and $[\tau, T]$. Furthermore, we denote these regions as R_0, R_1 and R_2, respectively. This division is shown in Figure 4.1. The input for this neural network is a collection of points contained within the interval of the region we are working with. We sample a number of S_0, S_1 and S_2 collocation points on the R_0, R_1 and R_2 regions, respectively. The selection of the number of points S_0, S_1 and S_2 is done in a way such that the density of points is similar among the three regions. The output of the network consists of three parts, each of these parts corresponds to one of the three regions we input into the network. We denote these outputs as: u_{R_0}, u_{R_1} and u_{R_2}.
The loss function employed for DDEs consists of three terms, each one of them corresponding to one of the subintervals of the domain. We go over the construction of this loss function.
For the R_0 region we have the interval $[-\tau, x_0]$, this is $t \leq x_0$ for all $t \in R_0$. Thus, for this region, we want the output of the neural network to match the initial condition function, $\phi(t)$, given by the DDE. Thus, the first part of the loss function is simply given by

$$\mathcal{L}_{R_0} = \frac{1}{S_0} \sum_{i=1}^{S_0} \left(\phi(t_i) - u_{R_0}(\theta, t_i)\right)^2 .$$

For the R_1 region, we are situated on the interval $[x_0, \tau]$, where we work with the differential equation $f(t, y(t), y(t-\tau))$. Notice the $y(t-\tau)$ term in the differential equation, when we subtract τ to every $t \in [x_0, \tau]$, this becomes $[-\tau, x_0]$. As a result, in this case, the $y(t-\tau)$ values will essentially be the y values in $[-\tau, x_0]$. But the y values in $[-\tau, x_0]$ are the values given by the initial condition function $\phi(t)$. Thus, for this region, the differential equation will be $f(t, y(t), \phi(t-\tau))$. The input for this region are S_1 points in the $[x_0, \tau]$ interval. We want the output for this region, u_{R_1}, to be a solution to the DDE in R_1. This means that the derivative of our approximation with respect to the input of this region should match the right-hand side of the differential equation, with the approximation substituted in. Hence, the loss function for R_1 is

$$\mathcal{L}_{R_1} = \frac{1}{S_1} \sum_{i=1}^{S_1} \left(\dot{u}_{R_1}(\theta, t_i) - f(t_i, u_{R_1}(\theta, t_i), \phi(t_i - \tau)) \right)^2 .$$

For the R_2 region we work on the $[\tau, T]$ subinterval and the differential equation $f(t, y(t), y(t-\tau))$. We sample a total of S_2 points in R_2, which will become the input of our network for this region. We desire the output for this region, u_{R_2}, to be a solution to the differential equation. Analogous to the previous region, this implies that the derivative of the output has to match the right-hand side of the differential equation substituted in by the output in order for u_{R_2} to be a solution on R_2. Therefore, the loss function for this region is

$$\mathcal{L}_{R_2} = \frac{1}{S_2} \sum_{i=1}^{S_2} \left(\dot{u}_{R_2}(\theta, t_i) - f(t_i, u_{R_2}(\theta, t_i), u_{R_2}(\theta, t_i - \tau)) \right)^2 .$$

Combining these three parts of the loss function, we end up with the total loss function for delay differential equations

$$\mathcal{L}_{total} = \omega_{R_0} \mathcal{L}_{R_0} + \omega_{R_1} \mathcal{L}_{R_1} + \omega_{R_2} \mathcal{L}_{R_2},$$

where $\omega_{R_0}, \omega_{R_1}$ and ω_{R_2} are weighting terms. By minimizing this loss function with a gradient descent-based algorithm, we are essentially solving the delay differential equation 4.2.

4.3 Examples

We show a few examples where we used the PINN method to approximate differential equations. The code used for each example can be found github.com/jcalvoq/thesisPINN.

4.3.1 Ordinary Differential Equations

Example with Known Solution

We begin with an example of a differential equation that has an analytic solution. We approximate the following first-order initial value problem

$$\frac{dx}{dt} + \frac{x}{5} = e^{-\frac{t}{5}} \cos(t), \quad [21] \tag{4.3}$$

with $x(0) = 0$ and $t \in [0, 10]$. The analytic solution to this IVP is given by $x(t) = e^{-\frac{t}{5}} \sin(t)$.
In this scenario, we compare the approximation given by the neural network with the known exact solution. For this example, we give the explicit loss function implemented. Let $u^T(\theta, t)$ be a trial solution of the IVP. The physics loss part is:

$$\mathcal{L}_{ODE} = \frac{1}{N_f} \sum_{i=1}^{N_f} \left(\dot{u}^T(\theta, t_i) + \frac{1}{5} u^T(\theta, t_i) - e^{-t_i/5} \cos(t_i) \right)^2 .$$

Because we only have one initial condition $N_{IC} = 1$, the data loss part of the loss function is:

$$\mathcal{L}_{IC} = (u^T(\theta, t_0) - 0)^2.$$

For this example, $\omega_{IC} = \omega_{ODE} = 1/2$. Then, the total loss function is:

$$\mathcal{L}_{total} = \frac{1}{2}(\mathcal{L}_{ODE} + \mathcal{L}_{IC}).$$

In this case, we used a network consisting of 4 hidden layers with 40 nodes in each one, and sampled a total of 5000 uniformly spaced points in the interval. The activation function used was the hyperbolic tangent throughout the whole network, while the selected optimizer was the Adam optimizer with the default learning rate of 1×10^{-3}. Figure 4.2 shows how the approximation

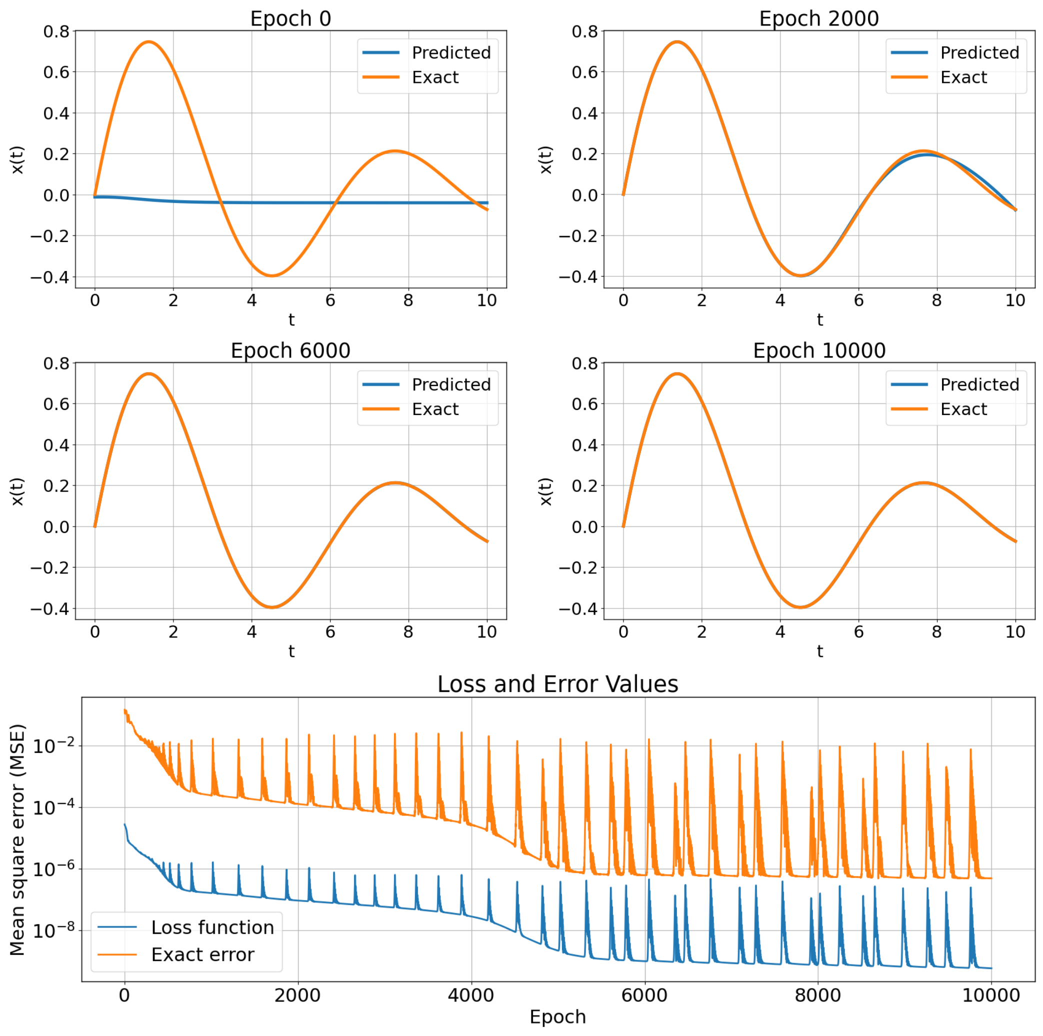


Figure 4.2: Evolution of approximations of IVP 4.3 and values of loss function and exact error.

changed over time, as well as the behavior of the loss function and the exact error in every epoch.
As it's a fairly simple differential equation, it did not take a lot of time for the model to converge to the exact solution. The final solution graph in Fig. 4.2 is the 10000th epoch we ran, and the approximation matched perfectly with the exact solution.

Logistic Equation

We now look into the logistic equation, which is used to model population growth with a carrying capacity. The population dynamic is given by the first-order differential equation

$$\frac{dP}{dt} = rP\left(1 - \frac{P}{K}\right),$$

where P represents the population size at any time t, r represents the growth rate of the population, and K represents the carrying capacity, or the maximum population the environment can sustain.
In this case, we solve the logistic differential equation with parameters $r = 1$ and $K = 2$. We are interested in solving the differential equation on the interval $[0, 15]$, with an initial condition $P(0) = 0.01$. Thus, the differential equation we look to solve is

$$\frac{dP}{dt} = P\left(1 - \frac{P}{2}\right). \tag{4.4}$$

For the network details for this example, we employed a neural network consisting of 3 hidden layers, with a total of 40 nodes in each layer. We deployed a total of 1,000 evenly spaced points on the $[0, 12]$ interval. We used the sigmoid activation function throughout the whole network. The designated optimizer this time was the Adam optimizer, with a learning rate of 1×10^{-2}. Because the logistic equation has a known solution, in this case given by $f(x) = \frac{2e^x}{e^x+199}$, we compare our approximation to the exact solution and compute the exact error. The evolution of the approximations over the 200,000 epochs we ran the program, as well as the values of the loss function and the exact error values, are shown in Figure 4.3.
As can be seen in Fig. 4.3, the approximations start taking the shape of the exact solution starting from the 10000th epoch, slowly ascending until the 180000th, where the approximation finally converges to the exact solution.

