CADI AYYAD UNIVERSITY
FACULTY OF SCIENCES AND TECHNIQUES GUELIZ

UNDERGRADUATE END OF STUDIES PROJECT

Solving a class of ODEs & PDEs using the deep learning library DEEPXDE

Author:
Tareq AIT HSAIN

Supervisor:
Pr. Noureddine ALAA

Jury members:
Pr. Noureddine ALAA
Pr. Abdeslem Hafid BENTBIB
Pr. Karim KREIT

A paper submitted in fulfillment of the requirements
for the degree of Licence Sciences et Techniques Mathématiques et Informatique Appliquées aux Sciences de l'Ingénieur (MIASI)

in the

Department of Mathematics

June 18, 2020

Abstract

Differential equations are an important aspect of mathematical models and problems, hence the ongoing need to find the appropriate tools to solve them. The subject matter of this project is DeepXDE, a deep learning library developed in Brown University that can solve different types of differential equations using neural networks. While the usage of our subject library is only a matter of defining of the problem using TensorFlow syntax, understanding its learning process would still require extensive knowledge of machine learning. Therefore we will give an overview for differential equations and their important role in the mathematical modeling of real life phenomenons, then we shall introduce key aspects of neural networks and deep learning, before engaging in the presentation of the library, its functionalities and its applications.

Acknowledgements

I would like to thank Professor Alaa for lending me his tremendous expertise in this field, for his concise and direct instructions that helped me better understand the scope of this project, and most importantly for the immense trust and patience he showed me throughout the development of this memoir.

I also thank the esteemed members of the jury Mr. Bentbib and Mr.Kreit for taking interest in this work, and taking the time to evaluate it. It would be an honor to have my work accepted by accomplished professors such as yourselves.

Contents

bismillah

Introduction

Deep learning is a very effective mathematical tool and set of techniques that achieved an outstanding performance on important problems in computer vision, natural language processing and speech recognition. However, its use in scientific computing has only just emerged in the form of neural networks (NNs) that can approximate functions and solutions.

On the other hand, many problems in science and engineering are reduced to a set of differential equations (DEs) through the process of mathematical modeling. Although model equations based on established physical laws may be constructed, analytical tools are frequently inadequate for the purpose of obtaining their closed form solution and usually numerical methods must be resorted to. These methods generally require some discretisation of the domain into a number of finite elements (FEs), which is not a straightforward task. In contrast to FE-type approximation, neural networks can be considered as approximation schemes where the input data for the design of a network consists of only a set of unstructured discrete data points. Thus an application of neural networks for solving DEs can be regarded as a mesh-free numerical method. It has been proved that radial basis function networks (RBFNs) with one hidden layer are capable of universal approximation (Park and Sandberg, 1993). Therefore it's convenient that we use different NN approaches to solve DEs.

Some of these approaches can only be applied to specific types of problems, such as image-like input domain or parabolic PDEs. Some researchers adopt the variational form of PDEs and minimize the corresponding energy functional. The problem is, not all PDEs can be derived from a known functional. Therefore, researchers resorted to physics informed neural networks (PINNs), a strong form in which automatic differentiation is used directly to avoid truncation errors and the numerical quadrature errors of variational forms.

DeepXDE, the Python library which we will be introducing in this paper, was developed using various PINN algorithms, so treating the subject matter will follow this structure:

- In Chapter 2, we give a brief overview for differential equations, the general notion of the existence of solutions and some numerical methods for solving them.
- In Chapter 3 we take on the theoretical and computational aspects of neural networks and deep learning.
- In Chapter 4 we introduce DeepXDE, explain the procedures and give some examples, and then in Chapter 5 we will elaborate on more diverse applications for DeepXDE.

On differential systems

Generalities

Definition. *A differential equation is an equation involving a function and its derivatives.*

Terminology

- A solution is any function (or formula for a function) that satisfies the equation.
- A general solution is a formula that describes all solutions to the equation. Typically, the general solution to a k-th order differential equation contains k arbitrary/undetermined constants.
- The order is the order of the highest order derivative of the unknown function explicitly appearing in the equation.
- Typically, a "differential equation problem" consists of a differential equation along with some auxiliary conditions the solution must also satisfy (e.i., initial values or boundary conditions for the solution). In practice you usually find the general solution first, and then choose values for the "undetermined constants" so that the auxiliary conditions are satisfied

Notations

The notation for differentiation depends on the author and the context.

- **Lagrange's notation** $(y', y', ..., y^{(n)})$ is used to compactly represent derivatives of any order .
- For differentiation and integration, we use **Leibniz's notation** $(\frac{dy}{dx}, \frac{d^2y}{dx^2}, ..., \frac{d^ny}{dx^n})$.
- There's also **Newton's notation** $(\dot{y}, \ddot{y}, \dddot{y})$ which is used in physics to represent derivatives of low order with respect to time.

2.1 Ordinary differential equations (ODEs)

A differential equation is called an ordinary differential equation (often shortened to "ODE") if only ordinary derivatives appear. That is, if the unknown function has only a single independent variable.

2.1.1 Background:

Ordinary differential equations appear in different fields of mathematics and social and natural sciences, because mathematical descriptions of change use differentials and derivatives. For mathematical modeling differentials, derivatives, and functions become related via equations, therefore rendering the differential equation to be a result that describes dynamically changing phenomena, evolution, and variation.

Examples.

- A simple example of an ODE is **the equation of intrinsic population growth** in the field of *population dynamics*; the study of an isolated population in an environment producing abundant food leads to the following model:

$$x'(t) = K.x(t),$$

i.e. population growth $x'(t)$ is, at any given time, proportional to the size of the population $x(t)$. The solutions to this equation, which would be in the form: $X(t) = e^{Kt} + c$ show an exponential growth phenomenon over time.

- A more complex system, made up of two species, prey and predator, leads to the Lotka-Volterra equations, which is a pair of first-order nonlinear differential equations:

$$\begin{cases} \dfrac{dx(t)}{dt} = x(t)(A - By(t)) \\ \dfrac{dy(t)}{dt} = y(t)(C - Dx(t)) \end{cases}$$

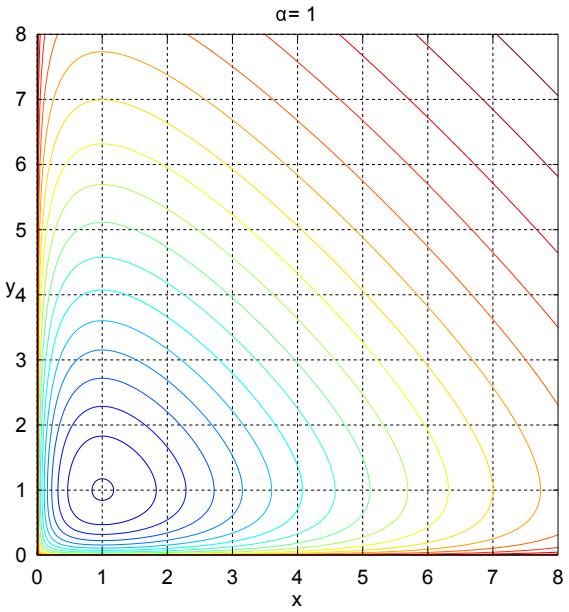

FIGURE 2.1: Population growth as descriped by the Lotka-Voletrra equations

where:

- x is the number of prey, y that of predators
- The derivatives $\dfrac{dx}{dt}$ and $\dfrac{dy}{dt}$ represent the variation of populations over time.
- A, B, C and D represent the interactions between the two populations

2.1.2 Definitions:

In what follows, let E be a normed vector space, I an interval of E, y a dependent variable and x an independent variable, and $y = f(x)$ is an unknown function of x. We will be using **Lagrange's notation**.

General definition. An ordinary differential equation (ODE) of order n is a functional relation of the form

$$F(x, y, y', ..., y^{(n-1)}, y^{(n)}) = 0$$

where F is a function of x, y, and derivatives of y ***and*** $F \subset C(U)$, U being an open subset of $E^{(n+1)} \times \mathbb{R}$

Remarks.

a) *$F(x, y, y', ..., y^{(n-1)}) = y^{(n)}$ is the **explicit** form of the differential equation in the above definition. DEs that can be put in an explicit form enjoy good theoretical properties, with - under certain assumptions - a theorem of existence and uniqueness of solutions: **the Cauchy-Lipschitz theorem** (which we will be revisiting later)*

b) *When the differential equation is in implicit form, we try, on the largest possible domains, to put the differential equation in its explicit form. Then we must proceed to the connection of the solutions obtained.*

c) *Let $m \geq 2$, and let's consider y a vector whose elements are functions; $y(x) = [y_1(x), y_2(x), ..., y_m(x)]$, and F a vector-valued function of y and its derivatives.*

Therefore $y^{(n)} = F(x, y, y', ..., y^{(n-1)})$ is equivalent to: $$\begin{pmatrix} y_1^{(n)} \\ y_2^{(n)} \\ \vdots \\ y_m^{(n)} \end{pmatrix} = \begin{pmatrix} f_1\left(x, \mathbf{y}, \mathbf{y}', \mathbf{y}'', \ldots, \mathbf{y}^{(n-1)}\right) \\ f_2\left(x, \mathbf{y}, \mathbf{y}', \mathbf{y}'', \ldots, \mathbf{y}^{(n-1)}\right) \\ \vdots \\ f_m\left(x, \mathbf{y}, \mathbf{y}', \mathbf{y}'', \ldots, \mathbf{y}^{(n-1)}\right) \end{pmatrix}$$

In this case, what we have on our hands is an ODE system.

Solutions. Following the general definition, a function y is called a solution for F if y is n-times differentiable on I and

$$\forall x \in I, F(x, y(x), y'(x), ...y^{(n)}(x)) = 0$$

A ***general solution*** of an nth-order equation is a solution containing n arbitrary independent constants of integration.
A ***particular solution*** is derived from the general solution by setting the constants to particular values, often chosen to fulfill set 'initial conditions or boundary conditions'.
A ***singular solution*** is a solution that cannot be obtained by assigning definite values to the arbitrary constants in the general solution.

2.1.3 Classification:

- **Linearity**: A differential equation is said to be linear if F can be written as a linear combination of the derivatives of y:

$$y^{(n)} = F(x, y, y', ..., y^{(n-1)}) = \sum_{i=0}^{n-1} a_i(x) y^{(i)} + g(x)$$

where $f_i(x)$ and $g_i(x)$ are continuous functions of x. $g_i(x)$ is called the source.

- It is called homogeneous if $g_i(x) \equiv 0$

Finally, note that we could admit complex values for the dependent variables. It will make no difference whether we use real or complex dependent variables.

Where do we go on now ?
Well, the study of differential equations is an ever developing field of mathematics. There are countless different but equally important aspects of ODEs, but since the subject matter of our paper is solving them, it's rather evident that our focus would be the solutions. Therefore, we are going to explore the conditions on the functions such that the differential system has a solution. We also study whether the solution is unique, subject to some additional initial conditions.

2.2 Existence and uniqueness theory

Our task now will be to prove the basic existence and uniqueness result for ordinary differential equations.

The study of existence and uniqueness of solutions became important due to the lack of general formula for solving nonlinear ODEs. Compact form of existence and uniqueness theory appeared nearly 200 years after the development of the theory of differential equation.

One of the most prominent facets of this study is the Cauchy problem, which is a problem in mathematics that asks for the solution of a differential equation that satisfies certain conditions that are given on a hypersurface in the domain. A Cauchy problem can be an initial value problem or a boundary value problem or it can be either of them. It is named after Augustin Louis Cauchy.

We will be introducing the notion of an initial value problem (IVP) for first order systems of ODE. Our main hypothesis in the studies of IVP is Hypothesis 1, which will be in force throughout this section.

Hypothesis 1. *Let $\Omega \subseteq \mathbb{R}^n$ be a domain and $I \subseteq \mathbb{R}$ be an open interval. Let $f : I \times \Omega \to \mathbb{R}^n$ be a continuous function defined by: $(x, y) \longmapsto f(x, y)$ where $y = (y_1, ...y_n)$.*
Let$(x_0, y_0) \subseteq I \times \Omega$ be an arbitrary point.

Definition 1. Assume **Hypothesis 1** on f.
An ***Initial Value Problem*** (also known as **Cauchy Problem**) for a first order system of n ordinary differential equations is given by:

$$y' = f(x, y) \tag{1.1}$$
$$y(x_0) = y_0 \tag{1.2}$$

Definition 2. An n-tuple of functions $y = (y_1, \ldots, y_n) \in C^1(I_0)$ where $I_0 \subset I$ is an open interval containing the point $x_0 \in I$ is called a solution of the IVP above if for every $x \in I_0$, the (n+1)-tuple $(x, y_1(x), y_2(x), \ldots, y_n(x)) \in I \times \Omega$,

$$y'(x) = f(x, y(x)) \quad \forall x \in I_0 \tag{1.3}$$
$$y(x_0) = y_0 \tag{1.4}$$

If I_0 is not an open interval, the left hand side of the equation (1.3) is interpreted as the appropriate one-sided limit at the end point(s).

Remarks 1. a) The IVP (1.1) is constituted of an interval I, a domain Ω, a continuous function f on $I \times \omega$, $x_0 \in I$, $y_0 \in \Omega$. Given I, $x_0 \in I$ andΩ. We can have many IVPs by changing the couple (f, y_0) belonging to the set $C(I \times \Omega) \times \Omega$.

b) Solutions which are defined on the entire interval I are called global solutions. Sometimes a solution is defined only on a subinterval of I. The solutions in this case are called local solutions to the IVP. See example for an IVP that has only a local solution.

c) In Definition 2, sometimes instead of the condition $y = (y_1, \ldots, y_n) \in C^1(I_0)$ it is required that the function u be differentiable. However, since the right hand side of (1.3) is continuous, both these conditions are equivalent.

Example of an IVP. An object falls under the influence of gravity near Earth's surface, where it can be assumed that the magnitude of the acceleration due to gravity is a constant g.

a) Construct a mathematical model for the motion of the object in the form of an initial value problem for a second order differential equation, assuming that the altitude and velocity of the object at time $t = 0$ are known. Assume that gravity is the only force acting on the object.

b) Solve the initial value problem derived in *a* to obtain the altitude as a function of time.

a) Let $y(t)$ be the altitude of the object at time t. Since the acceleration of the object has constant magnitude g and is in the downward (negative) direction, y satisfies the second order equation

$$y'' = -g,$$

where the prime now indicates differentiation with respect to t. If y_0 and v_0 denote the altitude and velocity when $t = 0$, then y is a solution of the initial value problem

$$y'' = -g, \quad y(0) = y_0, \quad y'(0) = v_0. \tag{1.5}$$

b) Integrating (1.5) twice yields

$$\begin{aligned} y' &= -gt + c_1, \\ y &= -\frac{gt^2}{2} + c_1 t + c_2. \end{aligned}$$

Imposing the initial conditions $y(0) = y_0$ and $y'(0) = v_0$ in these two equations shows that $c_1 = v_0$ and $c_2 = y_0$. Therefore the solution of the initial value problem (1.5) is

$$y = -\frac{gt^2}{2} + v_0 t + y_0.$$

An IVP is equivalent to an integral equation; this is a much needed property to prove the existence of solutions.

Lemma 1. Let y be a continuous function defined on an interval I_0 containing the point x_0 . The following statements are equivalent

a) The function y is a solution of the IVP

b) The function y satisfies the integral equation

$$y(x) = y_0 + \int_{x_0}^{x} f(s, y(s))ds \quad \forall x \in I_0$$

Proof..

- $a \Rightarrow b$: If y is a solution of IVP (1.1), then by definition of solution we have $y'(x) = f(x, y(x))$. Integrating this equation from x_0 to x gives the integral equation in b)
- $b \Rightarrow a$:
  Let y be a solution of our integral equation. Because the function $t \to y(t)$ is continuous, consequently the function $t \to f(t, y(t))$ is continuous on I_0 . Therefore the solution of our IE is a differentiable function on x (fundamental theorem of integration), and its derivative is defined as $x \to f(x, y(x))$ which is a continuous function.
  Therefore, because y is equal to a continuously differentiable function in the integral equation, y is also continuously differentiable.
  C/C: Differentiating the integral equation leads to the function y being a solution of our ODE, and the initial condition $y(x_0) = y_0$ is given by applying the integral equation at $x = x_0$.

Furthermore, to prove the existence and unicity theorem, we shall be using the notions of contraction mappings and Lipschitz continuity and more generally the Banach fixed point theorem.

Definition 3. Let (X, d) be a metric space, and suppose $F : X \to X$. We say that F is a contraction on X if there exists $0 < c < 1$ such that

$$\forall x, y \in Xd(F(x), F(y)) \leq cd(x, y)$$

(c is sometimes called the contraction constant).
A point $x_* \in X$ for which $F(x_*) = x_*$ is called a fixed point of F.