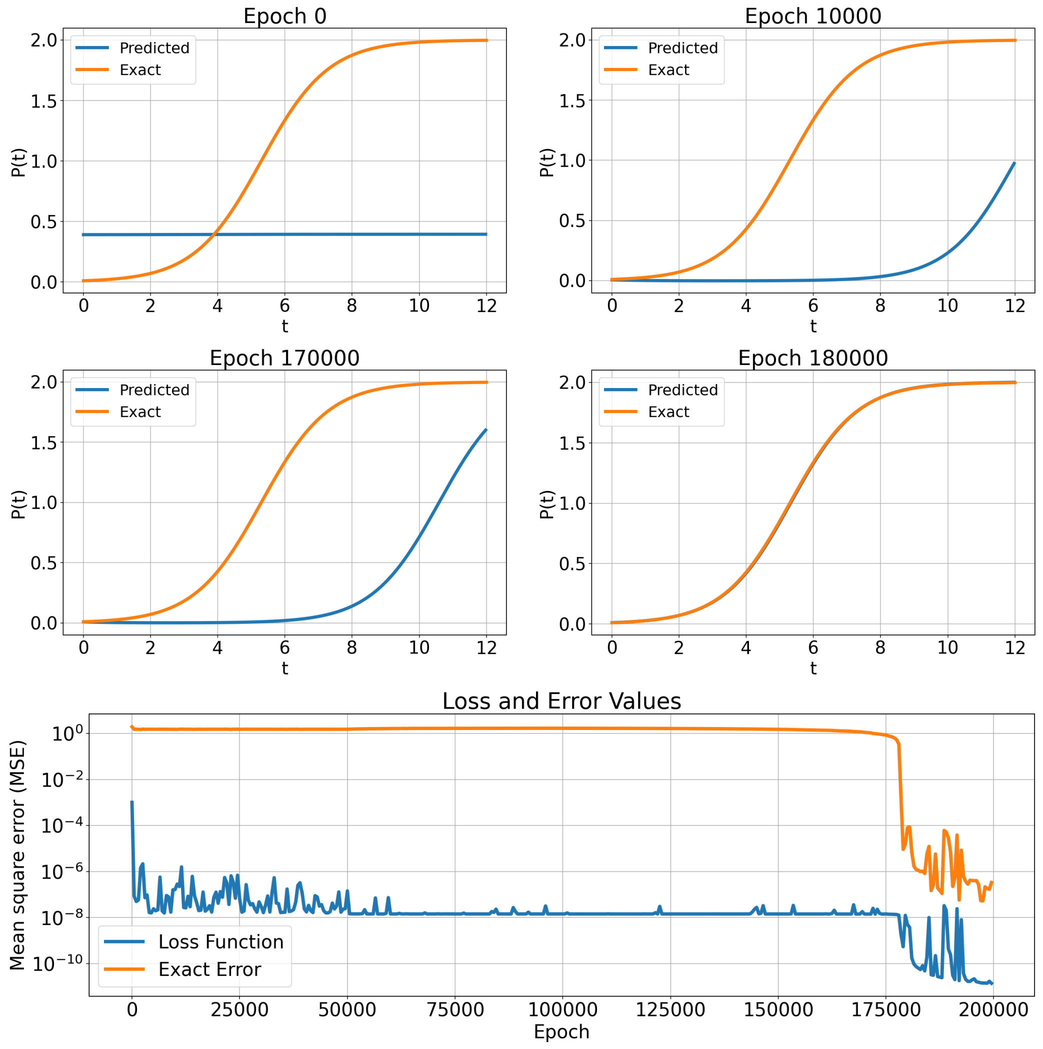


Figure 4.3: Approximation history and loss function values of logistic equation 4.4.

The values of the loss function and exact error follow the behavior presented by the approximations.
A small trick was used to make the approximation converge to the solution. For the first 50,000 epochs, we placed an additional term for the loss function to satisfy; we made it so that the last point in the collection of t values equaled 1 in our approximation. This is

$$\mathcal{L}_{ADD} = \frac{1}{N_f} \sum_{i=1}^{N_f} \left(y(t_{N_f}) - 1 \right)^2 .$$

Thus, the loss function for this example is

$$\mathcal{L}_{total} = \omega_{IC} \mathcal{L}_{IC} + \omega_{ODE} \mathcal{L}_{ODE} + \omega_{ADD} \mathcal{L}_{ADD}.$$

Note that we only compute this additional term for the first 50,000 epochs; for the remainder of the program's runtime $\mathcal{L}_{ADD}$ is equal to 0. If we were not to add this new condition, the approximation stays in the unstable solution $P(t) = 0$ for the entirety of the program's execution, as can be seen in Fig. 4.4. So by adding this extra term to the computation of the loss function for the first 50,000 epochs gives the approximation a small boost to escape the unstable solution $P(t) = 0$.

Van der Pol Oscillator

Our next example will be the van der Pol oscillator, a second-order differential equation that is a non-conservative oscillating system that evolves in time. This oscillating behavior is given by the differential equation

$$\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0,$$

where x is the position which depends on the time t, and μ is a parameter that indicates the nonlinearity and the strength of the damping.
This equation cannot be solved with analytic methods; thus, an exact solution is impossible to find. Therefore, we use numerical methods in order to approximate the solution to the differential equation.
In this case, we solve the equation with a parameter $\mu = 0.25$ on the interval $t \in [0, 20]$. With the initial conditions being $x(0) = 1$ and $x'(0) = 1$. Thus, the IVP we look to approximate is

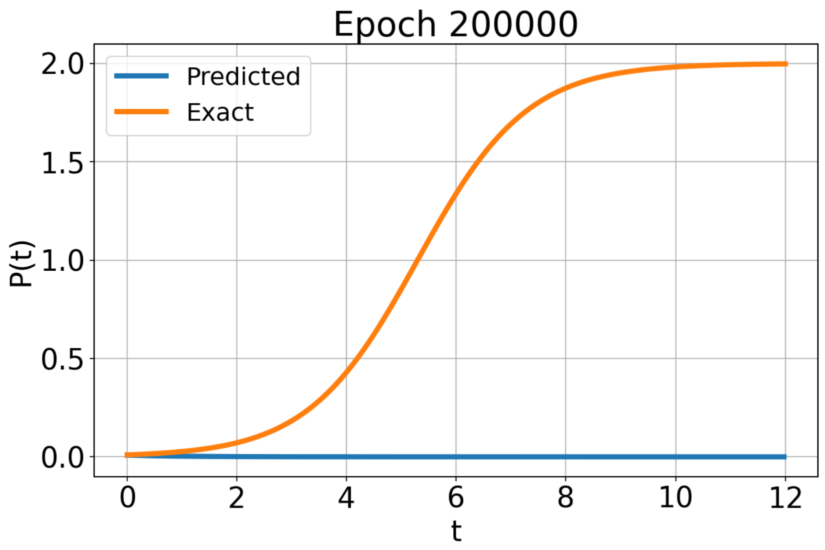


Figure 4.4: Approximation of logistic equation 4.4 without the additional condition.

$$\begin{aligned}\ddot{x} - 0.25(1 - x^2)\dot{x} + x = 0, \\ x(0) = 1\,;\ \dot{x}(0) = 0.\end{aligned} \tag{4.5}$$

The loss function for this example is thoroughly explained. Let $u^T(\theta, t)$ be a trial solution for the IVP. The physics loss part of the loss function is:

$$\mathcal{L}_{ODE} = \frac{1}{N_f}\sum_{i=1}^{N_f}\left(\ddot{u}^T(\theta, t_i) - 0.25(1 - u^T(\theta, t_i)^2)\dot{u}^T(\theta, t_i) + u^T(\theta, t_i)\right).$$

The data loss part of the loss function is:

$$\mathcal{L}_{IC1} = (u^T(\theta, t_0) - 1)^2.$$

And for the initial condition involving the derivative:

$$\mathcal{L}_{IC2} = (\dot{u}^T(\theta, t_0) - 0)^2.$$

For this example, $\omega_{ODE} = \omega_{IC1} = \omega_{IC2} = 1/3$. Then, the total loss function is:

$$\mathcal{L}_{total} = \frac{1}{3}\left(\mathcal{L}_{ODE} + \mathcal{L}_{IC1} + \mathcal{L}_{IC2}\right).$$

For this example, the structure of the network we employed consisted of 4 hidden layers, each one of them containing 60 neurons. We chose the hyperbolic tangent activation throughout the whole network; meanwhile, we utilized the Adam optimizer with the default learning rate of 1×10^{-3}. A total of $20,000$ uniformly spaced points were deployed in the interval $[0, 20]$. We ran our program for $50,000$ epochs to obtain a good approximation. Figure 4.5 shows the evolution of the approximation throughout the runtime compared to the ”exact” solution provided by the Runge-Kutta method of order 4 with a step-size of 0.001. We also show the values of the loss function at every 500 epochs to increase clarity.
Although a total of 50,000 epochs were run, around the 30,000th epoch, we already had a decent approximation.