Banach fixed point theorem. *Let*(X, d) be a complete metric space and $F : X \to X$ be a contraction (with contraction constant 0 < c < 1). Then F has a unique fixed point $x_* \in X$. Moreover, for any $x_0 \in X$, if we generate the sequence x_k iteratively by functional iteration $x_{(k+1)} = F(x_k)$ for $k \geq 0$ (sometimes called fixed-point iteration), then $x_k \to x_*$.

Definition 4. A function f is said to be Lipschitz continuous on a rectangle R with respect to the variable y if there exists a K > 0 such that

$$\|f(x, y_1) - f(x, y_2)\| \leq K\|y_1 - y_2\| \ \forall (x, y_1), (x, y_2) \in R$$

Now that we have established these notions, we shall state and prove the local existence and uniqueness theorem (also known as Picard–Lindelöf theorem, Picard's existence theorem or Cauchy–Lipschitz theorem)

Existence and uniqueness theorem.
Consider the initial value problem

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0.$$

Suppose f is Lipschitz continuous in y (meaning the Lipschitz constant can be taken independent of t) and continuous in t, then for some value $\varepsilon > 0$, there exists a unique solution $y(t)$ to the initial value problem on the interval $[t_0 - \varepsilon, t_0 + \varepsilon]$

Proof.

Let: $C_{a,b} = I_a(t_0) \times \overline{B_b(y_0)}$
where:

$$I_a(t_0) = [t_0 - a, t_0 + a]$$
$$\overline{B_b(y_0)} = [y_0 - b, y_0 + b].$$

With $\overline{B_b(y_0)}$ being a closed ball in the space of continuous and bounded functions centered at the constant function y_0.
Let L be the Lipschitz constant of f with regard to the second variable.
We must prove that the function:

$$T : C(I_a(t_0), B_b(y_0)) \longrightarrow C(I_a(t_0), B_b(y_0)) \ , T[y(t)] = y_0 + \int_{t_0}^{t} f(s, y(s))\, ds$$

admits a fixed point. To do so, we shall prove that it satisfies the conditions of the Banach fixed point theorem.
We must show that the operator T maps a complete non-empty metric space X into itself and also is a contraction mapping.
In fact, T maps $\overline{B_b(y_0)}$ into itself in the space of continuous functions with uniform norm, as follows:

$$\begin{aligned}\|T[y(t)] - y_0\| =& \| \int_{t_0}^{t} f(s, y(s))\, ds\| \\ &\leq \int_{t_0}^{t'} \|f(s, y(s))\| ds \\ &\leq M|t' - t_0| \\ &\leq Ma \\ &\leq b\end{aligned}$$

where $M = \sup_{C_{a,b}} \|f\|$, and $t' \in [t_0 - a, t_0 + a]$ where the maximum is achieved. The last step is true with the condition that $a < \dfrac{b}{M}$.

Now we prove that T is a contraction.

Given two functions $y_1, y_2 \in C(I_a(t_0), B_b(y_0))$, in order to apply the Banach fixed point theorem we want:

$$\|T[y_1] - T[y_2]\|_\infty \leq k \|y_1 - y_2\|_\infty \text{ for some k < 1.}$$

So let t be such that: $\|T[y_1] - T[y_2]\|_\infty = \|(T[y_1] - T[y_2])(t)\|$
then using the definition of T :

$$\begin{aligned}\|(T[y_1] - T[y_2])(t)\| &= \left\| \int_{t_0}^{t} (f(s, y_1(s)) - f(s, y_2(s)))\, ds \right\| \\ &\leq \int_{t_0}^{t} \|f(s, y_1(s)) - f(s, y_2(s))\|\, ds \\ &\leq L \int_{t_0}^{t} \|y_1(s) - y_2(s)\|\, ds \qquad\qquad f \text{ is L-Lipschitz-continuous} \\ &\leq La \|y_1 - y_2\|_\infty\end{aligned}$$

This is a contraction if $a < \frac{1}{L}$.

We have established that T is a contraction on the Banach spaces with the metric induced by the uniform norm. This allows us to apply the Banach fixed point theorem to conclude that the operator has a unique fixed point. In particular, there is a unique function:

$$y \in C(I_a(t_0), B_b(y_0))$$

such that $T[y] = y$. This function is the unique solution of the IVP, valid on the interval I_a where a satisfies the condition $a < min\{\frac{b}{M}, \frac{1}{L}\}$.

Remarks.

a) The conditions of the existence and uniqueness theorem are sufficient but not necessary. For example, consider

$$y' = \sqrt{y} + 1, \quad y(0) = 0 \text{ with } x \in [0, 1]$$

Clearly f does not satisfy Lipschitz condition near origin. But still it has unique solution. ***To prove this, we use*** y_1 ***and*** y_2 ***as two solutions and consider*** $z(x) = y_1(x) - y_2(x)$ **.**

b) The existence and uniqueness theorem is also valid for certain system of first order equations. These theorems are also applicable to certain higher order ODE since a higher order ODEs can be reduced to a system of first order ODE.

We have now covered the local existence and uniqueness of a solution under certain conditions, we shall now move on to an equally important aspect: global existence. For that we shall need to introduce other notions.

In the definitionsYannick Viossat, 2020 that follow, we shall consider the IVP

$$\begin{cases} Y'(t) = f(t, Y(t)) \\ Y(0) = Y_0 \end{cases}$$

with $Y(t) \in \mathbb{R}^d$ and $f : \Omega \to \mathbb{R}$ where Ω is an open set of $\mathbb{R} \times \mathbb{R}^d$.

Let J also be an interval such that $Y : J \to \mathbb{R}^d$ is derivable along J and $\forall t \in J$, $(t, Y(t)) \in \Omega$ (*).

Much like the previous definitions, we shall establish that $\Omega = I \times \mathbb{R}^d$ with I being an interval, which therefore reduces the condition in (*) to $J \subset I$.

Corollary 2. If $Y : J \to \mathbb{R}$ is a solution to (1), then any restriction of Y on an interval $\tilde{J} \subset J$ is still a solution.

To achieve unicity, we should be restricted on non-prolonged solutions. These are what we have come to label as **maximal solutions**

Definition. (Maximal solution). A solution $Y : J \to \mathbb{R}$ of (1) is ***maximal*** if **there doesn't exist** an interval $\tilde{J}$ strictly containing J and for which $\tilde{Y} : \tilde{J} \to \mathbb{R}$ is a solution such that $\tilde{Y}_{|J} = Y$.
i.e it is maximal if it cannot be extended to a larger time-interval.

Remarks.. The maximal solution y_{max} of an IVP which is locally unique is the solution such that :

- it has an interval as domain which contains x_0
- it is not a restriction of any other solution whose domain is a larger interval.
- Every IVP has a unique maximal solution

$Dom(y_{max})$ is called the maximal domain of solution and is denoted $]x_-, x_+[$

Theorem (Explosion alternative - Blow-up in finite time). *Let I be an open interval and $f : I \times \mathbb{R} \to \mathbb{R}^d$ a class C^1 function. Let α and β be in $\overline{\mathbb{R}}$ and $Y :]\alpha, \beta[\to \mathbb{R}$ a maximal solution of $Y'(t) = f(t, Y(t))$.*

a) If $\beta < supI$ then $\|Y(t)\| \to +\infty$ when $t \to \beta$.

b) If $\alpha > infI$ then $\|Y(t)\| \to +\infty$ when $t \to \alpha$.

($infI, supI \in \overline{\mathbb{R}}$).

Corollary 3 (Non-explosion criterion). *Let $\Omega = I \times \mathbb{R}^d$. Let Y be a maximal solution of IVP 1.1. Let $g : I \to \mathbb{R}$ be continuous. Let $t_0 \in [\alpha, \beta]$.*
1) If $\|Y(t)\| \leq g(t)$ on $[t_0, \beta[$, then $\beta = supI$.
2) If $\|Y(t)\| \leq g(t)$ on $]\alpha, t_0]$, then $\alpha = infI$.
3) If $\|Y(t)\| \leq g(t)$ on $[\alpha, \beta]$, then Y is a global solution.

Invariant set for a System of Differential Equation

Consider the following system:

$$(CP_f) \begin{cases} \dfrac{du(t)}{dt} = f(u(t)) \quad \forall t \in [0, T] \\ u(0) = u_0 \end{cases}$$

where $f : \mathbb{R}^d \longrightarrow \mathbb{R}^d$ locally Lipscitzian and $u_0 \in \mathbb{R}^d$.

Definition: We say that the convex set $\mathcal{C}$ is invariant for the Cauchy problem $\left(CP_f\right)$ if for each $u_0 \in \mathcal{C}$ the solution u of $\left(CP_f\right)$ satisfies u(t) $\in \mathcal{C}$ for all t $\in [0, T(u_0)[$.
$T(u_0)$ = maximum time of existence of the solution with initial data u_0.

Proposition
The convex set $\mathcal{C}$ is invariant for the Cauchy problem (CP_f) if :

$$\forall u \in \partial\mathcal{C}, p \in n(u) \quad < p, f(u) > \leq 0$$

where $n(u)$ is normal cone at $\partial\mathcal{C}$ in u.

Then we can prove the following result:

Theorem 3.
Let $f : \mathbb{R}^d \longrightarrow \mathbb{R}^d$ locally Lipschitz. Let $\mathcal{C}$ be a convex and closed set of $\mathbb{R}^d$. Then we have the following equivalences:
1) $\forall u \in \partial\mathcal{C} \quad \forall p \in n(u) \quad < p, f(u) > \leq 0$
2) $\forall u_0 \in \mathcal{C} \quad \forall T > 0$ there exists a solution u of

$$\begin{cases} u \in C^1([0,T], \mathbb{R}^d) \\ \dfrac{du}{dt}(t) = f(u(t)) \quad \forall t \in [0,T] \\ u(0) = u_0 \\ u(t) \in \mathcal{C} \quad t \in [0,T]. \end{cases} \tag{1.6}$$

Example:
Consider the following system $(d = 2)$:

$$\begin{cases} \dfrac{du}{dt}(t) = v^2 - u^2 \\ \dfrac{dv}{dt}(t) = u^2 - v^2 \\ u(0) = u_0, v(0) = v_0 \end{cases} \tag{1.7}$$

Let $\mathcal{C} = [0, M] \times [0, M]$ where $M > 0$. We can show that $\mathcal{C}$ is closed convex of $\mathbb{R}^2$ and that $f(u, v) = (v^2 - u^2, u^2 - v^2)$ is locally Lipschitz.
We can also verify that condition (1) of the last theorem is satisfied: on each border of the square, we can easily verify that for every u $\in \partial\mathcal{C}$ and for every $p = n(u)$ with n begin the normal of u at the borders, we always get $< p, f(u) > \leq 0$.
Consequently, by application of the 2nd property in Theorem 3 and Corollary 3 we have $T = +\infty$ therefore we have global existence

2.3 Periodic solutions to ODEs

Periodic solutions of differential equations are functions that describe regularly repeating processes. In such branches of science as the theory of oscillations and celestial mechanics, periodic solutions of systems of differential equations are of special interest.

It is a solution that periodically depends on the independent variable t. For a periodic solution $y(t)$ (in the case of a system, y is a vector), there is a number $T \neq 0$ such that

$$x(t + T) = x(t) \quad \forall t \in \mathbb{R}$$

All possible such T are called periods of this periodic solution; the continuity of y implies that either $y(t)$ is independent of t or that all possible periods are integral multiples of one of them — the minimal period $T_0 > 0$. When one speaks of a periodic solution, it is often understood that the second case applies, and T_0 is simply termed the period.

A periodic solution is usually considered for a system of ordinary differential equations where the right-hand sides either are independent of t (an autonomous system):

$$\dot{x} = f(x), \quad x \in U, \quad (1)$$

where is U a region in $\mathbb{R}^n$,
or else periodically depend on :

$$\dot{x} = f(x,t), \quad f(t+T_1,x) = f(t,x), \quad x \in U, \quad (2)$$

(In a system with a different type of dependence on t for the right-hand sides there is usually no periodic solution.) In case (2) the period T_0 of the periodic solution usually coincides with the period T_1 of the right-hand side or is an integer multiple of T_1; other T_0 are possible only in exceptional cases.

Periodic solutions with periods $T_0 = kT_1$, $k > 1$, describe subharmonic oscillations (Forced oscillations) and therefore are themselves sometimes called subharmonic periodic solutions (or subharmonics).

Example

In scaled form the equations are

$$\dot{x} = y, \dot{y} = -x(*)$$

It is easy to solve the IVP for this system analytically. If we let $x(0) = a$, and $\dot{x}(0) = b$, we get the solution

$$x(t) = acos(t) + bsin(t), y(t) = -asin(t) + bcos(t).$$

Thus all solutions are periodic with the same period, namely 2π. From the above solution, we find by direct calculation that

$$[x(t)]^2 + [y(t)]^2 = a^2 + b^2$$

so that all of the orbits are circles.

Existence of periodic solution for an ordinary differential equation

Consider the following equation:

$$(PS_f) \begin{cases} \dfrac{du}{dt}(t) = f(t,u(t)) \quad \forall t \in [0,T] \\ u(0) = u(T) \end{cases}$$

where $T > 0$ is a period, $f : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$ locally Lipscitzian and .

Then we can prove the following result:

Theorem

Let $f : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$ locally Lipscitzian. We assume that there is $T > 0$ such that:

1) $\forall r \in \mathbb{R} \quad t \to f(t,r)$ is peridic with period T

2) $\forall t \in [0,T] \quad r \to f(t,r)$ is monotone

a necessary and sufficient condition for (PS_f) to admit a periodic solution is that there is a continuous function $y : [0,T] \to \mathbb{R}$ such that $\int_0^T f(t,y(t)) = 0$.

Example:

Consider the following equation:

$$\begin{cases} \dfrac{du}{dt}(t) = -u^3 + sin(t) \quad 0 < t < 2\pi \\ u(0) = u(2 \cdot \pi) \end{cases}$$

So this equation admits a $2\pi - periodic$ solution.

Indeed, here $f(t,r) = -r^3 + sin(t)$ is locally Lipschitzian, $t \longrightarrow f(t,r)$ is $2\pi - periodic$ and $t \longrightarrow f(t,r)$ is is decreasing. In addition there exists $y(t) = sin(t)$ such that $\int_0^{2\cdot\pi}(sin(t) - sin(t)^3)dt = \int_0^{2\cdot\pi} sin(t)cos(t)^2 = 0$ (This is the criterion of existence of a periodic solution for an ODE)

2.4 Boundary value problem

A boundary problem is a differential equation with a set of additional constraints for which the solution takes values imposed at the limits (i.e boundaries) of the resolution range.

As opposed to the analogous initial value problem, where one or several conditions at the same place are imposed (typically the value of the solution and its successive derivatives at a point), to which the Cauchy-Lipschitz theorem brings a general answer, boundary problems are often difficult problems, and whose solution may each time lead to different interpretations.

A boundary value corresponds to a minimum or maximum input or output value specified for a system.

Examples.

1. If the independent variable is time over the domain [0,1], a boundary value problem would specify values for $y(t)$ at both $t = 0$ and $t = 1$, whereas an initial value problem would specify a value of $y(t)$ and $y'(t)$ at time $t = 0$.

2. A more concrete example would be the 2^{nd} order PDE

$$\forall x \in [0, \pi/2],\ y''(x) + y(x) = 0$$

For which we do not have initial conditions, but rather boundary conditions on the extremities of the definition interval:

$$y(0) = 0,\ y(\pi/2) = 2.$$

Solving this problem should go as follows:
Let A, B such that .

$$y(x) = A\sin(x) + B\cos(x).$$

Using the boundary conditions we get

$$y(0) = 0 = A \cdot 0 + B \cdot 1,\ y(\pi/2) = 2 = A \cdot 1 + B \cdot 0$$

We get A = 2, B =0, therefore the solution is now well defined as:

$$y(x) = 2\sin(x).$$

Types of boundary conditions

- ***Dirichlet boundary condition (named after Johann Dirichlet)*** .
  which specifies the values that **a solution** needs to take along the boundary of the domain.
  - Consider the ODE: $y'' - y = 0$
    The Dirichlet boundary conditions on the interval [a,b] take the form

    $$y(a) = c_1, \quad y(b) = c_2$$

    where c_1 and c_2 are given numbers.
  - If E is the domain on which the given equation is to be solved and ∂E denotes its boundary, consider the PDE: $\nabla^2 y + y = 0$,
    The Dirichlet boundary conditions on a domain $E \subset \mathbb{R}^n$ take the form

    $$y(x) = f(x) \quad \forall x \in \partial E,$$

    where f is a known function defined on the boundary ∂E .
- ***Neumann boundary conditions***
  which specifies the conditions for which ***the derivative*** of a solution is applied within the boundary of the domain.

– Consider the same ODE : $y'' - y = 0$
The Neumann boundary conditions on the interval [a,b] take the form

$$y'(a) = c_1, \quad y'(b) = c_2$$

where c_1 and c_2 are given numbers.

– Consider the same PDE: $\nabla^2 y + y = 0$,
The Neumann boundary conditions on a domain $E \subset \mathbb{R}^n$ take the form

$$\frac{\partial y}{\partial \overrightarrow{n}}(x) = f(x) \quad \forall x \in \partial\Omega$$

where f is a known scalar function defined on the boundary ∂E and $\overrightarrow{n}$ is the **normal** to the boundary ∂E . The normal derivative on the left hand side of the solution is given by:

$$\frac{\partial y}{\partial \overrightarrow{n}}(x) = \overrightarrow{\text{grad}}\, y(x) \cdot \overrightarrow{n}(x)$$

It's evident that the boundary must be well defined so that the normal derivative exists.

- ***Mixed boundary condition*** .
which is a combination of Dirichlet BCs and Neumanns BCs. For example

– considering the PDE: $\nabla^2 y + y = 0$, on a an interval [0,L]
With the conditions that:

$$y(0) = 0$$

$$y'(L) = 1 \quad \forall x \in \partial E,$$

- ***Robin boundary condition*** .
which is a **weighted combination** of Dirichlet BCs and Neumanns BCs.

– Consider the equation
$fu + g\frac{\partial u}{\partial n} = \varphi$ on ∂E with f, g and φ know functions defined on ∂O.
For exemple, E = [0, a], the Robin BC would be as follows:

$$fu(0) - gu'(0) = \varphi(0)$$
$$fu(a) + gu'(a) = \varphi(a)$$

- ***Periodic boundary condition*** .
One example of periodic boundary conditions can be defined according to smooth real functions $\phi : \mathbb{R}^n \to \mathbb{R} by$, $\frac{\partial^m}{\partial x_1^m}\phi(a_1, x_2, ..., x_n) = \frac{\partial^m}{\partial x_1^m}\phi(b_1, x_2, ..., x_n)$, $\frac{\partial^m}{\partial x_2^m}\phi(x_1, a_2, ..., x_n) = \frac{\partial^m}{\partial x_2^m}\phi(x_1, b_2, ..., x_n), \ldots, \frac{\partial^m}{\partial x_n^m}\phi(x_1, x_2, ..., a_n) = \frac{\partial^m}{\partial x_n^m}\phi(x_1, x_2, ..., b_n)$
for all m = 0, 1, 2, ... and for constantsa_i and b_i.

2.5 Linear differential systems

Many physical situations are modeled by systems of n differential equations in n unknown functions,Trench, 2013 where $n \geq 2$.

Example

Let $\mathbf{X} = \mathbf{X}(t) = x(t)\,\mathbf{i} + y(t)\,\mathbf{j} + z(t)\,\mathbf{k}$ be the position vector at time t of an object with mass m, relative to a rectangular coordinate system with origin at Earth's center. According to Newton's law of gravitation, Earth's gravitational force $\mathbf{F} = \mathbf{F}(x, y, z)$ on the object is inversely proportional to the square of the distance of the object from Earth's center, and directed toward the center; thus,

$$\mathbf{F} = \frac{K}{\|\mathbf{X}\|^2}\left(-\frac{\mathbf{X}}{\|\mathbf{X}\|}\right) = -K\frac{x\,\mathbf{i} + y\,\mathbf{j} + z\,\mathbf{k}}{(x^2 + y^2 + z^2)^{3/2}}, \tag{1.8}$$

where K is a constant. To determine K, we observe that the magnitude of $\mathbf{F}$ is

$$\|\mathbf{F}\| = K\frac{\|\mathbf{X}\|}{\|\mathbf{X}\|^3} = \frac{K}{\|\mathbf{X}\|^2} = \frac{K}{(x^2 + y^2 + z^2)}.$$

Let R be Earth's radius. Since $\|\mathbf{F}\| = mg$ when the object is at Earth's surface,

$$mg = \frac{K}{R^2}, \quad \text{so} \quad K = mgR^2.$$

Therefore we can rewrite (1.8) as

$$\mathbf{F} = -mgR^2\frac{x\,\mathbf{i} + y\,\mathbf{j} + z\,\mathbf{k}}{(x^2 + y^2 + z^2)^{3/2}}.$$

Now suppose $\mathbf{F}$ is the only force acting on the object. According to Newton's second law of motion, $\mathbf{F} = m\mathbf{X}''$; that is,

$$m(x''\,\mathbf{i} + y''\,\mathbf{j} + z''\,\mathbf{k}) = -mgR^2\frac{x\,\mathbf{i} + y\,\mathbf{j} + z\,\mathbf{k}}{(x^2 + y^2 + z^2)^{3/2}}.$$

Cancelling the common factor m and equating components on the two sides of this equation yields the system

$$\begin{aligned} x'' &= -\frac{gR^2x}{(x^2 + y^2 + z^2)^{3/2}} \\ y'' &= -\frac{gR^2y}{(x^2 + y^2 + z^2)^{3/2}} \\ z'' &= -\frac{gR^2z}{(x^2 + y^2 + z^2)^{3/2}}. \end{aligned} \tag{1.9}$$

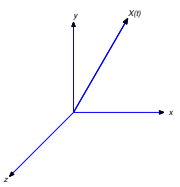

FIGURE 2.2: (x,y,z) coordinate system

Reduction of higher order systems to first order systems

A system of the form

$$\begin{array}{rcl} y_1' &=& g_1(t, y_1, y_2, \ldots, y_n) \\ y_2' &=& g_2(t, y_1, y_2, \ldots, y_n) \\ &\vdots& \\ y_n' &=& g_n(t, y_1, y_2, \ldots, y_n) \end{array} \tag{1.10}$$

is called a *first order system*, since the only derivatives occurring in it are first derivatives. The derivative of each of the unknowns may depend upon the independent variable and all the unknowns, but not on the derivatives of other unknowns. When we wish to emphasize the number of unknown functions in (1.10) we will say that (1.10) is an $n \times n$ system.

Systems involving higher order derivatives can often be reformulated as first order systems by introducing additional unknowns.

Example For example, let's rewrite the system

$$\begin{array}{rcl} x'' &=& f(t, x, x', y, y', y'') \\ y''' &=& g(t, x, x', y, y'y'') \end{array}$$

as a first order system.

Solution We regard x, x', y, y', and y'' as unknown functions, and rename them

$$x = x_1,\ x' = x_2, \quad y = y_1, \quad y' = y_2, \quad y'' = y_3.$$

These unknowns satisfy the system

$$\begin{array}{rcl} x_1' &=& x_2 \\ x_2' &=& f(t, x_1, x_2, y_1, y_2, y_3) \\ y_1' &=& y_2 \\ y_2' &=& y_3 \\ y_3' &=& g(t, x_1, x_2, y_1, y_2, y_3). \end{array}$$

Example Another example is rewriting the equation

$$y^{(4)} + 4y''' + 6y'' + 4y' + y = 0 \tag{1.11}$$

as a 4×4 first order system.

Solution We regard y, y', y'', and y''' as unknowns and rename them

$$y = y_1, \quad y' = y_2, \quad y'' = y_3, \quad \text{and} \quad y''' = y_4.$$

Then $y^{(4)} = y_4'$, so (1.11) can be written as

$$y_4' + 4y_4 + 6y_3 + 4y_2 + y_1 = 0.$$

Therefore $\{y_1, y_2, y_3, y_4\}$ satisfies the system

$$\begin{array}{rcl} y_1' &=& y_2 \\ y_2' &=& y_3 \\ y_3' &=& y_4 \\ y_4' &=& -4y_4 - 6y_3 - 4y_2 - y_1. \end{array}$$

Example We can also transform

$$x''' = f(t, x, x', x'')$$

into a system of first order equations by the same way

Solution We regard x, x', and x'' as unknowns and rename them

$$x = y_1, \quad x' = y_2, \quad \text{and} \quad x'' = y_3.$$

Then

$$y_1' = x' = y_2, \quad y_2' = x'' = y_3, \quad \text{and} \quad y_3' = x'''.$$

Therefore $\{y_1, y_2, y_3\}$ satisfies the first order system

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= f(t, y_1, y_2, y_3). \end{aligned}$$

2.6 Solving differential equations

Solving differential equations by quadrature (i.e. using elementary operations and primitivation) is only possible in a very limited number of cases. For example, even the second order scalar linear differential equations do not allow such a general solution formula. Even when a solution formula is available, it can involve integrals that can be calculated only by using a numerical quadrature formula.

In either situation, numerical methods provide a powerful alternative tool for solving the differential equation.Trench, 2013

Let us consider IVP 1 as follows

$$y' = f(t, y) \tag{1.12}$$
$$y(t_0) = y_0 \tag{1.13}$$

Numerical methods for solving IVP 1 should find an approximate solution $y(t)$ on an interval [a,b] at a discrete set of evenly spaced nodes, such that:

$$t_n = t_0 + nh \qquad n = 0, 1, ..., N$$

.

with

$$a \leq t_0 < t_1 < t_2 < \cdots < t_n \leq b$$

The approximate solution would be denoted as follows:

$$y(t_n) = y_h(t_n) = y_n, \qquad n = 0, 1, ..., N.$$

If an initial value problem

$$y' = f(x, y), \quad y(x_0) = y_0 \tag{1.14}$$

can't be solved analytically, we resort to numerical methods to obtain approximations to a solution of (1.14). We will treat these methods in this section.

We are interested in calculating approximate values of the solution of (1.14) at equally spaced points $x_0, x_1, \ldots, x_n = b$ in an interval $[x_0, b]$. Thus,

$$x_i = x_0 + ih, \quad i = 0, 1, \ldots, n,$$

where

$$h = \frac{b - x_0}{n}.$$

The approximate values of the solution at these points are denoted by $y_0, y_1, \ldots, y_n$; therefore, y_i is an approximation to $y(x_i)$.

$$e_i = y(x_i) - y_i$$

is called the *error at the ith step.* Because of the initial condition $y(x_0) = y_0$, we always have $e_0 = 0$. However, in general $e_i \neq 0$ if $i > 0$.

We generally encounter two sources of error in applying a numerical method to solve an initial value problem:

- The formulas defining the method are based on some sort of approximation. Errors that result from the inaccuracy of the approximation are called *truncation errors*.
- Computers do operations (arithmetic) with a fixed number of digits, so therefore they make errors in evaluating the formulas defining the numerical methods. Errors due to the computer's inability to do exact arithmetic are called *roundoff errors.*

Since a careful analysis of roundoff error is beyond the scope and interest of our project, we will consider only truncation errors.

2.6.1 Euler's method

The simplest numerical method for solving an initial value problem is called Euler's method, which is not an efficient numerical method, but many of the ideas around the numerical solutions of differential equations are introduced with it.
The Euler method is rarely used in practice, but its simplicity makes it useful for illustrative purposes.

It is method is based on the assumption that the tangent line to the integral curve of (1.14) at $(x_i, y(x_i))$ approximates the integral curve over the interval $[x_i, x_{i+1}]$. Since the slope of the integral curve of (1.14) at $(x_i, y(x_i))$ is $y'(x_i) = f(x_i, y(x_i))$, the equation of the tangent line to the integral curve at $(x_i, y(x_i))$ is

$$y = y(x_i) + f(x_i, y(x_i))(x - x_i). \tag{1.15}$$

Setting $x = x_{i+1} = x_i + h$ in (1.15) yields

$$y_{i+1} = y(x_i) + hf(x_i, y(x_i)) \tag{1.16}$$

as an approximation to $y(x_{i+1})$. Since $y(x_0) = y_0$ is known, we can use (1.16) with $i = 0$ to compute

$$y_1 = y_0 + hf(x_0, y_0).$$

However, setting $i = 1$ in (1.16) yields

$$y_2 = y(x_1) + hf(x_1, y(x_1)),$$

which isn't useful, since we *don't know* $y(x_1)$. Therefore we replace $y(x_1)$ by its approximate value y_1 and redefine

$$y_2 = y_1 + hf(x_1, y_1).$$

Having computed y_2, we can compute

$$y_3 = y_2 + hf(x_2, y_2).$$

In general, Euler's method starts with the known value $y(x_0) = y_0$ and computes $y_1, y_2, \ldots, y_n$ successively by with the formula

$$y_{i+1} = y_i + hf(x_i, y_i), \quad 0 \leq i \leq n - 1. \tag{1.17}$$

The next example illustrates the computational procedure indicated in Euler's method.

Example:3.1.1 Use Euler's method with $h = 0.1$ to find approximate values for the solution of the initial value problem

$$y' + 2y = x^3 e^{-2x}, \quad y(0) = 1 \tag{1.18}$$

at $x = 0.1, 0.2, 0.3$.

Solution We rewrite (1.18) as

$$y' = -2y + x^3 e^{-2x}, \quad y(0) = 1,$$

which is of the form (1.14), with

$$f(x, y) = -2y + x^3 e^{-2x}, \; x_0 = 0, \text{ and } y_0 = 1.$$

Euler's method yields

$$\begin{aligned} y_1 &= y_0 + hf(x_0, y_0) \\ &= 1 + (.1)f(0, 1) = 1 + (.1)(-2) = .8, \end{aligned}$$

$$\begin{aligned} y_2 &= y_1 + hf(x_1, y_1) \\ &= .8 + (.1)f(.1, .8) = .8 + (.1)\left(-2(.8) + (.1)^3 e^{-.2}\right) = .640081873, \end{aligned}$$

$$\begin{aligned} y_3 &= y_2 + hf(x_2, y_2) \\ &= .640081873 + (.1)\left(-2(.640081873) + (.2)^3 e^{-.4}\right) = .512601754. \end{aligned}$$

Examples Illustrating The Error in Euler's Method

Example:3.1.2 Use Euler's method with step sizes $h = 0.1$, $h = 0.05$, and $h = 0.025$ to find approximate values of the solution of the initial value problem

$$y' + 2y = x^3 e^{-2x}, \quad y(0) = 1$$

at $x = 0$, 0.1, 0.2, 0.3, ..., 1.0. Compare these approximate values with the values of the exact solution

$$y = \frac{e^{-2x}}{4}(x^4 + 4), \tag{1.19}$$

which can be obtained by the method of Section 2.1. (Verify.)

Solution Table 2.1 shows the values of the exact solution (1.19) at the specified points, and the approximate values of the solution at these points obtained by Euler's method with step sizes $h = 0.1$, $h = 0.05$, and $h = 0.025$. In examining this table, keep in mind that the approximate values in the column corresponding to $h = .05$ are actually the results of 20 steps with Euler's method. We haven't listed the estimates of the solution obtained for $x = 0.05$, 0.15, ..., since there's nothing to compare them with in the column corresponding to $h = 0.1$. Similarly, the approximate values in the column corresponding to $h = 0.025$ are actually the results of 40 steps with Euler's method.

Table 2.1. Numerical solution of $y' + 2y = x^3 e^{-2x}$, $y(0) = 1$, by Euler's method.

x	$h = 0.1$	$h = 0.05$	$h = 0.025$	Exact
0.0	1.000000000	1.000000000	1.000000000	1.000000000
0.1	0.800000000	0.810005655	0.814518349	0.818751221
0.2	0.640081873	0.656266437	0.663635953	0.670588174
0.3	0.512601754	0.532290981	0.541339495	0.549922980
0.4	0.411563195	0.432887056	0.442774766	0.452204669
0.5	0.332126261	0.353785015	0.363915597	0.373627557
0.6	0.270299502	0.291404256	0.301359885	0.310952904
0.7	0.222745397	0.242707257	0.252202935	0.261398947
0.8	0.186654593	0.205105754	0.213956311	0.222570721
0.9	0.159660776	0.176396883	0.184492463	0.192412038
1.0	0.139778910	0.154715925	0.162003293	0.169169104

Python implementation

We will program Euler's method so it can solve the IVP in Example 3.1.2:

$$\begin{cases} y' + 2y = x^3 e^{-2x}, \\ y(0) = 1 \end{cases}$$

Which has an exact solution: $y(x) = \frac{1}{4}e^{-2x}(x^4 + 4)$

We define a function Euler that takes as arguments the interval, the initial value and the number of steps (we try 100). Note that we use the variables `vx` and `vy` so we can plot the solutions, both predicted and exact.

```
from math import exp
import matplotlib.pyplot as plt


def euler(f, a, y0, b, n):
        t, y = a, y0
        vt = [0] * (n + 1)
        vy = [0] * (n + 1)
        h = (b - a) / float(n)
        vt[0] = t = a
        vy[0] = y = y0
        for i in range(1, n + 1):
                t += h
                y += h * f(t, y)
                vt[i] = t
                vy[i] = y
        return vt, vy

vt, vye = euler(f, 0, 1, 10, 100)

#y'(t)=f(t,y)

def f(t, y):
        return -2*y + (t**3)*exp(-2*t)

#exact solution, we will use it to compare
def exact(n):
        vx = [0] * (n + 1)
        vy = [0] * (n + 1)
        vx[0] = 0
        vy[0] = 1
        for i in range(1, n+1):
                vx[i] = i
                vy[i] = 0.25*np.exp(-2*i)*((i**4)+4)
        return vx, vy

plt.plot(vt, vye, exx, exy)
```

The resulting plot shows us that although the Euler method does give us a good general notion of the solution, it's still very far from the desired output.