Lotka-Volterra Equations

As seen in Section 2.6 of Chapter 1, the Lotka-Volterra equations are a pair of first-order differential equations used to describe the changes in population of a two-species biological environment, the predator and the prey. We now look to approximate both of these equations using the PINN method. Recalling both of these equations:

$$\begin{aligned} \dot{x} &= \alpha x - \beta xy, \\ \dot{y} &= -\gamma y + \delta xy, \end{aligned} \tag{4.6}$$

where α, β, γ and δ are real-valued parameters, used to describe how the predator and prey interact.
For this example, we look to solve this system with parameters $\alpha = 0.1, \beta = 0.002, \gamma = 0.2$ and $\delta = 0.0025$. Also, we name both species of the system; we use rabbits as the prey and foxes as the predator. The initial conditions of the system are the number of rabbits and foxes we begin with. In this case, we initially have 80 rabbits and 20 foxes; thus, the system of differential equations we look to solve is

$$\begin{aligned} \dot{x} &= 0.1x - 0.002xy, \\ \dot{y} &= -0.2y + 0.0025xy, \end{aligned} \tag{4.7}$$

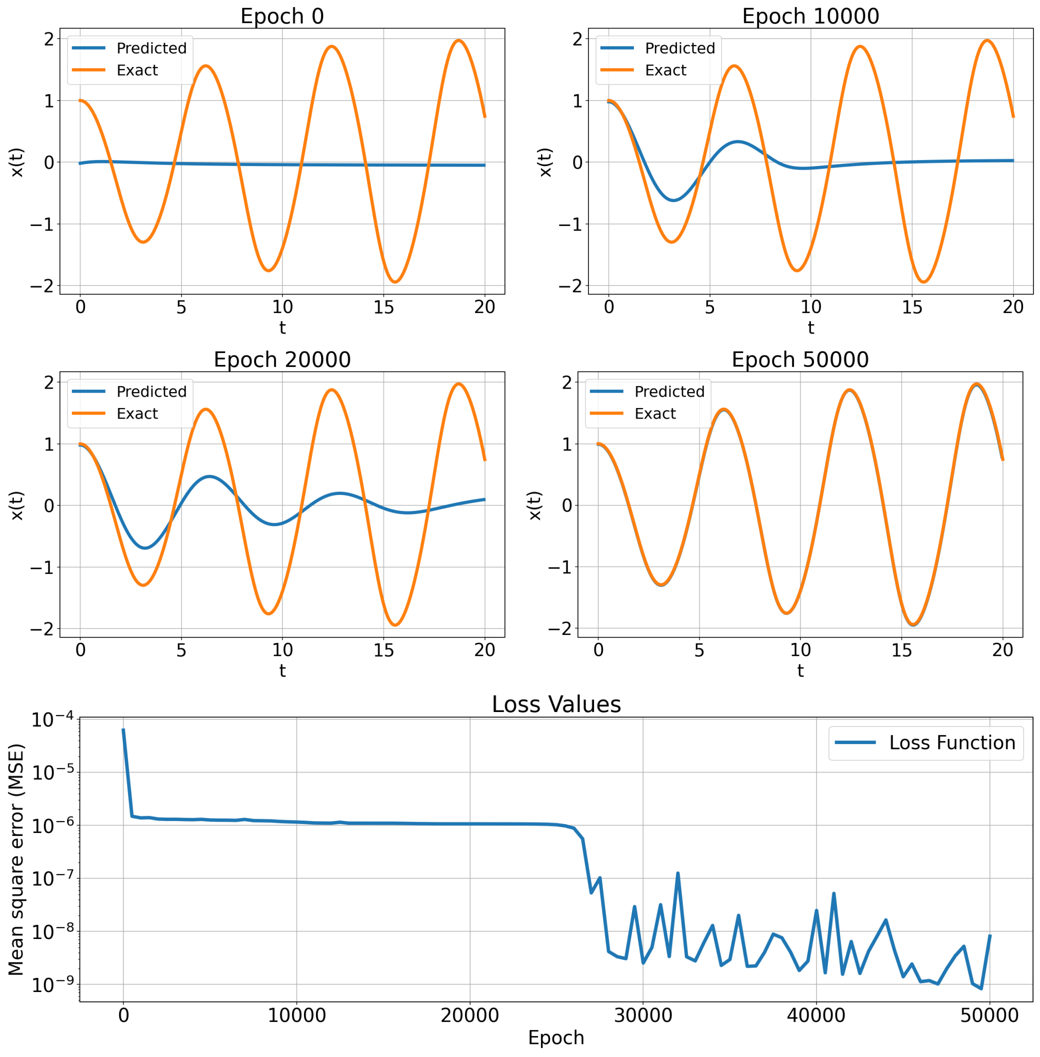


Figure 4.5: Evolution of approximations and values of the loss function every 500 epochs of the Van der Pol equation 4.5.

with $x(0) = 80$ and $y(0) = 20$, where x and y are the number of rabbits and foxes at any given time, respectively.
For this example, we approximate the system on the interval $[0, 50]$, where we deployed a total of 2000 uniformly spaced points. For the details of the neural network, we employed a network consisting of 5 hidden layers with a total of 120 nodes per layer. The sigmoid activation function was used throughout the whole network; meanwhile, the utilized optimizer was the Adam optimizer with the default learning rate 1×10^{-3}. The results are shown in Figure 4.6, where we compare the approximations with the "exact" solution provided by Euler's method with a step-size of 0.05. The loss function's values every 500 epochs are shown to increase clarity.
As shown in Fig. 4.6 the approximations stay as a straight line throughout the first 10,000 epochs. It isn't until epoch 20,000 that the approximations start to resemble the "exact" solution. We ran the program for a total of 100,000 epochs, but we obtained the best approximation on the 99000th epoch. The loss function behaves as desired, decreasing its values along the program's runtime.

4.3.2 Delay Differential Equations

Exponential Equation with Delay

The first DDE example we look at is the exponential differential equation, but with the addition of a delay to the equation. This delay differential equation is given by

$$\frac{dy}{dt} = ky(t - \tau),$$

where k is a parameter and τ indicates the delay of the differential equation. In this case, we solve the differential equation with a value of $k = -0.25$ on the interval $[0, 70]$ with a delay of $\tau = 15$. The initial condition function is given by $y(t) = 10$ for $t \leq 0$. Therefore, the delay differential equation we are looking to solve is

$$\begin{aligned} \frac{dy}{dt} &= -0.25y(t - 15), \\ y(t) &= 10 \ , \ t \in [-15, 0]. \end{aligned} \tag{4.8}$$

The loss function utilized for this example will be explained thoroughly. Let

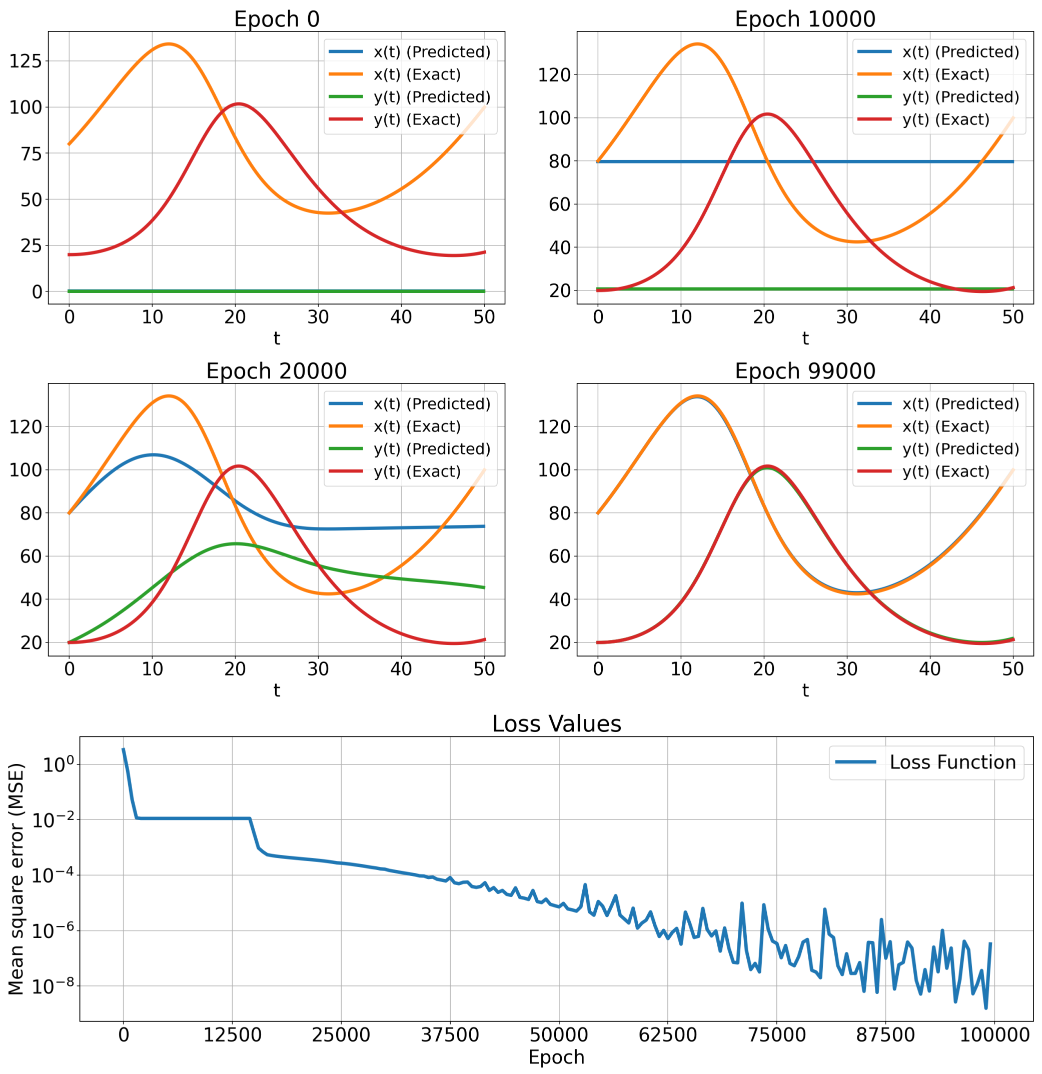


Figure 4.6: Approximation history and loss function values for the Lotka-Volterra system 4.7.

$u^T_{R_0}(\theta, t), u^T_{R_1}(\theta, t)$, and $u^T_{R_2}(\theta, t)$ be trial solutions of the DDE in their respective regions. For R_0:

$$\mathcal{L}_{R_0} = \frac{1}{S_0} \sum_{i=1}^{S_0} \left(u^T_{R_0}(\theta, t_i) - \phi(t_i) \right)^2 .$$

For R_1:

$$\mathcal{L}_{R_1} = \frac{1}{S_1} \sum_{i=1}^{S_1} \left(\dot{u}^T_{R_1}(\theta, t_i) - 0.25\phi(t_i - \tau) \right)^2 .$$

For R_2:

$$\mathcal{L}_{R_2} = \frac{1}{S_2} \sum_{i=1}^{S_2} \left(\dot{u}^T_{R_2}(\theta, t_i) - 0.25u^T_{R_2}(\theta, t_i) \right)^2 .$$

For this example we have $\omega_{R_0} = \omega_{R_1} = \omega_{R_2} = 1/3$. The total loss function for the DDE is:

$$\mathcal{L}_{total} = \frac{1}{3}(\mathcal{L}_{R_0} + \mathcal{L}_{R_1} + \mathcal{L}_{R_2}).$$

For the details of the network employed, we used a network consisting of 4 hidden layers with 50 nodes in each layer. We deployed a total of $S_0 = S_1 = 1100$ and $S_2 = 4000$ uniformly spaced points on the respective regions. We chose the hyperbolic tangent activation function throughout the whole network, and we used the Adam optimizer with the default learning rate of 1×10^{-3}. The evolution of the approximations across the 100,000 epochs we ran our program, as well as the values of the loss function, is found in Figure 4.7.

As can be seen in Fig. 4.7, the approximation starts as a straight line at the value $t = 0$ and slowly starts taking shape into the "exact" solution given by Euler's method with a step-size of 0.01. By the 90000th epoch, we already had a good approximation of the solution.

Logistic Equation with Delay

We look at the logistic equation with delay, which models the dynamics of populations. The delay logistic equation is given by

$$y'(t) = ay(t)(1 - y(t - \tau)), \tag{4.9}$$

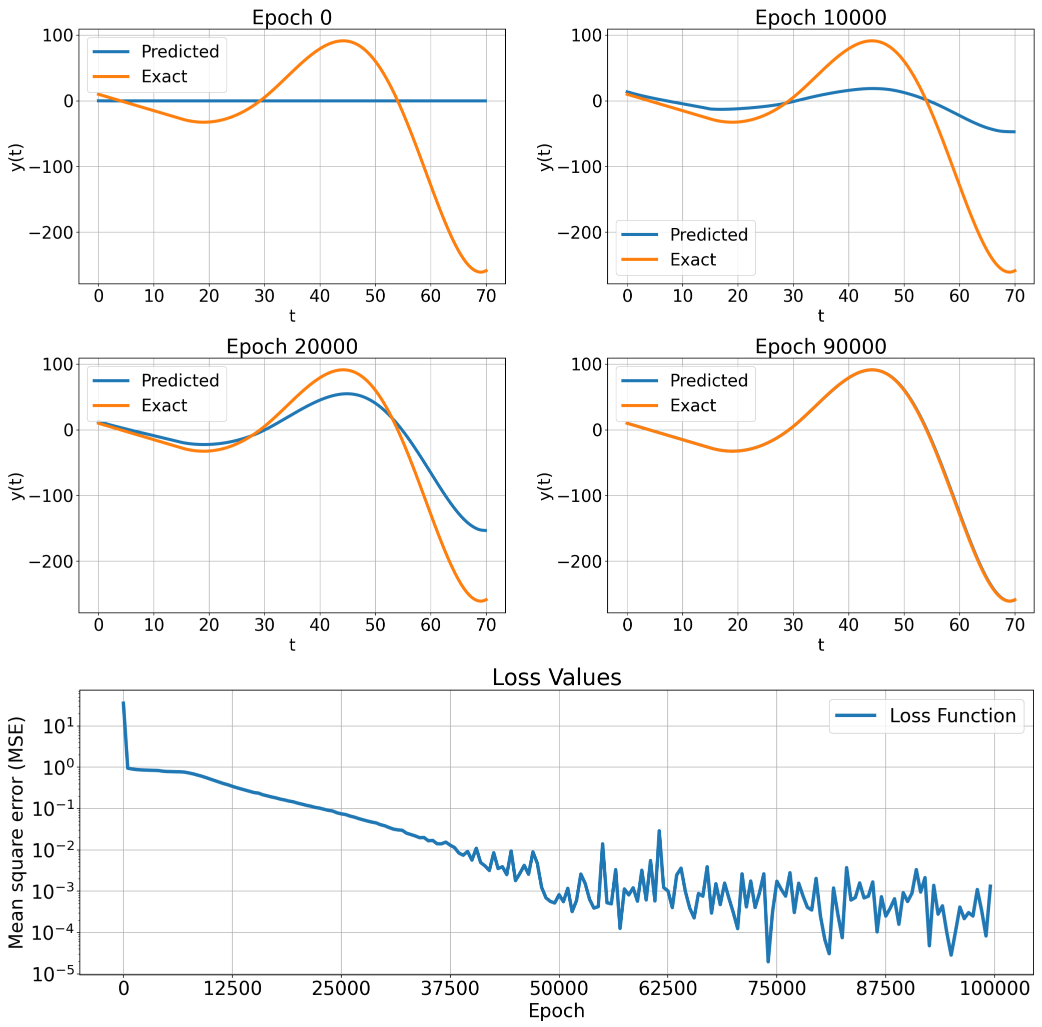


Figure 4.7: Evolution of the approximations and loss values of the delay exponential equation 4.8.

where a is a parameter and τ is the delay or lag of the equation.
In this case, we solve the equation with a value of $a = 1.4$ on the interval $t \in [0, 20]$ with a delay of $\tau = 1$, and an initial condition set by the function $y(t) = 0.1$ for $t \in [-\tau, 0]$ [5]

$$\begin{aligned} y'(t) &= 1.4y(t)(1 - y(t-1)), \\ y(t) &= 0.1 \ , \ -1 \leq t \leq 0. \end{aligned} \tag{4.10}$$

We sampled a total of $S_0 = S_1 = 250$ and $S_2 = 5000$ uniformly spaced points on the three different regions. As for the details of the network employed, it consisted of a network with 4 hidden layers, with each one of them containing 25 nodes. The chosen activation function this time was the sigmoid activation function, while the optimizer utilized was the Adam optimizer with a learning rate of 1×10^{-2}. The evolution of the approximation compared to the "exact" solution given by Euler's method for DDEs with a step-size of 0.004, and the behavior of the loss function can be seen in Figure 4.8.
We ran the program for a total of 140,000 epochs to achieve a fairly decent result. A similar trick, like in the case of the logistic equation without delay, was employed. We added an extra term to the computation of the loss function for the first 40,000 epochs for the approximation to escape the unstable solution $y(t) = 0$. If this additional term was not added, the approximation stays in the unstable solution $y(t) = 0$for the total of the program's execution, as evidenced by Fig. 4.9.
It is unknown whether or not the approximation will eventually converge to the solution given a larger number of epochs, but in our case, it does not seem like it will. As it is done in [5], we now make $a = 0.3$ in Eq. 4.9 and keep everything else the same as in the previous case.

$$\begin{aligned} y'(t) &= 0.3y(t)(1 - y(t-1)), \\ y(t) &= 0.1 \ , \ -1 \leq t \leq 0. \end{aligned} \tag{4.11}$$

We also keep the same network configuration as the previous case, but now we only run the program for 20,000 epochs. The development of the approximations, as well as the loss function's values, can be found in Figure 4.10.
As can be seen in Fig. 4.10 the approximation converges almost immediately to the "exact" solution given by Euler's method with a step-size of 0.003. Like in the previous case, we employed the same trick of using a temporary

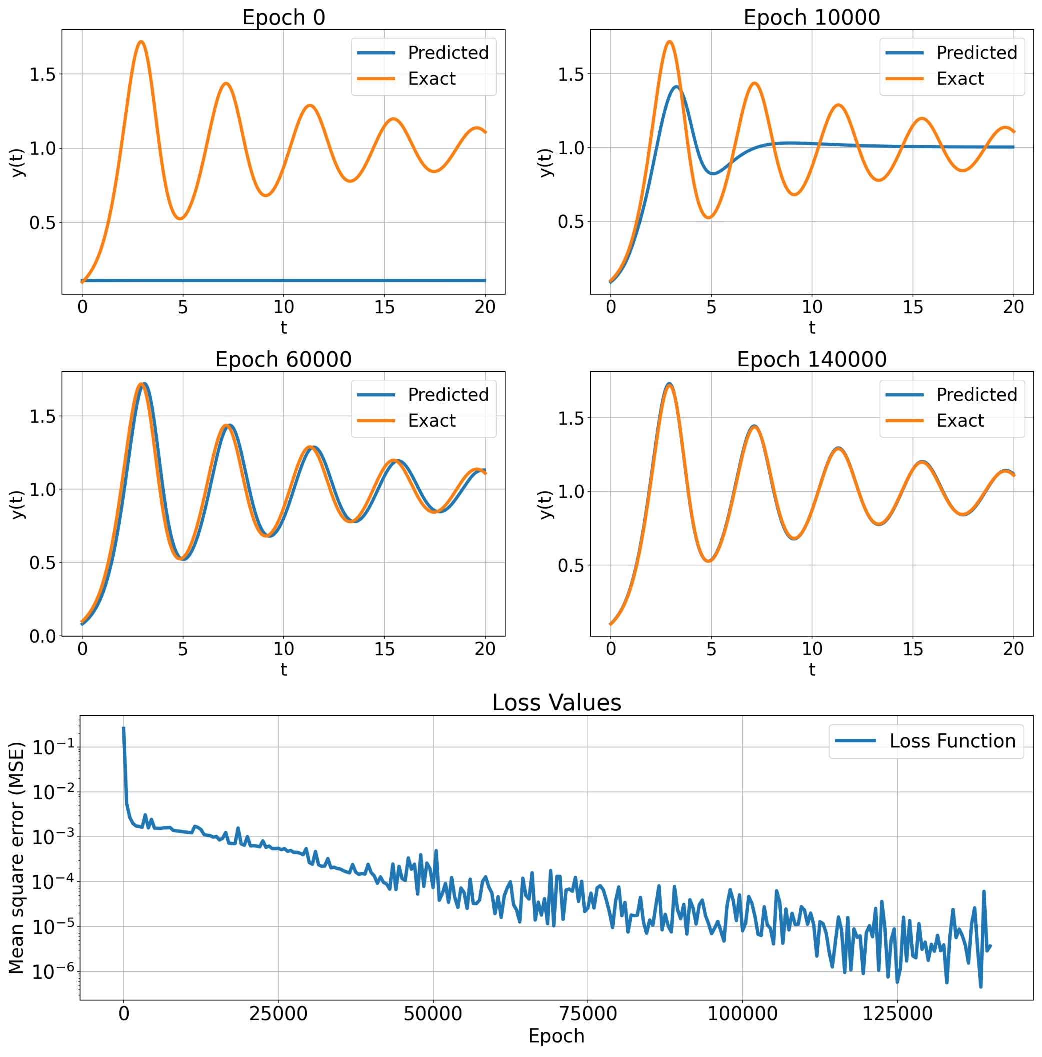


Figure 4.8: Evolution of the approximations and loss values of the logistic delay differential equation 4.10.