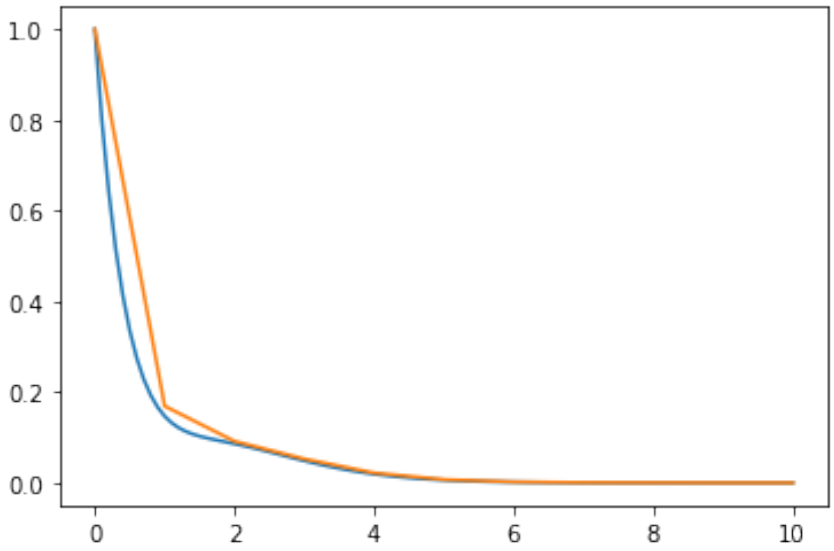

FIGURE 2.3: Euler method solution (yellow) compared to exact solution (blue)

2.6.2 Runge-Kutta method (RK4)

In general, if k is any positive integer and f satisfies appropriate assumptions, there are numerical methods for solving an initial value problem

$$y' = f(x,y), \quad y(x_0) = y_0. \tag{1.20}$$

We present a numerical methods where $k = 1$ and $k = 2$. We'll skip methods for which $k = 3$ and proceed to the Runge-Kutta method, the most widely used method, for which $k = 4$. The magnitude of the local truncation error is determined by the fifth derivative $y^{(5)}$ of the solution of the initial value problem. Therefore the local truncation error will be larger where $|y^{(5)}|$ is large, or smaller where $|y^{(5)}|$ is small. The Runge-Kutta method computes approximate values $y_1, y_2, \ldots, y_n$ of the solution of (1.20) at $x_0, x_0 + h, \ldots, x_0 + nh$ as follows: Given y_i, compute

$$\begin{aligned} k_{1i} &= f(x_i, y_i), \\ k_{2i} &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_{1i}\right), \\ k_{3i} &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_{2i}\right), \\ k_{4i} &= f(x_i + h, y_i + hk_{3i}), \end{aligned}$$

and

$$y_{i+1} = y_i + \frac{h}{6}(k_{1i} + 2k_{2i} + 2k_{3i} + k_{4i}).$$

Example:3.3.1 Use the Runge-Kutta method with $h = 0.1$ to find approximate values for the solution of the initial value problem

$$y' + 2y = x^3 e^{-2x}, \quad y(0) = 1, \tag{1.21}$$

at $x = 0.1, 0.2$.

Solution Again we rewrite (1.21) as

$$y' = -2y + x^3 e^{-2x}, \quad y(0) = 1,$$

which is of the form (1.20), with

$$f(x,y) = -2y + x^3 e^{-2x}, \; x_0 = 0, \text{ and } y_0 = 1.$$

The Runge-Kutta method yields

$$\begin{aligned}
k_{10} &= f(x_0, y_0) = f(0,1) = -2,\\
k_{20} &= f(x_0 + h/2, y_0 + hk_{10}/2) = f(.05, 1 + (.05)(-2))\\
&= f(.05, .9) = -2(.9) + (.05)^3 e^{-.1} = -1.799886895,\\
k_{30} &= f(x_0 + h/2, y_0 + hk_{20}/2) = f(.05, 1 + (.05)(-1.799886895))\\
&= f(.05, .910005655) = -2(.910005655) + (.05)^3 e^{-.1} = -1.819898206,\\
k_{40} &= f(x_0 + h, y_0 + hk_{30}) = f(.1, 1 + (.1)(-1.819898206))\\
&= f(.1, .818010179) = -2(.818010179) + (.1)^3 e^{-.2} = -1.635201628,\\
y_1 &= y_0 + \frac{h}{6}(k_{10} + 2k_{20} + 2k_{30} + k_{40}),\\
&= 1 + \frac{.1}{6}(-2 + 2(-1.799886895) + 2(-1.819898206) - 1.635201628) = .818753803,
\end{aligned}$$

$$\begin{aligned}
k_{11} &= f(x_1, y_1) = f(.1, .818753803) = -2(.818753803)) + (.1)^3 e^{-.2} = -1.636688875,\\
k_{21} &= f(x_1 + h/2, y_1 + hk_{11}/2) = f(.15, .818753803 + (.05)(-1.636688875))\\
&= f(.15, .736919359) = -2(.736919359) + (.15)^3 e^{-.3} = -1.471338457,\\
k_{31} &= f(x_1 + h/2, y_1 + hk_{21}/2) = f(.15, .818753803 + (.05)(-1.471338457))\\
&= f(.15, .745186880) = -2(.745186880) + (.15)^3 e^{-.3} = -1.487873498,\\
k_{41} &= f(x_1 + h, y_1 + hk_{31}) = f(.2, .818753803 + (.1)(-1.487873498))\\
&= f(.2, .669966453) = -2(.669966453) + (.2)^3 e^{-.4} = -1.334570346,\\
y_2 &= y_1 + \frac{h}{6}(k_{11} + 2k_{21} + 2k_{31} + k_{41}),\\
&= .818753803 + \frac{.1}{6}(-1.636688875 + 2(-1.471338457) + 2(-1.487873498) - 1.334570346)\\
&= .670592417.
\end{aligned}$$

Table 2.2. Numerical solution of $y' + 2y = x^3 e^{-2x}$, $y(0) = 1$, by the Runge-Kuttta method and the Euler method.

x	$h = 0.05$	$h = 0.025$	$h = 0.1$	$h = 0.05$	Exact
0.0	1.000000000	1.000000000	1.000000000	1.000000000	1.000000000
0.1	0.810005655	0.814518349	0.818753803	0.818751370	0.818751221
0.2	0.656266437	0.663635953	0.670592417	0.670588418	0.670588174
0.3	0.532290981	0.541339495	0.549928221	0.549923281	0.549922980
0.4	0.432887056	0.442774766	0.452210430	0.452205001	0.452204669
0.5	0.353785015	0.363915597	0.373633492	0.373627899	0.373627557
0.6	0.291404256	0.301359885	0.310958768	0.310953242	0.310952904
0.7	0.242707257	0.252202935	0.261404568	0.261399270	0.261398947
0.8	0.205105754	0.213956311	0.222575989	0.222571024	0.222570721
0.9	0.176396883	0.184492463	0.192416882	0.192412317	0.192412038
1.0	0.154715925	0.162003293	0.169173489	0.169169356	0.169169104
	Euler		Runge-Kutta 4		Exact

We can clearly see that even with a bigger step (0.1), the Runge-Kutta 4 method is much more accurate than the Euler method at every point. With a 0.05 step, the error is of order 10^{-7}.

Python implementation

We program the Runge-Kutta 4 method to solve

$$\begin{cases} y' + 2y = x^3 e^{-2x}, \\ y(0) = 1 \end{cases}$$

We will basically use the same concept and parameters of the Euler method we did earlier

```
def rk4(f, x0, y0, x1, n):
        vx = [0] * (n + 1)
        vy = [0] * (n + 1)
        h = (x1 - x0) / float(n)
        vx[0] = x = x0
        vy[0] = y = y0
        for i in range(1, n + 1):
                k1 = h * f(x, y)
                k2 = h * f(x + 0.5 * h, y + 0.5 * k1)
                k3 = h * f(x + 0.5 * h, y + 0.5 * k2)
                k4 = h * f(x + h, y + k3)
                vx[i] = x = x0 + i * h
                vy[i] = y = y + (k1 + k2 + k2 + k3 + k3 + k4) / 6
        return vx, vy

vx, vy = rk4(f, 0, 1, 10, 100)
```

We combine the two codes so we can get a plot that shows all 3 methods, in which we see how accurate the RK4 method is compared to the Euler method:

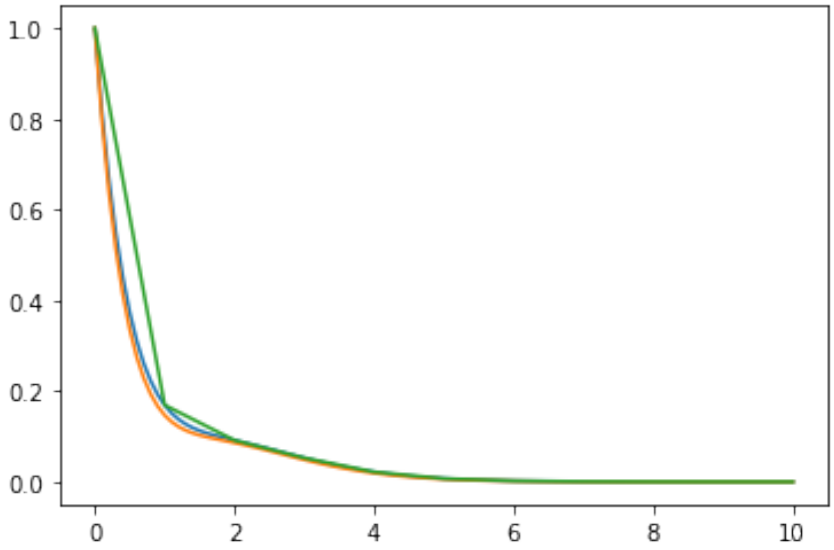

FIGURE 2.4: Plot of the solutions: Euler (green), RK4(blue), exact (yellow)

Remarks

We can always be content with the performance of the Runge-Kutta 4 method, but we still face some difficulties with the discretisation, since step by step methods tend to be quite obsolete for problems of large time intervals. There's also the additional constraint of systems -be it first order of higher order- which can be pretty tedious to program for these classical methods. This is where the interest of deep learning lies: we can completely absolve these constraints, and we get the added bonus of accuracy. This stems from the fact that deep learning uses sophisticated and high-performance techniques and algorithms that we shall introduce in the following chapter.

Neural Networks and Deep Learning

3.1 Introduction

Deep learning is based on large and complex networks made up of a large number of simple computational units: artificial neural networks. On a basic level, ANNs are a large set of differently interconnected units, each performing a specific (and usually relatively easy) computation. Using simple computational units, you can build very complex systems. In its most general form, an artificial neural network is a machine that is designed to model the way in which the brain performs a particular task or function of interest. The ANNs are usually implemented by using electronic components or are simulated in software in a digital computer. We can vary the basic units, changing how they compute the result, how they are connected to each other, how they use the input values, and so on. Roughly formulated, all those aspects define what is known as the network architecture. Changing it will change how the network learns, how accurate the predictions are, and so on.

3.1.1 Analogy with the human brain

Work on artificial neural networks has been inspired right from its inception by the acknowledgment that the human brain computes in an entirely different way from the traditional digital (von Neumann) computer.

The brain is a highly complex, nonlinear, and parallel computer (information-processing system). It has the capability to organize its structural constituents, known as neurons, so as to perform certain computations (e.g., pattern recognition, perception, and motor control) many times faster than the fastest digital computer in existence today.

Consider, for example, human vision, which is an information-processing task. It is the function of the visual system to provide a representation of the environment around us and, more important, to supply the information we need to interact with the environment. To be specific, the brain routinely accomplishes perceptual recognition tasks (e.g., recognizing a familiar face embedded in an unfamiliar scene) in approximately 100–200 ms, whereas tasks of much lesser complexity take a great deal longer on a powerful computer.

The human nervous system may be viewed as a three-stage system. Central to this system is the brain, represented by the neural (nerve) net, which continually receives information, perceives it, and makes appropriate decisions. Two sets of arrows are shown in the figure. Those pointing from left to right indicate the forward transmission of information-bearing signals through the system. The arrows pointing from right to left signify the presence of feedback in the system. The receptors convert stimuli from the human body or the external

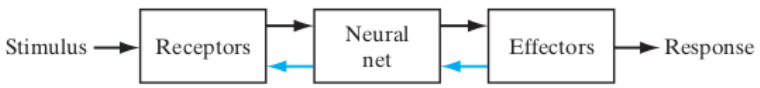

FIGURE 3.1: Block diagram representation of nervous system.

environment into electrical impulses that convey information to the neural net (brain). The effectors convert electrical impulses generated by the neural net into discernible responses as system outputs.

The struggle to understand the brain has been made easier because of the work of Ramón y Cajál (1911), who introduced the idea of **neurons** as structural constituents of the brain. Typically, neurons are five to six orders of magnitude slower than silicon logic gates; events in a silicon chip happen in the nanosecond range, whereas neural events happen in the millisecond range. However, the brain makes up for the relatively slow rate of operation of a neuron by having a staggering number of neurons (nerve cells) with massive interconnections between them. It is estimated that there are approximately 10 billion neurons in the human cortex, and 60 trillion synapses or connections (Shepherd and Koch, 1990).

The net result is that the brain is an enormously efficient structure. Specifically, the energetic efficiency of the brain is ap- proximately 10 -16 joules (J) per operation per second, whereas the corresponding value for the best computers is orders of magnitude larger.

3.2 Artificial Neural Networks

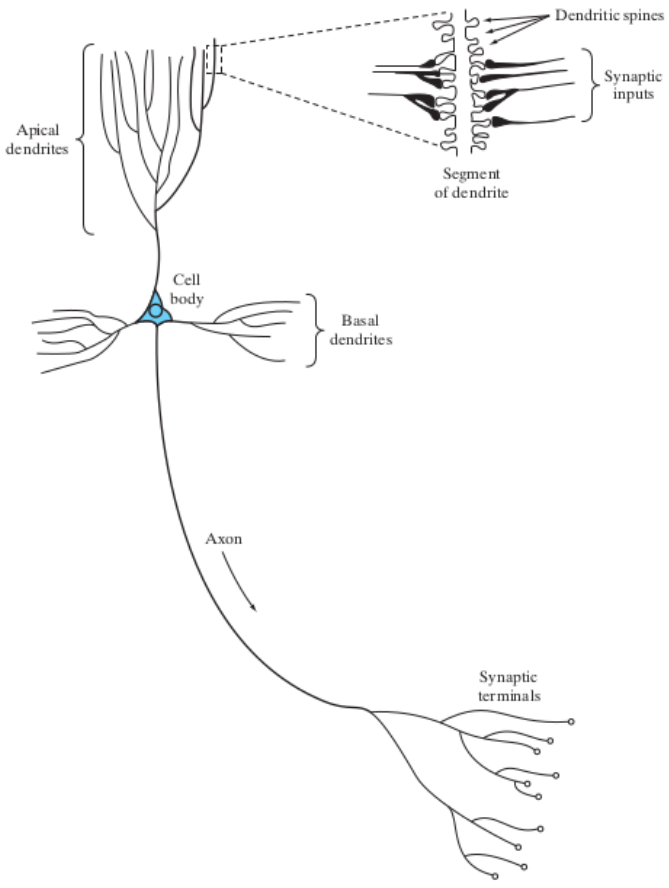

FIGURE 3.2: Graphic representing a neuron

Figure 3.2 shows a hierarchy of interwoven levels of organization that has emerged from the extensive work done on the analysis of local regions in the brain (Shepherd and Koch, 1990; Churchland and Sejnowski, 1992).