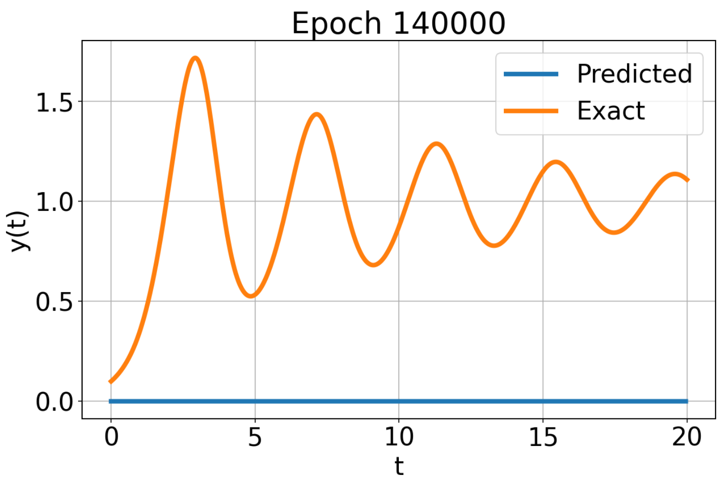


Figure 4.9: Approximation of logistic equation with delay 4.9 without the boost.

condition in the computation of the loss function to boost the approximation out of the unstable solution $y(t) = 0$. If we did not include this short-term condition in the loss function, the approximations stay in the unstable solution $y(t) = 0$ for the duration of the program's runtime, as evidenced by Fig. 4.11.
As in the previous case, it is unknown whether or not the approximation will eventually converge to the solution given a larger number of epochs. But it is highly unlikely it will.

Lotka-Volterra Equations with Delay

We now incorporate a delay into the Lotka-Volterra equations. Firstly, a brief explanation of the reasoning behind adding the delay. We add a delay to the α parameter, which represents the maximum per capita growth rate of the prey. The addition of this delay makes sense biologically speaking, because animals take time to mature and become able to reproduce. This delay represents the time between when an animal is born and the time they become reproductively capable, which is when they start to contribute to the population's dynamics.

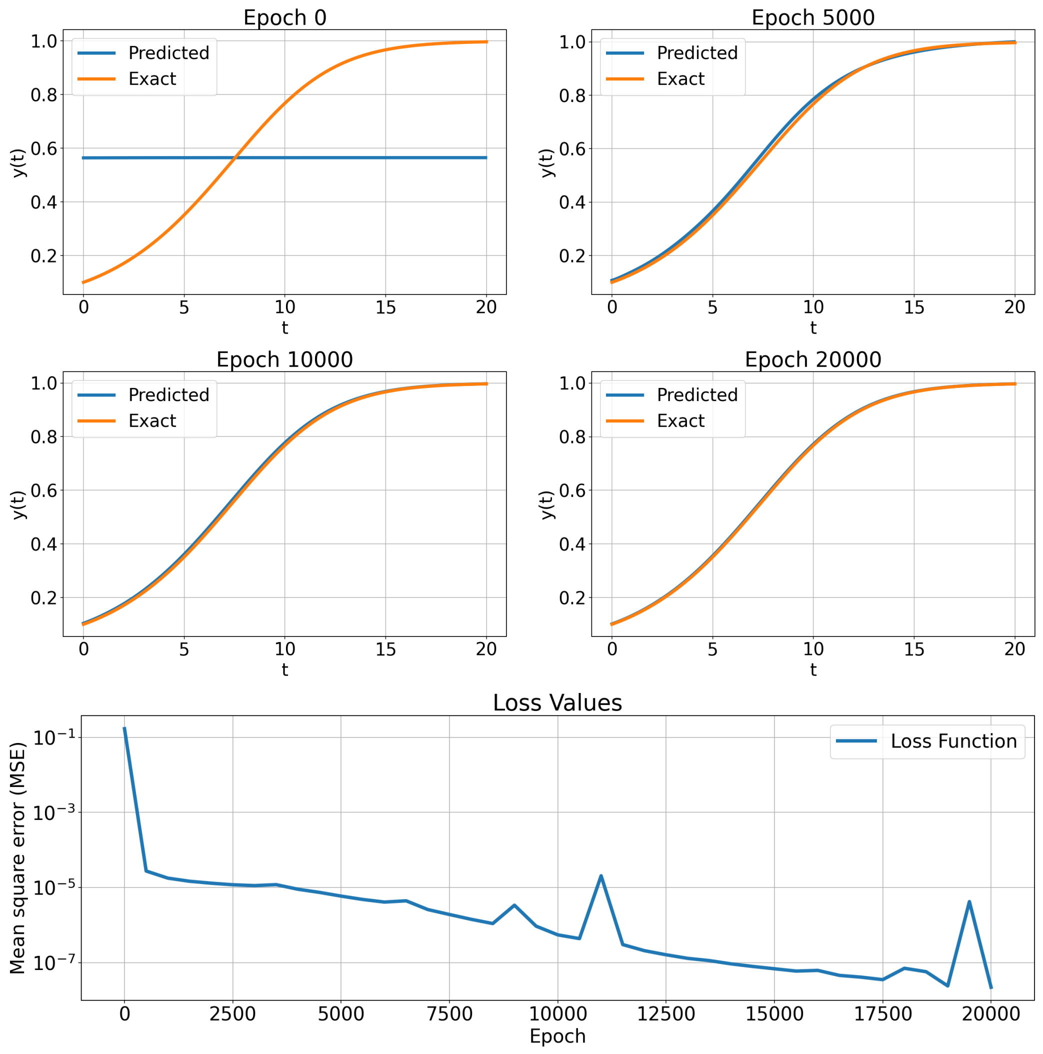


Figure 4.10: Evolution of approximations and loss function values of logistic delay differential equation 4.11.

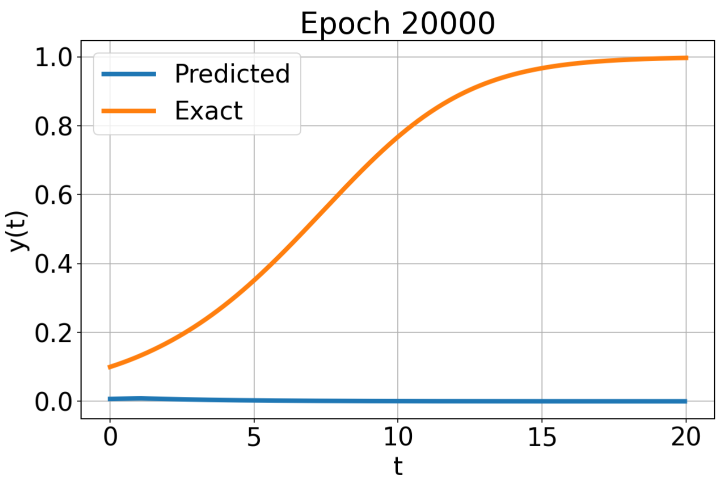


Figure 4.11: Approximation of logistic equation 4.11 without the boost.

The Lotka-Volterra equations with delay on the reproduction term are

$$\begin{aligned}\dot{x} &= \alpha x(t-\tau) - \beta x(t)y(t),\\ \dot{y} &= -\gamma y(t) + \delta x(t)y(t).\end{aligned} \tag{4.12}$$

For this example, we keep the same species and parameters from the Lotka-Volterra without delay example we presented before. These parameters are $\alpha = 0.1, \beta = 0.002, \gamma = 0.2$ and $\delta = 0.0025$. Therefore, the system we want to approximate is

$$\begin{aligned}\dot{x} &= 0.1x(t-\tau) - 0.002x(t)y(t),\\ \dot{y} &= -0.2y(t) + 0.0025x(t)y(t),\end{aligned} \tag{4.13}$$

where the initial condition function for x is $\phi(t) = 80$ for $x \leq 0$ and the initial condition for y is $y(0) = 20$. In this case, x represents the number of rabbits and y represents the number of foxes at any given time. We solve this system on the $[0, 52]$ interval, with a value of $\tau = 16$.
For the implementation of this example, we used a two neural network design. Each network represents one of the variables of our system. The input for our networks is the same $[-\tau, T]$ interval, but split into three parts, just as we discussed previously.
The loss functions used for this example are explicitly explained next. We used two neural networks for this example; hence, we define two loss functions, one for each network.

Let $u^T_{R_0}(\theta,t), u^T_{R_1}(\theta,t)$, and $u^T_{R_2}(\theta,t)$ be trial solutions of the equation corresponding to the x variable, in their respective regions. Also, let $\mathcal{L}_{R_0}, \mathcal{L}_{R_1}$, and $\mathcal{L}_{R_2}$ be the loss function values for this equation.
Similarly, let $\mathfrak{u}^T_{R_0}(\theta,t), \mathfrak{u}^T_{R_1}(\theta,t)$, and $\mathfrak{u}^T_{R_2}(\theta,t)$ be trial solutions of the equation corresponding to the y variable. Also, let $\mathfrak{L}_{R_0}, \mathfrak{L}_{R_1}$, and $\mathfrak{L}_{R_2}$ be the loss function values for this equation.
For the R_0 region of the x equation and the initial condition of the y equation:

$$\mathcal{L}_{R_0} = \frac{1}{S_0}\sum_{i=1}^{S_0}\left(u_{R_0}(\theta,t_i) - \phi(t_i)\right)^2;$$

$$\mathfrak{L}_{IC} = (\mathfrak{u}^T_{R_1}(\theta,t_i) - 20)^2.$$

For R_1 of both equations:

$$\mathcal{L}_{R_1} = \frac{1}{S_1}\sum_{i=1}^{S_1}\left(\dot{u}^T_{R_1}(\theta,t_i) - 0.1\phi(t_i-\tau) + 0.002u^T_{R_1}(\theta,t_i)\mathfrak{u}^T_{R_1}(\theta,t_i)\right)^2;$$

$$\mathfrak{L}_{R_1} = \frac{1}{S_1}\sum_{i=1}^{S_1}\left(\dot{\mathfrak{u}}^T_{R_1}(\theta,t_i) + 0.2\mathfrak{u}^T_{R_1}(\theta,t_i) - 0.0025u^T_{R_1}(\theta,t_i)\mathfrak{u}^T_{R_1}(\theta,t_i)\right)^2.$$

Finally, for the R_2 region:

$$\mathcal{L}_{R_2} = \frac{1}{S_2}\sum_{i=1}^{S_2}\left(\dot{u}^T_{R_2}(\theta,t_i) - 0.1u^T_{R_2}(\theta,t_i-\tau) + 0.002u^T_{R_2}(\theta,t_i)\mathfrak{u}^T_{R_2}(\theta,t_i)\right)^2;$$

$$\mathfrak{L}_{R_2} = \frac{1}{S_2}\sum_{i=1}^{S_2}\left(\dot{\mathfrak{u}}^T_{R_2}(\theta,t_i) + 0.2\mathfrak{u}^T_{R_2}(\theta,t_i) - 0.0025u^T_{R_2}(\theta,t_i)\mathfrak{u}^T_{R_2}(\theta,t_i)\right)^2.$$

The total loss function for the x variable equation is:

$$\mathcal{L}_{total} = \frac{1}{3}(\mathcal{L}_{R_0} + \mathcal{L}_{R_1} + \mathcal{L}_{R_2}).$$

And, the total loss function for the y variable equation is:

$$\mathfrak{L}_{total} = \frac{1}{3}(\mathfrak{L}_{IC} + \mathfrak{L}_{R_1} + \mathfrak{L}_{R_2}).$$

For the details of the networks employed, we used the same network structure for both networks. We utilized two networks consisting of 5 hidden layers, each one of them containing a total of 120 nodes. The activation function used throughout both networks was the sigmoid activation function, while the chosen optimizer was the Adam optimizer with the default learning rate of 1×10^{-3} for both of them. We sampled a total of $S_0 = S_1 = 2000$ and $S_2 = 4500$ uniformly spaced points for each of the three regions. We ran our program for a total of $40,000$ epochs. The evolution of the approximation throughout the execution time compared with the "exact" solution given by Euler's method for DDEs with a step-size of 0.008, as well as both of the network's loss function values, can be found in Figure 4.12.
As shown in Fig. 4.12, the approximations converged faster than the example we presented earlier, where we did not incorporate delay, but used the same parameters. This may be caused by the initial parameters of the neural networks; in this case, we might have been closer to a minimum than in the previous example. By the 35000th epoch, we already had a good approximation of the solution of the system. As for the behavior of the loss functions, they both follow a very similar behavior throughout the execution time of the program. In some cases, they even overlap with each other.

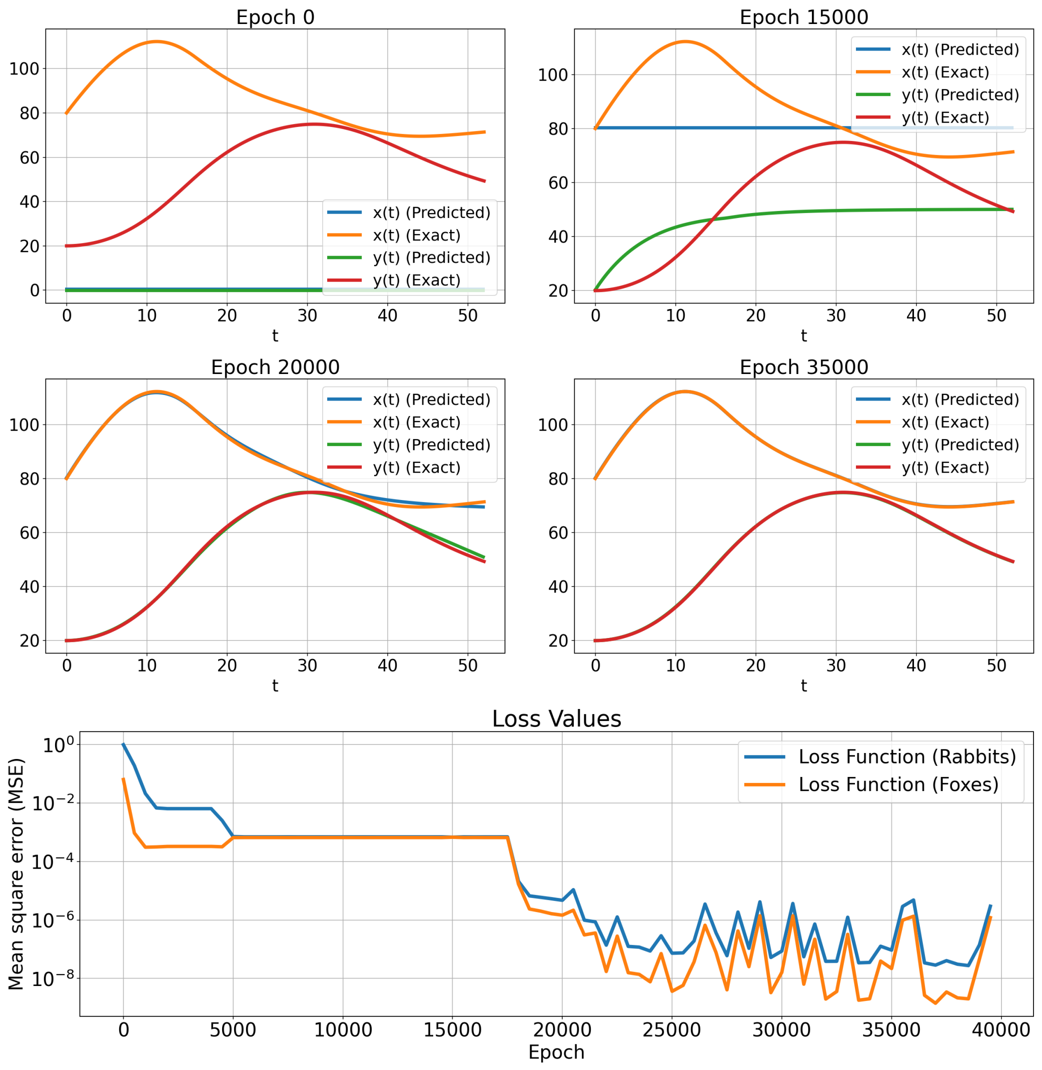


Figure 4.12: Evolution of approximations and loss function values of Lotka-Volterra system with delay 4.13.

Chapter 5

Conclusions and Discussion

In this thesis, we have reviewed the Physics-Informed Neural Network method for solving ordinary differential equations and delay differential equations, and how it is an effective and powerful method for approximating these types of equations.
One of the advantages of using this method is the small amount of data that is needed to train the neural network. The only data needed to train the neural network using the PINN method is the differential equation itself and its initial conditions. This is vastly different than other applications of neural networks, like image recognition, where a large amount of data is needed to properly train the neural network.
Experimentation is needed to utilize this method effectively. We employed the L-BFGS optimization algorithm when we first approached this method. Because this optimizer's implementation is not straightforward compared with the implementation of the Adam optimizer, we decided to stop using it and kept using Adam through the entirety of this thesis.
Another thing we noted is that the hyperbolic tangent activation function seemed to perform better than the sigmoid function for equations whose solutions have oscillations, like the Van der Pol oscillator and the logistic equation with delay. We do not know if this is intended, but further research is needed to claim the validity of this conjecture.
The flexibility of this method is noteworthy; the PINN method can be used to approximate various types of differential equations. In this thesis, we only explored examples of DDEs and ODEs, but it can also be used to approximate PDEs. With only small modifications to the base code for approximating ODEs, it was also possible to use it with DDEs.

One advantage the PINN method has over traditional methods like the Runge-Kutta methods is the ability to work with higher-order differential equations without the need to convert them into a system of first-order differential equations. Not having to convert the differential equation into a system of differential equations is especially useful when working with higher-order differential equations.

One of the main drawbacks of using this method is that there is no established blueprint or recipe you can follow that guarantees the approximations' convergence to the solution. When using this method, one has to employ a trial-and-error approach. Carefully inspect the behavior of the approximations and loss function to learn how the method is performing. Adding to this, the PINN method is not an all-purpose method, meaning there is a lack of a one-size-fits-all network configuration that can be used for every differential equation. As we explored in Chapter 4 of this thesis, each one of the examples we showed used a different setup from the others. So this means that even if you find an optimal network configuration and optimizer choice for a differential equation, it does not imply that this setup will work for a different differential equation.

Another drawback of PINN is the computational power needed to use this method. Optimizing the neural networks requires a lot of computational power, especially GPUs. When working with more complex problems, not having access to optimal equipment may lead to problems like having large runtimes. As this is an open topic, research is still being conducted, so optimizations for this method can still be found. This, combined with technological advancements in the area of GPUs, might make this stop being an issue in the future.