The transfer of "information", for the sake of simplicity, is done through the tree in figure 3.3

It is important to recognize that the structural levels of organization described herein are a unique characteristic of the brain. They are nowhere to be found in a digital computer, and we

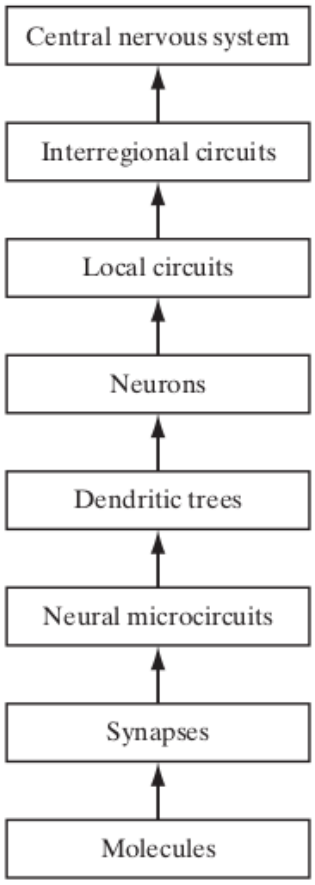

FIGURE 3.3: Structural organization of levels in the brain.

are nowhere close to re-creating them with artificial neural networks. Nevertheless, we are inching our way toward a hierarchy of computational levels similar to that described in Fig. 3.2.

The artificial neurons we use to build our neural networks are truly primitive in comparison with those found in the brain. The neural networks we are presently able to design are just as primitive compared with the local circuits and the interregional circuits in the brain. What is really satisfying, however, is the remarkable progress that we have made on so many fronts. With neurobiological analogy as the source of inspiration, and the wealth of theoretical and computational tools that we are bringing together, it is certain that our understanding of artificial neural networks and their applications will continue to grow in depth as well as breadth, year after year.

Understanding the artificial neuron

Artificial neurons, which from hereon will be simply referred to as "Neurons". Also known as units, nodes or cells, are simple processing elements. Neurons are connected through communication links associated to weights. They receive input, combine the input with their internal state (activation) and an optional threshold using an activation function, and produce output using an output function.

Basically, each neuron does a very simple thing: takes a certain number of inputs (real numbers) and calculates an output (also a real number). In our case, inputs will be indicated by $x_i \in \mathbb{R}$ (real numbers), with $i = 1, 2, ..., n_x$, where $i \in \mathbb{N}$ is an integer and n_x is the number of input attributes (often called features). As an example of input features, you can imagine the

age and weight of a person (so, we would have $n_x = 2$). x_1 could be the age, and x_2 could be the weight. In real life, the number of features easily can be very big.

We will concentrate on the most commonly used one: the ***perceptron***. The neuron we are interested in simply applies a function to a linear combination of all the inputs. In a more mathematical form, given n_x, real parameters w_i (with $i = 1, 2, ..., n_x$), and a constant $b \in \mathbb{R}$ (usually called bias), the neuron will calculate first what is usually indicated in literature and in books by z.

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_{n_x} x_{n_x} + b$$

It will then apply a function f to z, giving the output $\hat{y}$.

$$\hat{y} = f(z) = f(w_1 x_1 + w_2 x_2 + \cdots + w_{n_x} x_{n_x} + b)$$

Owing to a biological parallel, the function f is called the neuron activation function (and sometimes transfer function), which will be discussed at length in the next sections.

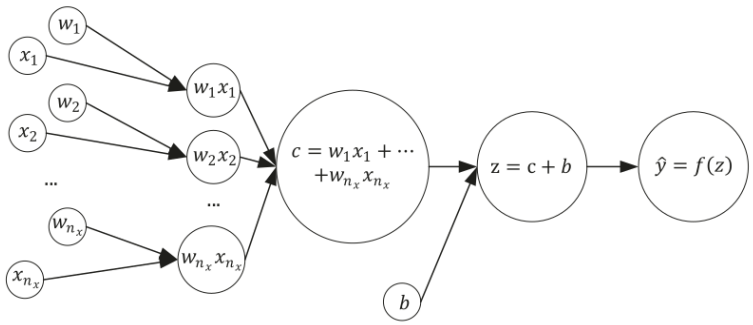

FIGURE 3.4: The computational graph for the neuron described above

For the sake of simplicity, we shall use a more straightforward neuron model, although it's the equivalent of the one in Figure 3.4

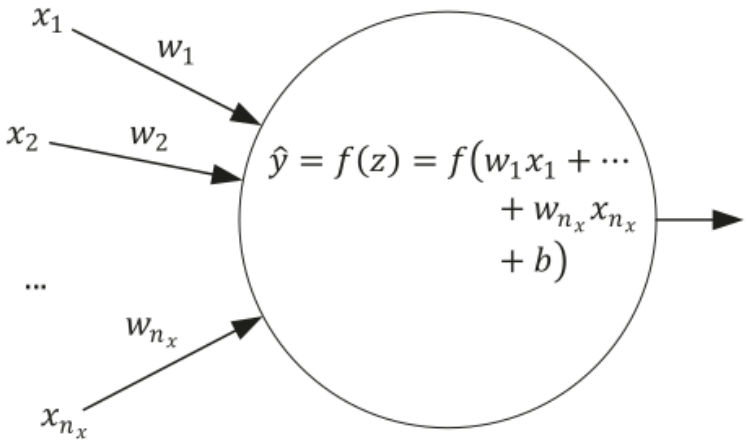

FIGURE 3.5: A simpler neuron representation

Matrix notations

Considering that deep learning problems deal with very large datasets, it is recommendable to use matrix notations for inputs, weights and outputs i.e :

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_{n_x} \end{pmatrix} \text{ and } w = \begin{pmatrix} w_1 \\ \vdots \\ w_{n_x} \end{pmatrix}$$

Furthermore will use the matrix multiplication notation to multiply x and w such that:

$$w^T x = (w_1, \ldots, w_{n_x}) \begin{pmatrix} x_1 \\ \vdots \\ x_{n_x} \end{pmatrix} = w_1 x_1 + \cdots + w_{n_x} x_{n_x}$$

Multilayered feedforward neural networks

In a layered neural network the neurons are organised in the form of layers. We have at least two layers: an input and an output layer. The layers between the input and the output layer (if any) are called hidden layers, whose computation nodes are correspondingly called hidden neurons or hidden units. The source nodes in the input layer of the network supply respective elements of the activation pattern (input vector), which constitute the input signals applied to the neurons (computation nodes) in the second layer (i.e., the first hidden layer). The output signals of the second layer are used as inputs to the third layer, and so on for the rest of the network. A layer of nodes projects onto the next layer of neurons (computation nodes), but not vice versa. In other words, this network is strickly a feedforward or acyclic type. The neurons in each layer of the network have as their inputs the output signals of the preceding layer only. The set of output signals of the neurons in the output (final) layer of the network constitutes the overall response of the network to the activation pattern supplied by the source nodes in the input (first) layer. Neural networks with this kind of architecture are also called multilayer perceptron (MLP).

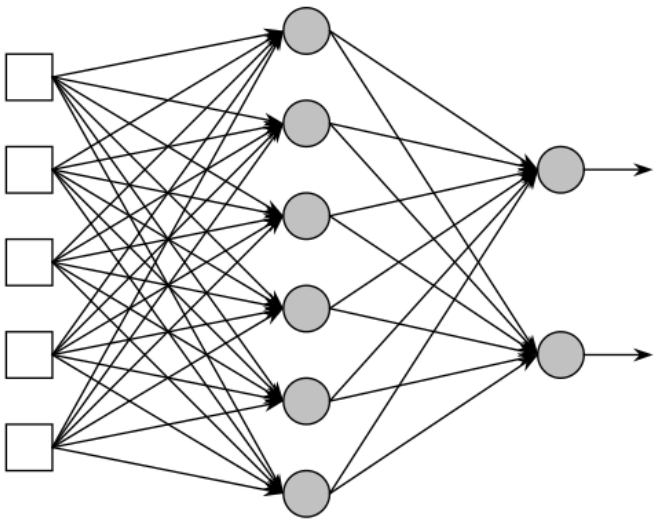

FIGURE 3.6: A fully connected feed-forward neural network with one hidden layer

The function of hidden neurons is to intervene between the external input and the network output in some useful manner. By adding one or more hidden layers, the network is enabled to extract high-order statistics. In a rather loose sense the network acquires a global perspective despite its local connectivity due the extra synaptic connections and the extra dimension of neural interactions. The neural network is said to be fully connected in the sense that every node in each layer of the network is connected to every other node in the adjacent forward layer, otherwise the network is called partially connected.

3.3 The learning process

The ability to learn is the key aspect of neural networks. The aim of this process is to find the optimal parameters (and structure) of the network for solving the given task. Before the training process starts, network parameters need to be initialized. Initial values are often chosen randomly, but some applications use available datasets which leads to a faster parameter adjustment towards the optimal values. Learning is then carried out on the training set by feeding the training data through the network.

It is an iterative process, where the outputs produced on each input from the training set are analyzed and the network is repeatedly being adjusted to produce better results.

The network is considered to be trained after reaching the target performance on the training data.

There are two type of learning processes:

- **Supervised learning**, *which will be the type we'll study in the following section since our subject library is based on such a learning model (we will elaborate more on the following chapter)*

  It is called supervised learning because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process. We know the correct answers, the algorithm iteratively makes predictions on the training data and is corrected by the teacher. Learning stops when the algorithm achieves an acceptable level of performance.

  Supervised learning problems can be further grouped into regression and classification problems.

  - Classification: A classification problem is when the output variable is a category, such as "red" or "blue" or "disease" and "no disease".
  - Regression: A regression problem is when the output variable is a real value, such as "dollars" or "weight".

  Some common types of problems built on top of classification and regression include recommendation and time series prediction respectively.

  Some popular examples of supervised machine learning algorithms are:

  - Linear regression for regression problems.
  - Random forest for classification and regression problems.
  - Support vector machines for classification problems.

- ***Unsupervised learning***

  Unsupervised learning is where you only have input data (X) and no corresponding output variables.

  The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to learn more about the data.

  These are called unsupervised learning because unlike supervised learning above there is no correct answers and there is no teacher. Algorithms are left to their own devises to discover and present the interesting structure in the data.

  Unsupervised learning problems can be further grouped into clustering and association problems.

  - Clustering: A clustering problem is where you want to discover the inherent groupings in the data, such as grouping customers by purchasing behavior.

- Association: An association rule learning problem is where you want to discover rules that describe large portions of your data, such as people that buy X also tend to buy Y.

Some popular examples of unsupervised learning algorithms are:

- k-means for clustering problems.
- Apriori algorithm for association rule learning problems.

Key definitions

- ***Loss function***

  For the process of learning, we first choose a suitable activation function and then define a loss function (also known as an error function, or simply *loss*), whose value is optimized (maximized or minimized) by changing the ANN parameters. It is desirable to have a loss function that is convex with a continuous first derivative. This simplifies the optimization, which can be performed using the gradient descent or related algorithms. The most commonly used one is the squared-sum error function defined as:

$$E = \frac{1}{2}\sum_{n=1}^{N} \|f(x_n; w, b) - t_n\|^2$$

  where N is the number of samples, x_n is the nth input sample, t_n its output,$f(;w,b)$ and $\|\|$ resp. represent the ANN output and the L^2 norm. The optimal values of the parameters $-\tilde{W}$ and $\tilde{b}-$ are obtained by solving an optimization problem.

  A smooth and convex loss function can be optimized using the gradient optimization method provided that the derivative of the loss function is known.

- **Backpropagation**

  Backpropagation is the application of the chain rule[1] of calculating the derivative in ANN. In backpropagation, the information first flows from the input to the output and then in the reverse direction. The difference of the predicted output and the actual output (the error) is then determined. The error made by the network to predict true value or label is differentiated with respect to all the parameters and weights of the network and then the weights are adjusted accordingly.

  In fact, for each set, we compute the prediction and its associated loss. We sum up the loss to compute the final error. Then we use the backpropagation algorithm to propagate the error in order to compute the partial derivatives $\frac{\partial E}{\partial w}$ and $\frac{\partial E}{\partial b}$ of the loss function E for all weights w and bias b.

- **Optimization**

  Once all the derivatives are computed, we update our parameters using a chosen optimization technique such as Stochastic Gradient Descent. We then iterate the prediction (e.g. forward pass), the backpropagation of errors (e.g. backward pass) and the optimization until convergence, eventually to find a local minimum low enough to ensure good predictions. Even if the chosen surrogate loss function of a neural network is non convex, GD works well in practice. Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. Every

[1] the chain rule is a formula to compute the derivative of a composite function. That is, if f and g are differentiable functions, then the chain rule expresses the derivative of their composite $f \circ g$ in terms of the derivatives of f and g and the product of functions as follows:$(f \circ g)' = (f' \circ g) \cdot g'$.

deep learning library contains implementations of various algorithms to optimize gradient descent (e.g. lasagne, caffe, and keras). These algorithms, however, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by. Other optimization algorithms are "adam" and "Limited-memory BFGS".

- **Overfitting**

  A common problem encountered in the process of learning is overfitting. It generally occurs when the learning is performed for too long, and especially when the training set is too small to evenly represent all types of patterns from the domain of possible network inputs. In such a case the learning may adjust the network to random features present in the training data. Overfitting is observed during the learning process, when the network's predictive performance is improving on the training set, however worsening on previously unseen test data. To combat this issue the labeled data is split into a training and a validation set. The main reason why to use the validation set is that it shows the error rates on the data independent from the data we are training on. A study by Guyon suggests that the optimal ratio between the size of training vs. validation data set depends on the number of recognized classes and the complexity of class features. An estimation of feature complexity is however quite cumbersome. While learning, the performance of the ANN is regularly examined on validation data set. When errors retrieved on validation data reach a stopping point, learning process is stopped and the network is considered trained.

- **Hyperparameters**

  Hyperparameters are variables that we need to set before applying a learning algorithm to a dataset.

  - **Number of hidden layers**: adding more hidden layers of neurons generally improves accuracy, to a certain limit which can differ depending on the problem.
  - **Dropout**: what percentage of neurons should be randomly "killed" during each epoch to prevent overfitting.
  - **Neural network activation function** : which function should be used to process the inputs flowing into each neuron. The activation function can impact the network's ability to converge and learn for different ranges of input values, and also its training speed.
  - **Weight initializer** it is necessary to set initial weights for the first forward pass. Two basic options are to set weights to zero or to randomize them. However, this can result in a vanishing or exploding gradient, which will make it difficult to train the model. To mitigate this problem, we use a heuristic[2] to determine the weights. Common heuristics used for the Tanh activation are Xavier initialization or Glorot Uniform.
  - **Learning rate**: how fast the backpropagation algorithm performs gradient descent. A lower learning rate makes the network train faster but might result in missing the minimum of the loss function
  - **Optimizer algorithm and neural network momentum** : these parameters determine the rate at which samples are fed to the model for training. An epoch is a group of samples which are passed through the model together (forward pass) and then run through backpropagation (backward pass) to determine their optimal weights. If the epoch cannot be run all together due the size of the sample or complexity of the network, it is split into batches, and the epoch is run in two or more iterations. The number of epochs and batches per epoch can significantly affect model fit.

[2] a formula tied to the number of neuron layers

Activation functions

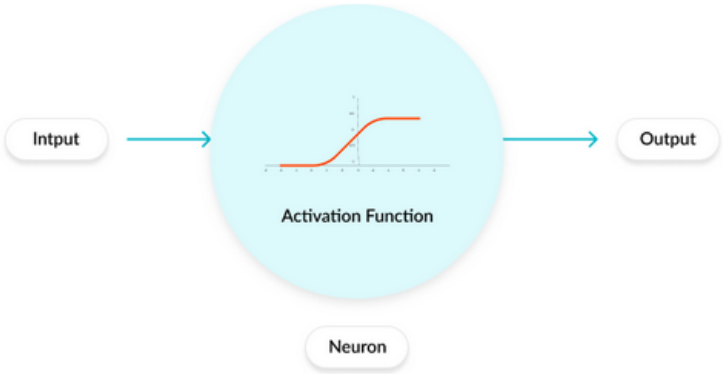

FIGURE 3.7: Role of an activation function

Activation functions are mathematical functions that determine the output $\hat{y}$ of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated ("fired") or not, based on whether each neuron's input z is relevant for the model's prediction. Activation functions also help normalize the output of each neuron to a range between 1 and 0 or between -1 and 1.

An additional aspect of activation functions is that they must be computationally efficient because they are calculated across thousands or even millions of neurons for each data sample. Modern neural networks use a technique called backpropagation to train the model, which places an increased computational strain on the activation function, and its derivative function.[3]

There are many activation functions at our disposal to change the output of our neuron. Remember: An activation function is simply a mathematical function that transforms z in the output $\hat{y}$.

We have 3 types of activation functions:

- ***Binary step functions***

  A binary step function is a threshold-based activation function. If the input value is above or below a certain threshold, the neuron is activated and sends exactly the same signal to the next layer.

  The most commonly used one is the Heaviside step function (or the unit step function), which is usually denoted by H or θ, is a discontinuous function, named after Oliver Heaviside (1850–1925), whose value is zero for negative arguments and one for positive arguments.

  The value of the Heaviside function at the position $x = 0$ can also be set as follows. To identify the definition:

$$\Theta_c : \mathbb{R} \to \mathbb{K}$$
$$x \mapsto \begin{cases} 0 : & x < 0 \\ c : & x = 0 \\ 1 : & x > 0 \end{cases}$$

  with $0, 1, c \in \mathbb{R}$

  The problem with a step function is that it does not allow multi-value outputs e.g it cannot support classifying the inputs into one of several categories. Therefore we need a function

[3] We will be explaining these notions later in this chapter

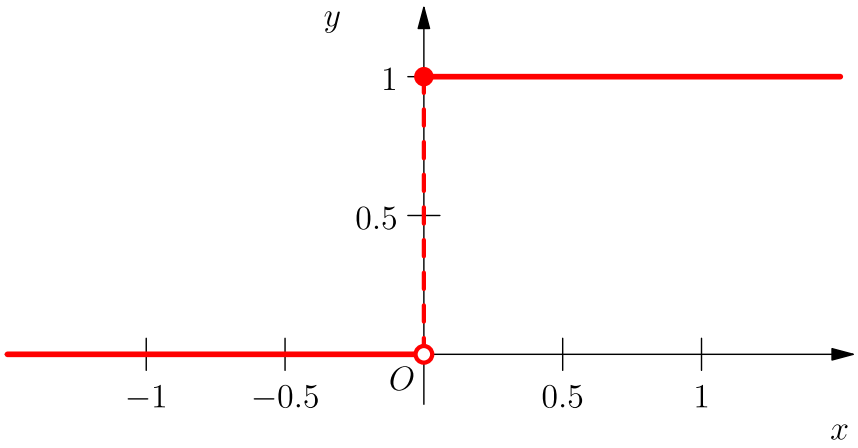

FIGURE 3.8: Heaviside step function

that gives us intermediate (analog) activation values rather than “activated or not" (binary).

The first thing that comes to our minds would be linear functions.

- ***Linear activation functions A = cx***

  A straight line function where activation is proportional to input (which is the weighted sum from neuron).

  The most commonly used linear function is the Identity.

  It is the most basic function that we can use. We indicate it by I(z). It returns simply the input value unchanged. Mathematically we have

$$f(z) = I(z) = z$$

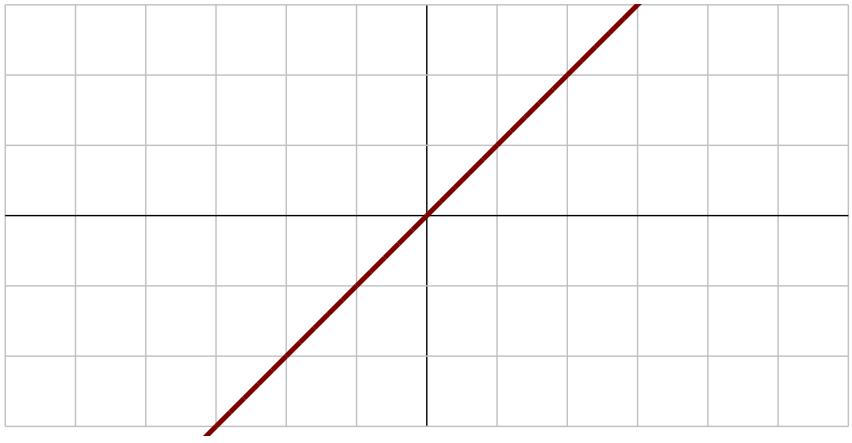

FIGURE 3.9: Identity activation function

This way, it gives a range of activations, so it is not binary activation.

We can connect neurons together and if more than 1 fires, we could take the max and decide based on that.

But there’s a problem:

For this function, the derivative is a constant: $A = cx$, its derivative with respect to x is c. That means, the gradient has no relationship with X. It is a constant gradient and the descent is going to be on constant gradient. If there is an error in prediction, the changes made by back propagation are constant and not depending on the change in the input.

There is another problem : connected layers. Each layer is activated by a linear function. That activation in turn goes into the next level as input and the next layer calculates weighted sum on that input and in its turn fires based on another linear activation function.

No matter how many layers we have, if all are linear in nature, the final activation function of the last layer is nothing but just a linear function of the input of first layer.

That means these two layers (or N layers) can be replaced by a single layer. We just lost the ability to stack layers this way. No matter how we stack, the whole network is still equivalent to a single layer with linear activation (a combination of linear functions in a linear manner is still another linear function). Which brings us to the use of non-linear functions

- ***Non-linear activation functions***

  Modern neural network models use non-linear activation functions. They allow the model to create complex mappings between the network's inputs and outputs, which are essential for learning and modeling complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality.

  Almost any process imaginable can be represented as a functional computation in a neural network, provided that the activation function is non-linear. [4]

  Non-linear functions address the problems of a linear activation function:

  They allow *backpropagation* because they have a derivative function which is related to the inputs. They allow "stacking" of multiple layers of neurons to create a deep neural network. Multiple hidden layers of neurons are needed to learn complex data sets with high levels of accuracy.

  - ***Sigmoid function*** (also known as Logistic function or Soft step)
    This is a very commonly used function that gives only values between 0 and 1. It is usually indicated by $\sigma(z)$.

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

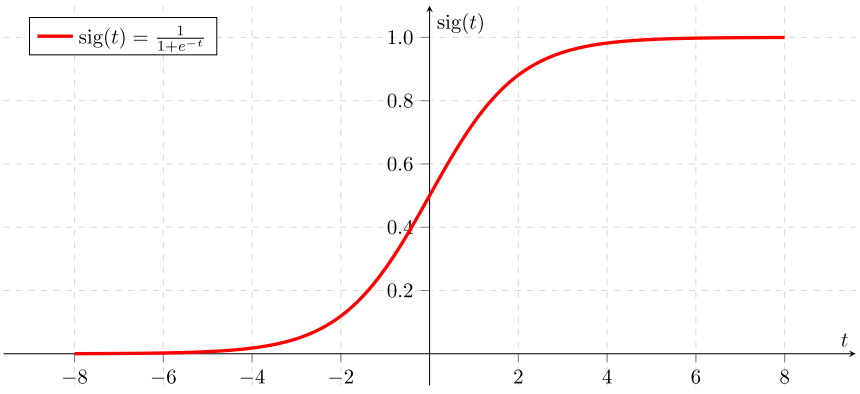

FIGURE 3.10: Sigmoid function

It tends to bring the activations to either side of the curve (above x = 2 and below x = -2 for example). Making clear distinctions on prediction.
Another advantage of this activation function is, unlike linear functions, the output of the activation function is always going to be in range (0,1) compared to $(-\infty, +\infty)$ of a linear function. So we have our activations bound in a range.

[4] to be further explored later

Sigmoid functions are one of the most widely used activation functions today.
The gradient on either end is small or has vanished (cannot make significant change because of the extremely small value). The network will therefore refuse to learn further or will be very slow. There are ways to work around this problem and sigmoid is still very popular in classification problems.

– ***Hyperbolic tangent (tanh)***

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} = 1 - \frac{2}{e^{2x} + 1}$$

Tanh has characteristics similar to sigmoid: it is nonlinear in nature, so we can stack layers. It is also bound to the range [-1, 1] so no activations blowing up. The gradient is also stronger for tanh than sigmoid (derivatives are steeper). Deciding between the sigmoid or tanh will depend on your requirement of gradient strength. Like sigmoid, tanh also has the vanishing gradient problem.

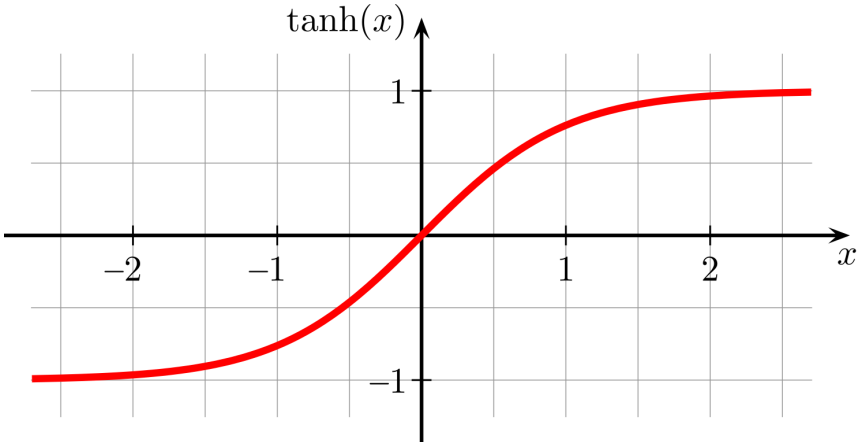

FIGURE 3.11: Hyperbolic tangent

– ***ReLU (Rectified Linear Unit)***
The rectifier is an activation function defined as the positive part of its argument:

$$f(x) = x^+ = \max(0, x),$$

where x is the input to a neuron. This is also known as a ramp function and is analogous to half-wave rectification in electrical engineering.

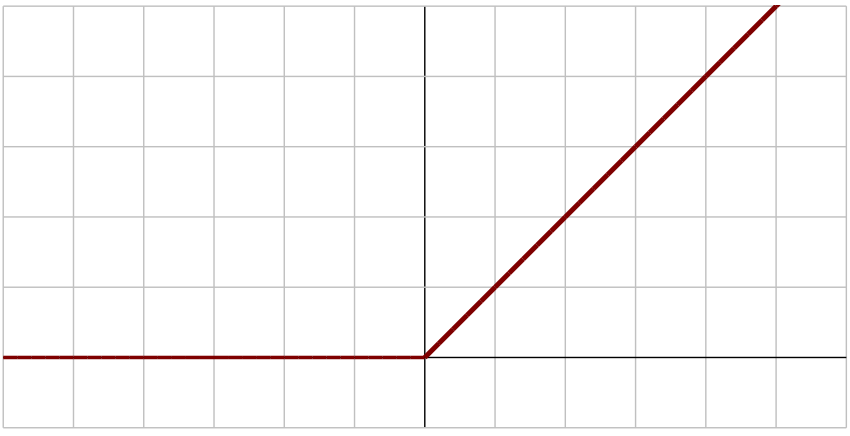

FIGURE 3.12: Rectified linear unit

Because of the horizontal line in ReLu(for negative X), the gradient can go towards 0. For activations in that region of ReLu, the gradient will be 0, because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called dying ReLu problem. This problem can cause several neurons to just die and not respond making a substantial part of the network passive.

There are variations in ReLu to mitigate this issue by simply making the horizontal line into non-horizontal component . for example $y = 0.01x$ for $x < 0$ will make it a slightly inclined line rather than horizontal line. This is ***leaky ReLu***. There are other variations too. The main idea is to let the gradient be non zero and recover during training eventually.

ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. That is a good point to consider when we are designing deep neural nets.

Gradient descent optimization algorithms

Optimization through gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum.

There are three variants of gradient descent, the difference being in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

a) **Batch gradient descent**: Commonly known as "Vanilla" gradient descent, it computes the gradient of the cost function w.r.t.to the parameters θ for the entire training dataset:

$$\theta = \theta - \eta \nabla_\theta J(\theta)$$

And since we need to calculate the gradients for the whole dataset to perform just one update, batch gradient descent can be very slow and is impractical for datasets that do not fit in memory. Batch gradient descent also does not allow us to update our model online, i.e. with new examples on-the-fly.

Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

b) **Stochastic gradient descent**

Stochastic gradient descent (SGD) on the other hand performs a parameter update for each training example $x^(i)$ and label $y^(i)$:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

As we noted, batch gradient descent performs redundant computations for large datasets, because it recomputes gradients for similar examples before each parameter update. SGD avoids this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily.

And while batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to

the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

c) **Mini-batch gradient descent**

Mini-batch gradient descent is an accumulation of both sides: it performs an update for every *mini-batch* of n training examples:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i;i+n)}; y^{(i;i+n)})$$

This way, it both reduces the variance of the parameter updates, which can lead to more stable conver- gence ***and*** can make use of highly optimized matrix optimizations common to advanced deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used.

The formal process of learning

In this subsection, we will concisely explain the important aspects of the learning (i.e training) process of an artificial neural network.

Let us consider neuron j with its input $\overrightarrow{x_j} = (x_{1j}, ..., x_{nj})$, weights $w_{1j}, ..., w_{nj}$ and bias b_j Then the potential of the neuron is computed:

$$z_j = \sum_{i=1}^{n} w_{ij} x_{ij} + b_j$$

Consider f the activation function such as:

$$f(z_j) = \frac{1}{1 + exp(-z_j)}$$

Therefore the output y_i of the neuron j is:

$$y_j = f(z_j)$$

We shall consider that our neural network consists of m such neurons in the output layer, therefore the output of the network is $\overrightarrow{y} = (y_1, ..., y_m)$

Suppose that our ANN is a multi-layered perceptron. It shall be split into mutually disjunct layers $L_1, ..., L_k$ $(k)2$, L_1 being the input layer, $L_2, ..., L_{k-1}$ being the hidden layers and L_k being the output layer.

Now let us consider P training patterns labeled $(\overrightarrow{x^p}, \overrightarrow{d^p})$, where $\overrightarrow{x^p}$ is the input vector, $\overrightarrow{d^p}$ is the desired output vector, and $1 \leq p \leq P$. Given the current configuration of the network, the input $\overrightarrow{x^p}$ yields the output $\overrightarrow{y^p}$. Then for every pattern p we want $\overrightarrow{y^p}$ to be as close to the desired output $\overrightarrow{d^p}$ as possible.

We can define the error of each neuron j in the output layer as:

$$e_j^p = y_j^p - d_j^p$$

Now we can define the squared error for pattern p as:

$$E_p = \frac{1}{m_k} \sum_{j=1}^{m_k} (e_j^p)^2 = \frac{1}{m_k} \sum_{j=1}^{m_k} (y_j^p - d_j^p)^2 \quad \star$$

where m_k is the number of neurons in the output layer. Note that if the actual output is exactly the same as the desired output, we get zero for the squared error. In other words the following holds true:

$$\forall j : E_p = 0 \Leftrightarrow e_j^p = 0 \Leftrightarrow y_j^p = d_j^p$$

It may be useful to sum up the average error for all input patterns to assess the network performance on the whole dataset, which can be achieved simply by computing the mean squared error:

$$E_{avg} = \frac{1}{P} \sum_{p=1}^{P} E_p$$

When learning, for each interconnected pair of neurons (i, j), where i is a neuron in layer l, j is a neuron in layer $l + 1$ and $w_{i,j}$ weights their connection, we want to adjust $w_{i,j}$ to minimize the mean squared error E_{avg} . Provided the activation function is differentiable everywhere on its domain, E_{avg} is also differentiable. When adjusting the weight $w_{i,j}$ of the neuron j located in the output layer k, we are therefore interested in the partial derivative:

$$\frac{\partial E_{avg}}{\partial w_{i,j}} = \frac{1}{P} \frac{\partial}{\partial w_{i,j}} \sum_{p=1}^{P} E_p = \frac{1}{P} \sum_{p=1}^{P} \frac{\partial E_p}{\partial w_{i,j}} \qquad (\star\star)$$

To be able to adjust the network after each presented input pattern, we are actually interested in the derivative for each given pattern p and its corresponding E_p . In the following equations we will therefore omit the pattern index p. Applying the chain rule to Equation $\star\star$ we get:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w_{i,j}}$$

Using equation $\star$ we can evaluate the first factor as

$$\frac{\partial E}{\partial y_j} = (y_j - d_j)$$

Then we evaluate the second factor:

$$\frac{\partial y_j}{\partial w_{i,j}} = \frac{\partial y_j}{\partial z_j} \frac{\partial z_j}{\partial w_{i,j}} = f'(z_j) \frac{\partial}{\partial w_{i,j}} \sum_k w_{k,j} y_k = f'(z_j) y_i$$

By combining both evaluated factors we get:

$$\frac{\partial E}{\partial w_{i,j}} = (y_j - d_j) f'(z_j) y_i \qquad (1)$$

We defines the local gradient δ_j for neuron j in the output layer as follows:

$$\delta_j = \frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j} = (y_j - d_j) f'(z_j)$$

We therefore get from (1):

$$\frac{\partial E}{\partial w_{i,j}} = \delta_j y_i \qquad (2)$$

Using Equation (2) we can compute the gradient of the error function for each of the given patterns p.