Appendix A

Proof of theorem 2.1

Before the theorem's proof we look at some preliminary results.

Definition A.1 (Cauchy Sequence)**.** A sequence $\{p_n\}$ in a normed linear space X is said to be Cauchy in X if for each $\epsilon > 0$, there is a natural number N such that for all $m, n \geq N$

$$||p_n - p_m|| < \epsilon.$$

Definition A.2 (Complete Space)**.** A normed linear space X is said to be complete if every Cauchy sequence in X converges to an element in X.

Definition A.3 (Lipschitz Condition)**.** Let U be an open subset of $\mathbb{R}^n$. A function $f : U \to \mathbb{R}^n$ satisfies the Lipschitz condition if there exists a constant $K > 0$ such that

$$|f(x) - f(y)| \leq K|x - y| \qquad \forall x, y \in U.$$

Lemma A.1. Let U be an open subset of $\mathbb{R}^n$ and $f : U \to \mathbb{R}^n$. If $f \in C^1(U)$, then f is locally Lipschitz on U.

Proof. Let x_0 be an arbitrary point in U. Since $U \subset \mathbb{R}^n$ is open, there exists an open ball of radius $\epsilon > 0$, such that $B_\epsilon(x_0) \subset U$,

$$B_\epsilon(x_0) = \{x \in \mathbb{R}^n \ : \ |x - x_0| < \epsilon\} \subset U.$$

Since $B_\epsilon(x_0)$ is open, we can find a closed ball of radius $\epsilon/2$ such that $B_{\epsilon/2}(x_0) \subset B_\epsilon(x_0)$. Thus $B_{\epsilon/2}(x_0)$ is compact.

Let K be the maximum of $f'(x)$ on $B_{\epsilon/2}(x_0)$,

$$K = \max_{|x-x_0|\leq\epsilon/2} ||f'(x)||.$$

Set $u = y - x$, for $x, y \in B_{\epsilon/2}(x_0)$. Then, $x + su \in B_{\epsilon/2}(x_0)$ for $0 \leq s \leq 1$, since $B_{\epsilon/2}(x_0)$ is a convex set.
Let $F : [0,1] \to \mathbb{R}^n$ such that

$$F(s) = f(x + su).$$

Applying the chain rule to $F(s)$ we have

$$F'(s) = f'(x + su) \cdot u.$$

Then by the fundamental theorem of calculus,

$$f(y) - f(x) = F(1) - F(0) = \int_0^1 F'(s)\, ds = \int_0^1 f'(x + su) \cdot u\, ds.$$

Applying integral properties and inequalities,

$$\begin{aligned}|f(y) - f(x)| &= \left|\int_0^1 f'(x + su) \cdot u\, ds\right| \\ &\leq \int_0^1 |f'(x + su) \cdot u|\, ds \\ &\leq \int_0^1 \|f'(x + su)\|\, |u|\, ds \\ &\leq K|u| = K|y - x|\end{aligned}$$

Therefore, f is locally Lipschitz on U. □

The proof of the next three theorems can be found in Rudin [27].

Theorem A.1
Let $\{f_n\}$ be a sequence of functions defined on a set E. Then, $\{f_n\}$ converges uniformly on E if and only if for every $\epsilon > 0$ there exists an integer N such that $m, n \geq N, x \in E$, implies that

$$|f_n(x) - f_m(x)| \leq \epsilon.$$

Theorem A.2
Suppose that $f_n \to f$ uniformly on a set E in a metric space. Let x be a limit point of E, and suppose that

$$\lim_{t \to x} f_n(t) = A_n.$$

Then $\{A_n\}$ converges, and

$$\lim_{t \to x} f(t) = \lim_{n \to \infty} A_n.$$

Theorem A.3
If $\{f_n\}$ is a sequence of functions on a set E, and if $f_n \to f$ uniformly on E, then f is continuous on E.

Now we prove the theorem.

Theorem A.4
Let E be an open subset of $\mathbb{R}^n$ that contains x_0 and assume $f \in C^1(E)$. Then there exists an $a > 0$ such that the initial value problem

$$\begin{aligned} \dot{x} &= f(x) \\ x(0) &= x_0 \end{aligned}$$

has a unique solution $x(t)$ on the interval $[-a, a]$.

Proof. Since $f \in C^1(E)$, by lemma A.1 there exists a neighborhood around x_0, $B_\epsilon(x_0)$, such that $B_\epsilon(x_0) \subset E$, and a constant $K > 0$ such that

$$|f(x) - f(y)| \leq K|x - y| \qquad \forall x, y \in B_\epsilon(x_0).$$

Let $b = \epsilon/2$ and $B_0 = \{x \in \mathbb{R}^n \mid |x - x_0| \leq b\}$. Since f is continuous and B_0 is a compact set, then f is bounded on B_0. Furthermore, f reaches its maximum value on B_0, let

$$M = \max_{x \in B_0} |f(x)|.$$

Let $u_k(t)$ be a sequence of functions defined by Picard's method of successive approximations ($u_{k+1}(t) = x_0 + \int_0^t f(u_k(s))\, ds$). Assuming that there exists an $a > 0$ such that $u_k(t)$ is defined and continuous on $[-a, a]$ and satisfies

$$\max_{t \in [-a, a]} |u_k(t) - x_0| \leq b. \tag{A.1}$$

By the last statement and because f is continuous it follows that $f(u_k(t))$ is defined and continuous on $[-a, a]$. Then the next approximation $u_{k+1}(t)$, given by

$$u_{k+1}(t) = x_0 + \int_0^t f(u_k(s))\, ds$$

is defined and continuous on $[-a, a]$.
Now lets see that $|u_{k+1} - x_0|$ is bounded. By the definition of u_{k+1} we have

$$\begin{aligned} |u_{k+1} - x_0| &= \left| \int_0^t f(u_k(s))\, ds \right| \\ &\leq \int_0^t |f(u_k(s))|\, ds \\ &\leq \int_0^t M\, ds \\ &= Mt \\ &\leq Ma. \end{aligned}$$

Therefore, $|u_{k+1} - x_0|$ is bounded.
Choosing a such that $0 \leq a \leq b/M$, it follows by induction that $u_k(t)$ is defined, is continuous and satisfies equation A.1 for all $t \in [-a, a]$ and $k = 1, 2, 3, \ldots$.
Now, since interval $[-a, a]$ is in $B_0 \subset B_\epsilon(x_0)$, then f is Lipschitz for all $t \in [-a, a]$. In particular, taking $u_1(t)$ and $u_2(t)$, and doing $u_2(t) - u_1(t)$ we have

$$\begin{aligned} |u_2(t) - u_1(t)| &= \left| \int_0^t f(u_1(s)) - f(u_0(s)) \right| ds \\ &\leq \int_0^t |f(u_1(s)) - f(u_0(s))|\, ds \\ &\leq \int_0^t K|u_1(s) - u_0(s)|\, ds \\ &= K \int_0^t |u_1(s) - u_0(s)|\, ds. \end{aligned}$$

Notice that $u_0(s) = x_0$, then

$$|u_2(t) - u_1(t)| \leq K \int_0^t |u_1(s) - x_0|\, ds.$$

By equation A.1 it follows that

$$\begin{aligned}|u_2(t)-u_1(t)| &\le K\int_0^t b\,ds\\ &= Kb\int_0^t ds\\ &= Kbt\\ &\le Kba.\end{aligned}$$

We have shown that $|u_2(t)-u_1(t)|$ is bounded. Now we show that the general case is bounded as well.
Assume that for some integer $j \ge 2$

$$\max_{t\in[-a,a]} |u_j(t)-u_{j-1}(t)| \le (Ka)^{j-1}b. \tag{A.2}$$

Taking $u_{j+1}(t)$ and $u_j(t)$, we will show that $|u_{j+1}(t)-u_j(t)|$ is bounded.

$$\begin{aligned}|u_{j+1}(t)-u_j(t)| &= \left|\int_0^t f(u_j(s))-f(u_{j-1}(s))\right| ds\\ &\le \int_0^t |f(u_j(s))-f(u_{j-1}(s))|\,ds\\ &\le \int_0^t K|u_j(s)-u_{j-1}(s)|\,ds\\ &= K\int_0^t |u_j(s)-u_{j-1}(s)|\,ds.\end{aligned}$$

By equation A.2 it follows that

$$\begin{aligned}|u_{j+1}(t)-u_j(t)| &\le K\int_0^t |u_j(s)-u_{j-1}(s)|\,ds\\ &\le K\int_0^t \max_{t\in[-a,a]} |u_j(t)-u_{j-1}(t)|\,ds\\ &\le K\int_0^t (Ka)^{j-1}b\,ds\\ &= K(Ka)^{j-1}b\int_0^t ds\\ &= K(Ka)^{j-1}bt\\ &\le K(Ka)^{j-1}ba = Ka(Ka)^{j-1}b = (Ka)^j b.\end{aligned}$$

Thus, it follows by induction that equation A.2 holds for $j = 2, 3, \ldots$. Therefore, $|u_{j+1}(t) - u_j(t)|$ is bounded.
Setting $\alpha = Ka$, then $|u_{j+1}(t) - u_j(t)| \leq \alpha^j b$. Choosing a such that $0 < a < 1/K$. We prove that $u_k(t)$ is a Cauchy sequence.
Let $\epsilon > 0, m > k \geq N$ and $t \in [-a, a]$. We can write $u_m(t) - u_k(t)$ as a sum

$$u_m(t) - u_k(t) = \sum_{j=k}^{m-1} u_{j+1}(t) - u_j(t).$$

Then, taking the absolute value and using the triangle inequality yields

$$\begin{aligned} |u_m(t) - u_k(t)| &= \left| \sum_{j=k}^{m-1} u_{j+1}(t) - u_j(t) \right| \\ &\leq \sum_{j=k}^{m-1} |u_{j+1}(t) - u_j(t)| \\ &\leq \sum_{j=N}^{\infty} |u_{j+1}(t) - u_j(t)| \\ &\leq \sum_{j=N}^{\infty} \alpha^j b = b \sum_{j=N}^{\infty} \alpha^j. \end{aligned}$$