We need to adjust the weight $w_{i,j}$ proportionally to the gradient but in the opposite direction. However doing so for every input pattern would produce a very unstable system. To combat

this problem we can use a learning parameter $0 < \eta < 1$. The weight adjustment is then computed by:

$$\Delta w_{i,j} = -\eta \frac{\partial E}{\partial w_{i,j}} = -\eta \delta_j y_i \quad (3)$$

The weight adjustment $\delta w_{i,j}$ in (3) is only applicable to the neurons in the output layer. Computation of the adjustment for neurons in the hidden layers is more complicated. For instance, consider 3 neurons i, j, and k, all following each other on the same path along the layers l - 1, l and l + 1, respectively, as illustrated in Figure 3.13. Then the adjustment of $w_{i,j}$ needs to be done carefully, because besides influencing the output of neuron i itself, it also impacts all the outputs (and thus errors) in all layers following l.

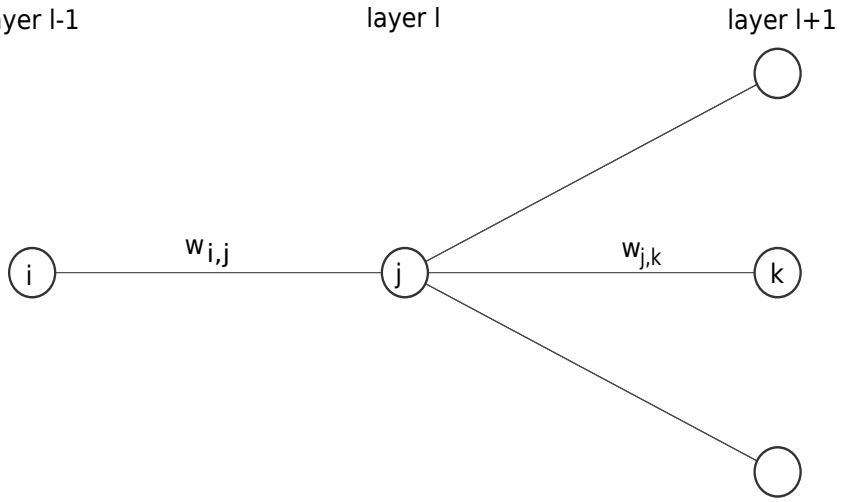

FIGURE 3.13: Illustration showing how a change in the weight w i,j of the neuron in the hidden layer l 1 influences the weight w j,k of the neuron in the following layer l

Note that Equation (1) still applies to hidden layers. However, we need to look at the definition of the local gradient again. In the previous Equation (2) we are using the desired output $\overrightarrow{d}^j$ to calculate $\frac{\partial E}{\partial y_j}$. Of course, there is no desired output known in hidden layers. It is actually dependent on the network design. Because of this, we need to step back and use the following definition for δ_j

$$\delta_j = \frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j} = \frac{\partial E}{\partial y_j} f'(z_j) \qquad (4)$$

Now we need to redefine $\frac{\partial E}{\partial y_j}$ for hidden layers. For any hidden layer l, the following layer $l+1$ must exist (otherwise l would be the output layer). Given the neurons i, j and k each in a different layer as illustrated in Figure 3.13, we can use the potential z_k

$$\frac{\partial E}{\partial y_j} = \sum_{k=1}^{m_l+1} \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial y_j} = \sum_{k=1}^{m_l+1} \frac{\partial E}{\partial z_k} w_{|j,k} = \sum_{k=1}^{m_l+1} \delta_k w_{j,k} \qquad (5)$$

Combining equations (4) and (5) we get:

$$\delta_j = (\sum_{k=1}^{m_l+1} \delta_k w_{j,k}) f'(z_j) \qquad (6)$$

Equation 6 tells us, that knowing the local gradient of neuron k in layer l+1, we can calculate the local gradient for neuron j in layer l. This fact will allow us to recursively adjust the network

weights going layer by layer in the direction from the output to the input (backwards). It is used in the most common learning algorithm: Backpropagation.

Finally, we can summarize the weight adjustment applicable to all the layers in a given network:

$$\Delta w_{i,j} = -\eta \frac{\partial E}{\partial w_{i,j}} = -\eta \delta_j y_i$$

where

$$\forall j \text{ in layer } l < L \qquad \delta_j = \delta_j = (\sum_{k=1}^{m_l+1} \delta_k w_{j,k}) f'(z_j)$$

and also

$$\forall j \text{ in layer L} \qquad \delta_j = (y_j - d_j) f'(z_j)$$

The backpropagation algorithm

As described in the definition above, the purpose of the backpropagation algorithm is to adjust the synaptic weights of neurons, so that the network produces the desired output. The result of this algorithm is a neural network configured to minimize the error when solving a given problem.

Each input pattern with its class label $(\overrightarrow{x^p}, \overrightarrow{d^p})$ is sequentially processed in two phases. The first phase called the forward phase puts the input pattern as the input of the network, setting $\overrightarrow{y^0} = \overrightarrow{x^p}$. The network then computes its output $\overrightarrow{x^L}$. The sole purpose of the forward phase is to calculate the output for the presented pattern, and the network is not adjusted at all. At this point the backward phase starts. The purpose of this phase is to adjust the network weights to achieve a better assessment of the input data. Learning is performed in multiple epochs. In each epoch, all the data from the training set is processed. The duration of an epoch directly depends on the size of the network and the size of the training set, as each input pattern,from the training set is processed exactly once. However the number of epochs is not limited. An important decision of the learning process is therefore the determination of the stopping criterion. After each epoch we validate the error on the validation data set. Once this error starts to increase, we achieved some (local or perhaps global) minimum, and it is usually wise to stop the learning process at this point

Following the notations described above and earlier in this section, a simple depiction the algorithm is as follows:

Algorithm 1: Backpropagation

Input:

Training patterns $(\overrightarrow{x^p}, \overrightarrow{d^p})$,
Validation patterns $(\overrightarrow{v^q}, \overrightarrow{d^q})$,
Activation function f and its derivation $f'(z)$ e.g the sigmoidal function
$f(z) = \frac{1}{1+exp(-z)}$; $f'(z) = f(z)(1-f(z))$,
learning parameter $\eta \in]0,1[$,
Feedforward neural network $\mathcal{M}$ with randomly initialized weights

Output: A trained feedforward neural network $\mathcal{M}'$

1 Set $E_{avg} = \infty$

2 Start a new epoch

3 For each $p \in 1, \ldots, P$ present the pattern $(\overrightarrow{x^p}, \overrightarrow{d^p})$

- Set $\overrightarrow{y^0} = \overrightarrow{y^p}$
- **Forward phase**:
  for l=1,2,...,k compute the output of for $\overrightarrow{x^p}$: $\forall j \in L_i$ compute $z_j = \sum_{i=1}^{n} w_i x_i + b$ and $y_j = f(z)$
- **Backward phase**:
  - $\forall i \in L_{k-1}, j \in L_k$ compute $\delta_j = (y_j - d_j^p) f'(z_j)$ and $\Delta w_{i,j} = -\eta \delta_j y_i$
  - for l = k - 1, . . . , l do:
    $\forall i \in L_{l-1}, j \in L_l compute \delta_j = \sum_{k=1}^{m_l+1} \delta_k w_{j,k} f'(z_j) and \Delta w_{i,j} = -\eta \delta_j y_i$
  - for l=1,2,...,k $\forall (i,j) \in L_{l-1} \times L_i$ adjust $w_{i,j}$ by $\Delta w_{i,j}$
  - Compute the error for the pattern p: $E_p = \frac{1}{m_k} \sum_{j=1}^{m_k} (y_j^p - d_j^p)^2$

4 Set $E_{prev} = E_{avg}$ and compute new $E_{avg} = \frac{1}{P} \sum_{p=1}^{P} E_p$ for the validation data set

5 If $E_{avg} < E_{prev}$ go to step 2

Finish

Python implementation of the training of a neural network

Here is a simple example of the construction of a neural network using a class-based approach in Python, it highlights what we have discussed above.

```
import numpy as np # helps with the math
# input data
inputs = np.array([[0, 1, 0],
[0, 1, 1],
[0, 0, 0],
[1, 0, 0],
[1, 1, 1],
[1, 0, 1]])
# output data
outputs = np.array([[0], [0], [0], [1], [1], [1]])

# create NeuralNetwork class
class NeuralNetwork:

# intialize variables in class
def __init__(self, inputs, outputs):
        self.inputs  = inputs
        self.outputs = outputs
# initialize weights as .50 for simplicity
        self.weights = np.array([[.50], [.50], [.50]])
        self.error_history = []
        self.epoch_list = []

#activation function ==> S(x) = 1/1+e^(-x)
def sigmoid(self, x, deriv=False):
        if deriv == True:
        return x * (1 - x)
        return 1 / (1 + np.exp(-x))

# data will flow through the neural network.
def feed_forward(self):
        self.hidden = self.sigmoid(np.dot(self.inputs, self.weights))

# going backwards through the network to update weights
def backpropagation(self):
        self.error  = self.outputs - self.hidden
        delta = self.error * self.sigmoid(self.hidden, deriv=True)
        self.weights += np.dot(self.inputs.T, delta)

# train the neural net for 25,000 iterations
def train(self, epochs=25000):
        for epoch in range(epochs):
# flow forward and produce an output
self.feed_forward()
# go back though the network to make corrections based on the output
        self.backpropagation()
# keep track of the error history over each epoch
        self.error_history.append(np.average(np.abs(self.error)))
        self.epoch_list.append(epoch)
```

```
50
51  # function to predict output on new and unseen input data
52  def predict(self, new_input):
53          prediction = self.sigmoid(np.dot(new_input, self.weights))
54          return prediction
55
56  # create neural network
57  NN = NeuralNetwork(inputs, outputs)
58  # train neural network
59  NN.train()
```

3.4 Approximation capabilities of an artificial neural network

We have explained how neural networks solve problems by learning: they basically acquire experience, and modify their internal structure in order to accomplish a given task. During the training process, the available information is usually divided into two categories, examples of function values or training data and prior information, e.g. smoothness constraint, or other particular properties. From the learning point of view, the approximation of a function is equivalent with the learning problem of a neural network.

In this section, we will discuss the ***Universal Approximation Theorem***, the theory behind the capabilities of a neural network to approximate arbitrary continuous functions.

The universal approximation theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function. The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters; however, it does not touch upon the algorithmic of those parameters.

George Cybenko, 1989

One of the first versions of the theorem was proved by George Cybenko in 1989 for sigmoid activation functions.

Theorem 1. (Cybenko, 1989).
Let σ be a sigmoidal function.

Then finite sums of the form

$$g(x) = \sum_{j=1}^{N} w_j^2 \sigma((w_j^1)^T x + b_j^1)$$

are dense in $C(I_n)$ with respect to the supremum norm, such that $x \in \mathbb{R}^n, w_j^2 \in \mathbb{R}^n, w_j^1, b_j^1 \in \mathbb{R}$.

I_n is the n-dimensional unit cube $[0,1]^n$ and $C(I_n)$ is noted for the space of continuous functions on I_n.

In other words , given any function $f \in C(I_n)$ and $\epsilon > 0$, there is a sum g(x) of the above form for which

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

The main takeaway from Cybenko's theorem was that any continuous function on a compact subset of $[0,1]^n$ can be approximated by a feed forward neural network with only one single hidden layer and a finite number of neurons.

It is only required that the neural network and its parameters (number of hidden neurons, number of training epochs etc.) must be adjusted for each unique function. Under the right circumstances, it is possible to achieve any desired accuracy ϵ.

Remark. The class of functions that can be approximated does not *always* need to be continuous. For some non-continuous function, a continuous approximation works too and therefore

could be approximated by the neural network (for example a noncontinuous function with liftable positions).

Kurt Hornik, 1991

Kurt Hornik showed in 1991 that it is not the specific choice of the activation function, but rather the multilayer feed-forward architecture itself which gives neural networks the potential of being universal approximators. The output units are always assumed to be linear.

Hornik (1991) proves density for any continuous bounded and nonconstant activation function, and also in other norms.

Theorem 2. . Let $m^i \in \mathbb{Z}_+^n, i = 1, ..., s$ and set $m = max|m^i| : i = 1, ..., s$. Assume $\sigma \in C^m(\mathbb{R})$ and σ is not a polynomial. Then $M(\sigma)$ is dense in $C^{m^1,...,m^s}(\mathbb{R}^n)$.

Theorem 3. (reformulation of Theorem 2.) . Let $m^i \in \mathbb{Z}_+^n, i = 1, ..., s$ and set $m = max|m^i| : i = 1, ..., s$. Assume $\sigma \in C^m(\mathbb{R})$ andσ is not a polynomial.

Then the space of single hidden layer neural nets

$$M(\sigma) := span\sigma(wx + b) : w \in \mathbb{R}^d, b \in \mathbb{R}$$

is dense in $C^{m^1,...,m^s}(\mathbb{R}^d) := \cap_{m=1}^s C^m(\mathbb{R}^d)$.

According to Pinkus in Acta Numerica, 1999:

"This density question was first considered by Hornik, Stinchcombe and White (1990). They showed that, if $\sigma^{(m)} \in C(\mathbb{R}^n) \cap L^1(\mathbb{R})$ then $M(\sigma)$is dense in $C^m(\mathbb{R})$. Subsequently Hornik (1991) generalized this to $\sigma \in C^m(\mathbb{R})$ which is bounded, but not the constant function. Hornik uses a functional analytic method of proof. With suitable modifications his method of proof can be applied to prove Theorem 2. Itô (1993) reproves Hornik's result, but for $\sigma \in C^{\infty(\mathbb{R})}$ which is not a polynomial. His method of proof is different. [...] This approach is very similar to the approach taken in Li (1996) where Theorem 2. can effectively be found. Other papers concerned with this problem are Cardaliaguet and Euvrard (1992), Gallant and White (1992), Itô (1994 b), Mhaskar and Micchelli (1995) and Attali and Pages (1997). Some of these papers contain generalizations to density in other norms, and related questions."

———-

DEEPXDE

DeepXDE is a deep learning library on top of TensorFlow. It was developed in Brown University by Lu Lu, Xuhui Meng , Zhiping Mao and George Em Karniadakis. It is an application of physics-informed neural networks, and it is designed to serve both as an education tool to be used in the classroom as well as a research tool for solving problems in computational science and engineering.

DeepXDE can be used to solve multi-physics problems, and supports complex-geometry domains based on the technique of constructive solid geometry (CSG), therefore avoiding tedious and time-consuming computational geometry tasks. By using DeepXDE, time-dependent PDEs can be solved as easily as steady states by only defining the initial conditions. In addition to the main workflow of DeepXDE, users can readily monitor and modify the solution process via callback functions, e.g., monitoring the Fourier spectrum of the neural network solution, which can reveal the learning mode of the NN. Last but not least, DeepXDE is designed to make the user code stay compact and manageable, resembling closely the mathematical formulation.

And while the task of using DeepXDE requires only minimal knowledge, we can't introduce it without it providing the important algorithms and functions on which it is based upon.

4.1 Automatic Differentiation

4.1.1 Definition and comparison with backpropagation

The neural networks used in DeepXDE use an effective but generally underused algorithm: automatic differentiation.

Automatic differentiation (AD), also called algorithmic differentiation or simply "auto-diff", is a family of techniques similar to but more general than backpropagation for efficiently and accurately evaluating derivatives of numeric functions expressed as computer programs. It is particularly useful for creating and training complex deep learning models without needing to compute derivatives manually for optimization.

While the backpropagation algorithm is very easy to understand and implement, hence its very common use during deep learning processes, it's still in some cases bad for memory use and schedule optimization. That's why for this specific task of solving differential equations, AD was seen as the better option, because its strong points are being able to generate gradient computation to an entire computation graph and thus being great for system optimization.

In general, bakpropagation refers to the whole process of training an artificial neural network using multiple backpropagation steps, each of which computes gradients and uses them to perform a Gradient Descent step. In contrast, automatic differentiation is simply a technique used to compute gradients efficiently and it happens to be used by backpropagation.

4.1.2 The process

The method uses symbolic rules for differentiation, which are more accurate than finite difference approximations. Unlike a purely symbolic approach, automatic differentiation evaluates expressions numerically early in the computations, rather than carrying out large symbolic computations. In other words, automatic differentiation evaluates derivatives at particular numeric values; it does not construct symbolic expressions for derivatives.

- Forward mode evaluates a numerical derivative by performing elementary derivative operations concurrently with the operations of evaluating the function itself. As detailed in the next section, the software performs these computations on a computational graph.
- Reverse mode automatic differentiation uses an extension of the forward mode computational graph to enable the computation of a gradient by a reverse traversal of the graph.