Notice this last series is a geometric convergent series because $\alpha = Ka < 1$. Therefore

$$|u_m(t) - u_k(t)| = b\frac{\alpha^N}{1-\alpha}.$$

Because $\alpha < 1$, the right side of this last equality approaches zero as $N \to \infty$. Therefore, for all $\epsilon > 0$ there exists N such that $m, k \geq N$ implies

$$||u_m - u_k|| = \max_{t \in [-a,a]} |u_{m(t)} - u_k(t)| < \epsilon.$$

This proves that the sequence $\{u_k\}$ is a Cauchy sequence of continuous functions in $C([-a, a])$. By Theorems A.1 and A.3, $u_k(t)$ converges uniformly to a continuous function $u(t)$ for all $t \in [-a, a]$.
Thus,

$$u(t) = \lim_{k \to \infty} u_k(t).$$

Making $k \to \infty$ of both sides in the definition of the u_{k+1}'th term of successive approximations

$$\lim_{k\to\infty} u_{k+1}(t) = \lim_{k\to\infty} \left(x_0 + \int_0^t f(u_k(s))\, ds \right).$$

This is:

$$u(t) = x_0 + \lim_{k\to\infty} \left(\int_0^t f(u_k(s))\, ds \right).$$

By Theorem A.2 we have

$$u(t) = x_0 + \int_0^t \lim_{k\to\infty} f(u_k(s))\, ds.$$

$$u(t) = x_0 + \int_0^t f(u(s))\, ds. \tag{A.3}$$

Since $u(t)$ is continuous then $f(u(t))$ is continuous, and by the fundamental theorem of calculus equation A.3 is differentiable and

$$u'(t) = f(u(t)) \qquad \forall t \in [-a, a].$$

Furthermore, $u(0) = x_0$. From equation A.1 it follows that

$$\max_{t\in[-a,a]} |u_k(t) - x_0| \le b.$$

This is $u(t) \in B_\epsilon(x_0) \subset E$ for all $t \in [-a, a]$. Thus, $u(t)$ is a solution of the initial value problem on $[-a, a]$. It remains to show that the solution is unique.
Let $v(t)$ and $u(t)$ be two solutions of the initial value problem on $[-a, a]$. The function $|u_t - v_t|$ is continuous in the compact set $[-a, a]$. Furthermore, it reaches its maximum at some point $t_1 \in [-a, a]$

$$\max_{t\in[-a,a]} |u(t) - v(t)| = |u(t_1) - v(t_1)|.$$

Then

$$\begin{aligned}
||u-v|| &= \max_{t\in[-a,a]} |u(t)-v(t)| \\
&= \left|\int_0^{t_1} f(u(s)) - f(v(s))\, ds\right| \\
&\le \int_0^{|t_1|} |f(u(s)) - f(v(s))|\, ds \\
&\le K\int_0^{|t_1|} |u(s)-v(s)|\, ds \\
&\le Ka \max_{t\in[-a,a]} |u(t)-v(t)| \\
&\le Ka||u-v||
\end{aligned}$$

Suppose $||u-v|| > 0$, then we can divide this last inequality by $||u-v||$

$$\begin{aligned}
||u-v|| &\le Ka||u-v|| \\
1 &< Ka \Rightarrow\!\Leftarrow
\end{aligned}$$

This is a contradiction because $Ka < 1$. Thus

$$||u-v|| = 0.$$

Which implies

$$|u(t)-v(t)| = 0 \qquad \forall t \in [-a,a].$$

Then, $u(t) = v(t)$ for all $t \in [-a,a]$. Therefore, the successive approximations converge uniformly to a unique solution of the initial value problem on the interval $[-a,a]$, where a is any number satisfying $0 < a < \min\left(\frac{b}{M}, \frac{1}{K}\right)$. □

Bibliography

[1] C. C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*, chapter 4. Springer Nature, Cham, Switzerland, 2 edition, June 2023. ISBN 9783031296420.

[2] B. Basics. The life and death of a neuron— national institute of neurological disorders and stroke, 2019.

[3] P. Baxandall and H. Liebeck. *Vector Calculus*, chapter 3. Dover Books on Mathematics. Dover Publications, Mineola, NY, 2008. ISBN 0486466205.

[4] A. Baydin, B. Pearlmutter, A. Radul, and J. Siskind. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18:1–43, 04 2018.

[5] A. Bellen and M. Zennaro. *Numerical Methods for Delay Differential Equations*, chapter 1. Numerical Mathematics and Scientific Computation. Clarendon Press, Oxford, England, Apr. 2003. ISBN 0198506546.

[6] M. Braun. *Differential Equations and Their Applications*, chapter 3. Texts in Applied Mathematics. Springer, New York, NY, 4 edition, Dec. 1992. ISBN 9780387943305.

[7] R. L. Burden and J. D. Faires. *Numerical Analysis*, chapter 5. Brooks/Cole, Cengage Learning, ninth edition, 2011. ISBN 0538733519.

[8] S. L. Campbell and R. Haberman. *Introduction to Differential Equations with Dynamical Systems*, chapter 1. Princeton University Press, Princeton, NJ, 2008. ISBN 9780691124742.

[9] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl. On empirical comparisons of optimizers for deep learning. *CoRR*, abs/1910.05446, 2019. URL `http://arxiv.org/abs/1910.05446`.

[10] R. Courant and F. John. *Introduction to Calculus and Analysis: Volume II*, chapter 2. Springer New York, NY, New York, NY, oct 2011. ISBN 9781461389583. URL `http://dx.doi.org/10.1007/978-1-4613-8958-3`.

[11] C. F. Curtiss and J. O. Hirschfelder. Integration of stiff equations. *Proceedings of the National Academy of Sciences*, 38(3):235–243, Mar. 1952. ISSN 1091-6490. URL `http://dx.doi.org/10.1073/pnas.38.3.235`.

[12] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, Dec. 1989. ISSN 1435-568X. URL `http://dx.doi.org/10.1007/BF02551274`.

[13] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503:92–108, Sept. 2022. ISSN 0925-2312. URL `https://doi.org/10.1016/j.neucom.2022.06.111`.

[14] I. R. Epstein and J. A. Pojman. *An Introduction to Nonlinear Chemical Dynamics: Oscillations, Waves, Patterns, and Chaos*, chapter 1. Oxford University Press, 11 1998. ISBN 9780195096705. URL `https://doi.org/10.1093/oso/9780195096705.001.0001`.

[15] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*, chapter 6. MIT Press, 2016. http://www.deeplearningbook.org.

[16] K. Gurney. *An Introduction to Neural Networks*, chapter 1. CRC Press, Oct. 2018. ISBN 9781482286991. URL `http://dx.doi.org/10.1201/9781315273570`.

[17] M. Islam, G. Chen, and S. Jin. An overview of neural network. *American Journal of Neural Networks and Applications*, 5(1):7, 2019. ISSN 2469-7400. URL `http://dx.doi.org/10.11648/j.ajnna.20190501.12`.

[18] E. M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*, chapter 2. The MIT Press, July 2006. ISBN

9780262276078. URL `http://dx.doi.org/10.7551/mitpress/2526.001.0001`.

[19] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[20] H. Kinsley and D. Kukieła. *Neural Networks from Scratch (NNFS)*, chapter 4. Harrison Kinsley, 2020. URL `https://nnfs.io/`.

[21] I. Lagaris, A. Likas, and D. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998. ISSN 1045-9227. URL `http://dx.doi.org/10.1109/72.712178`.

[22] M. A. Nielsen. *Neural Networks and Deep Learning*, chapter 1. Determination Press, 2015.

[23] L. Perko. *Differential equations and dynamical systems*, chapter 2. Texts in Applied Mathematics. Springer, New York, NY, 3 edition, Feb. 2008. ISBN 0387951164.

[24] S. J. Prince. *Understanding Deep Learning*, chapter 6. The MIT Press, 2023. URL `http://udlbook.com`.

[25] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, Feb. 2019. ISSN 0021-9991. URL `https://www.sciencedirect.com/science/article/pii/S0021999118307125`.

[26] A. Ralston and P. Rabinowitz. *A First Course in Numerical Analysis*, chapter 5. Dover Books on Mathematics. Dover Publications, second edition, 2001. ISBN 9780486414546. LCCN 00064343.

[27] W. Rudin. *Principles of Mathematical Analysis*, chapter 7. International series in pure and applied mathematics. McGraw-Hill Professional, New York, NY, 3 edition, Feb. 1976. ISBN 007054235X; 9780070542358.

[28] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan. 2015. ISSN 0893-6080. URL `http://dx.doi.org/10.1016/j.neunet.2014.09.003`.

[29] S. Sharma, S. Sharma, and A. Athaiya. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, 04(12):310–316, May 2020. ISSN 2455-2143. URL `http://dx.doi.org/10.33564/IJEAST.2020.v04i12.054`.

[30] M. Spivak. *Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus*, chapter 2. Addison-Wesley Publishing Company, Jan. 1995. ISBN 0805390219.

[31] A. Subasi. *Practical machine learning for data analysis using python*, chapter 3. Academic Press, San Diego, CA, June 2020. ISBN 9780128213797. URL `http://dx.doi.org/10.1016/C2019-0-03019-1`.

[32] M. Tenenbaum and H. Pollard. *Ordinary Differential Equations: An Elementary Textbook for Students of Mathematics, Engineering, and the Sciences*, chapter 1,2. Dover Books on Mathematics. Dover Publications, Mineola, NY, Oct. 1985. ISBN 0486649407.

[33] X. Wen and M. Zhou. Evolution and role of optimizers in training deep learning models. *IEEE/CAA Journal of Automatica Sinica*, 11 (10):2039–2042, 2024. ISSN 2329-9274. URL `http://dx.doi.org/10.1109/JAS.2024.124806`.

[34] D. G. Zill. *A First Course in Differential Equations with Modeling Applications*, chapter 2,8. Brooks/Cole, Cengage Learning, tenth edition, 2013. ISBN 1111827052.