Forward mode

Consider the problem of evaluating this function and its gradient:

$$f(x_1, x_2) = x_1 exp(-\frac{1}{2}(x_1^2 + x_2^2))$$

Automatic differentiation works at particular points. In this case, take $x_1 = 2, x_2 = \frac{1}{2}$
The following computational graph encodes the calculation of the function f(x):

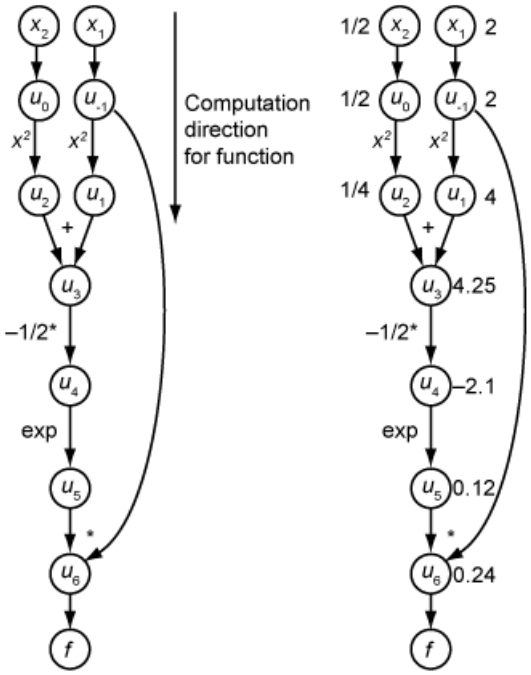

FIGURE 4.1: Computational graph of AD

To compute the gradient of $f(x)$ using forward mode, we compute the same graph in the same direction, but modify the computation based on the elementary rules of differentiation. To further simplify the calculation, we fill in the value of the derivative of each subexpression u_i as we go.

To compute the entire gradient, we must traverse the graph twice, once for the partial derivative with respect to each independent variable. Each subexpression in the chain rule has a numeric value, so the entire expression has the same sort of evaluation graph as the function itself.

The computation is a repeated application of the chain rule. In this example, the derivative of f with respect to x_1 expands to this expression:

$$\begin{aligned}\frac{df}{dx_1} &= \frac{du_6}{dx_1} \\ &= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial x_1} \\ &= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial x_1} \\ &= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial u_3}\frac{\partial u_3}{\partial x_1} \\ &= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial u_3}\frac{\partial u_3}{\partial u_1}\frac{\partial u_1}{\partial x_1}\end{aligned}$$

Let $\dot{u}_i$ represent the derivative of the expression u_i with respect to x_1. Using the evaluated values of the u_i from the function evaluation, we compute the partial derivative of f with respect to x_1 as shown in the following figure. Notice that all the values of the $\dot{u}_i$ become available as we traverse the graph from top to bottom.

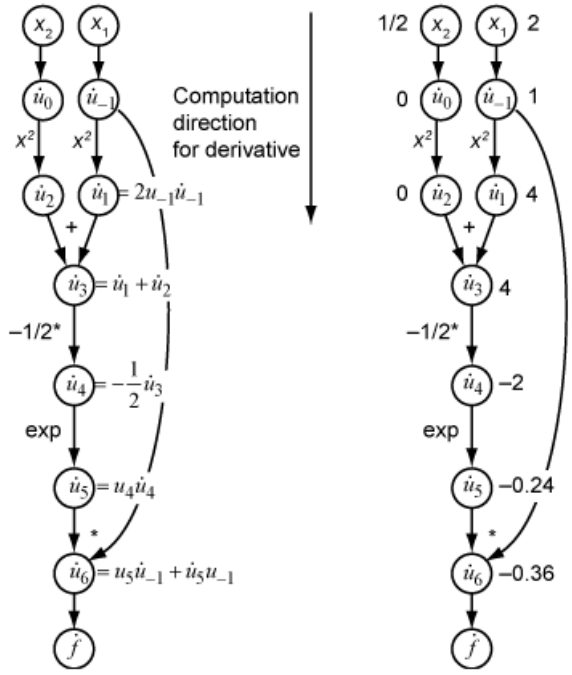

FIGURE 4.2: Graph for partial derivative wrt x_1

To compute the partial derivative with respect to x_2, we traverse a similar computational graph. Therefore, when we compute the gradient of the function, the number of graph traversals is the same as the number of variables. This process is too slow for typical deep learning applications, which have thousands or millions of variables.

Reverse mode

Reverse mode uses one forward traversal of a computational graph to set up the trace (which is basically the learning model of Forward Mode). Then it computes the entire gradient of the function in one traversal of the graph in the opposite direction. For deep learning applications, this mode is far more efficient.

The theory behind reverse mode is also based on the chain rule, along with associated adjoint variables denoted with an overbar. The adjoint variable for ui is

$$\overline{u_i} = \frac{\partial f}{\partial u_i}$$

In terms of the computational graph, each outgoing arrow from a variable contributes to the corresponding adjoint variable by its term in the chain rule. For example, the variable u_{-1} has outgoing arrows to two variables, u_1 and u_6. The graph has the associated equation:

$$\begin{aligned}\frac{\partial f}{\partial u_{-1}} &= \frac{\partial f}{\partial u_1}\frac{\partial u_1}{\partial u_{-1}} + \frac{\partial f}{\partial u_6}\frac{\partial u_6}{\partial u_{-1}} \\ &= \bar{u_1}\frac{\partial u_1}{\partial u_{-1}} + \bar{u_6}\frac{\partial u_6}{\partial u_{-1}}\end{aligned}$$

In this calculation, recalling that $u_1 = u_{-1}^2$ and $u_6 = u_5 u_{-1}$, we obtain

$$\overline{u}_{-1} = \bar{u_1} 2u_{-1} + \bar{u_6} u_5$$

During the forward traversal of the graph, the software calculates the intermediate variables u_i. During the reverse traversal, starting from the seed value $\overline{u}_6 = \dfrac{\partial f}{\partial f} = 1$, the reverse mode computation obtains the adjoint values for all variables. Therefore, the reverse mode computes the gradient in just one computation, saving a great deal of time compared to forward mode.

The following figure shows the computation of the gradient in reverse mode for $f(x_1, x_2)$

Again, the computation takes $x_1 = 2$, $x_2 = 1/2$. The reverse mode computation relies on the u_i values that are obtained during the computation of the function in the original computational graph. In the right portion of the figure, the computed values of the adjoint variables appear next to the adjoint variable names, using the formulas from the left portion of the figure.

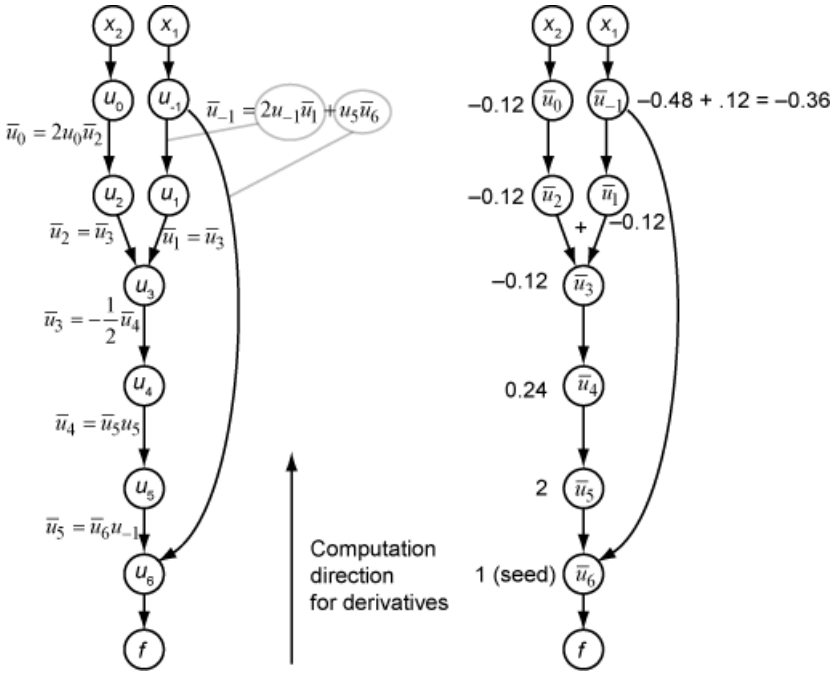

FIGURE 4.3: Reverse mode of AD

The final gradient values appear as $\overline{u}_0 = \dfrac{\partial f}{\partial u_0} = \dfrac{\partial f}{\partial x_2}$ and $\overline{u}_{-1} = \dfrac{\partial f}{\partial u_{-1}} = \dfrac{\partial f}{\partial x_1}$

As seen by this process, automatic differentiation only requires one forward pass and one backward pass to pass to compute all the partial derivatives.

In contrast, using finite differences to computing each partial derivative $\dfrac{\partial y}{\partial x_1}$ requires two function valuations $y(x_1, ..., x_i, ..., x_{d_{in}})$ and $y(x_1, ..., x_i + \Delta x_i, ..., x_{d_{in}})$ for some small number Δx_i, and thus in total $d_{in} + 1$ forward passes are required to evaluate all the partial derivatives. Hence, AD is much more efficient than finite difference when the input dimension is high. To compute nth-order derivatives, AD can be applied recursively n times. However, this nested

approach may lead to inefficiency and numerical instability, and hence other methods, e.g., Taylor-Mode AD, have been developed for this purpose. Finally we note that with AD we differentiate the NN and therefore we can deal with noisy data.

4.2 Physics-Informed Neural Networks (PINNs)

4.2.1 The concept of PINNs

"At first sight, the task of training a deep learning algorithm to accurately identify a nonlinear map from a few – potentially very high-dimensional – input and output data pairs seems at best naive. Coming to our rescue, for many cases pertaining to the modeling of physical and biological systems, there exists a vast amount of prior knowledge that is currently not being uti- lized in modern machine learning practice. Let it be the principled physical laws that govern the time-dependent dynamics of a system, or some empir- ically validated rules or other domain expertise, this prior information can act as a regularization agent that constrains the space of admissible solutions to a manageable size. [...] We exploit recent developments in automatic differenti- ation – one of the most useful but perhaps under-utilized techniques in scientific computing – to differentiate neural networks with respect to their input coordinates and model parameters to obtain physics-informed neural networks. Such neural networks are constrained to respect any symmetries, invariances, or conservation principles originating from the physical laws that govern the observed data, as modeled by general time-dependent and non- linear partial differential equations. This simple yet powerful construction allows us to tackle a wide range of problems in computational science and in- troduces a potentially transformative technology leading to the development of new data-efficient and physics-informed learning machines, new classes of numerical solvers for partial differential equations, as well as new data-driven approaches for model inversion and systems identification." ***-M. Raissi, P. Perdikaris, G.E. Karniadakis in Journal of Computational Physics, Feb.2019 issue.***

DeepXDE is simply an application of the above described physics-informed neural networks (PINNs).

They have coined them as a machine learning approach for predictive modeling of physical systems. Similar approaches employ machine learning algorithms like support vector machines, random forests, Gaussian processes, and feed-forward/convolutional/recurrent neural networks merely as **black-box tools**.

PINNs stand out because they went one step further by revisiting the construction of "custom" activation and loss functions that are tailored to the underlying differential operator. This allowed them to "open the black-box" by understanding and appreciating the key role played by automatic differentiation within the deep learning field.

Automatic differentiation in general, and the back-propagation algorithm in particular, is currently the dominant approach for training deep models by taking their derivatives with respect to the parameters (e.g., weights and biases) of the models.

PINNs use the exact same automatic differentiation techniques, employed by the deep learning community, to physics-inform neural networks by taking their derivatives with respect to their input coordinates (i.e., space and time) where the physics is described by partial differential equations. It was observed that this structured approach introduces a regularization mechanism that allows the use of relatively simple feed-forward neural network architectures and train them with small amounts of data.

4.2.2 PINNs for solving partial differential equations

Training a PINN to solve differential equations is no different that the learning process we described in the previous chapter.

To explain the procedure, consider the following PDE parameterized by λ for the solution $u(x)$ with $x = (x_1, ..., x_d)$ defined on a domain $\Omega \in \mathbb{R}^d$:

$$f(x; \frac{\partial u}{\partial x_1}, \dots, \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, ..., \frac{\partial^2 u}{\partial x_1 \partial x_d}; ...; \lambda) = 0, \quad x \in \Omega$$

with suitable boundary conditions

$$B(u, x) = 0 \text{ on } \partial\Omega$$

Step 1: Constructing the neural network.

We construct a neural network $\hat{u}(x;\theta)$ as a surrogate of the solution u(x), which takes the input x and outputs a vector with the same dimension as u.

Here, $\theta = (W^l, b^l)\ 1 \leq l \leq L$ is the set of all weight matrices and bias vectors in the neural network û. One advantage of PINNs by choosing neural networks as the surrogate of u is that we can take the derivatives of û with respect to its input x by applying the chain rule for differentiating compositions of functions using the automatic differentiation (AD), which is conveniently integrated in machine learning packages, such as TensorFlow and PyTorch.

Step 2: Specifying the two training sets for the equation and boundary/initial conditions.

The neural network $\hat{u}$ should be restricted to satisfy the physics imposed by the PDE and boundary conditions. In practice, $\hat{u}$ is restricted on randomly distributed points (i.e scattered points) on the domain: the training data $\mathcal{T} = x_1, x_2, ..., x_{|\mathcal{T}|}$ of size $|\mathcal{T}|$.

$\mathcal{T}$ is comprised of two sets: $\mathcal{T}_f \subset \Omega$ (the points on the domain) and $\mathcal{T}_b \subset \partial\Omega$ (the points on the boundary).

$\mathcal{T}_f$ and $\mathcal{T}_b$ are referred to as the sets of "residual points".

The authors offer three possible strategies for choosing residual points:

a) We can specify the residual points at the beginning of training, which could be grid points on a lattice or random points, and never change them during the training process.

b) In each optimization iteration, we could select randomly different residual points.

c) Residual-based adaptive refinement (RAR): Improve the location of the residual points adaptively during training. This is especially efficient for certain PDEs that exhibit solutions with steep gradients. To overcome this challenge, the authors propose a residual-based adaptive refinement (RAR) method to improve the distribution of residual points during the training process, conceptually similar to FEM refinement methods. The idea of RAR is to add more residual points in the locations where the PDE residual

$$\|f(x; \frac{\partial\hat{u}}{\partial x_1}, \ldots, \frac{\partial\hat{u}}{\partial x_d}; \frac{\partial^2\hat{u}}{\partial x_1\partial x_1}, ..., \frac{\partial^2\hat{u}}{\partial x_1\partial x_d}; ...; \lambda)\|$$

is large, to estimate (by Monte Carlo integration)the mean residual

$$\varepsilon_r = \frac{1}{V}\int_\Omega \|f(x; \frac{\partial\hat{u}}{\partial x_1}, \ldots, \frac{\partial\hat{u}}{\partial x_d}; \frac{\partial^2\hat{u}}{\partial x_1\partial x_1}, ..., \frac{\partial^2\hat{u}}{\partial x_1\partial x_d}; ...; \lambda)\|dx$$

with V being the volume of Ω. We then repeat adding points until ε_r is smaller than a threshold ε_0.

When the number of residual points required is very large, e.g., in multiscale problems, it is computationally expensive to calculate the loss and gradient in every iteration. Instead of using all residual points, we can split the residual points into small batches, and in each iteration we only use one batch to calculate the loss and update model parameters; this is the so-called "mini-batch" gradient descent. The aforementioned strategy (b), i.e., re-sampling in each step, is a special case of mini-batch gradient descent by choosing $\mathcal{T} = \Omega$ with $|\mathcal{T}| = \infty$.

Step 3: Compute the difference between the neural network and the constraints

This step is about specifying a loss function by summing the weighted L^2 norm of both the PDE and boundary condition residuals. This function is defined as:

$$\mathcal{L}(\theta, \mathcal{T}) = w_f \mathcal{L}_f(\theta, \mathcal{T}_f) + w_b \mathcal{L}_b(\theta, \mathcal{T}_b)$$

where w_f and w_b are the weights and

$$\mathcal{L}_f(\theta, \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{x \in \mathcal{T}_f} \|f(x; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, ..., \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; ...; \lambda)\|_2^2$$

$$\mathcal{L}_b(\theta, \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{x \in \mathcal{T}_b} \|B(\hat{u}, x)\|_2^2$$

The derivatives involved in the loss are handled via automatic differentiation.

Soft constraints of boundary/initial conditions are enforced through the loss $\mathcal{L}_b$. This approach can be used for complex domains and any type of boundary conditions. On the other hand, it is possible to enforce hard constraints for simple cases. For example, when the boundary condition is $u(0) = u(1) = 0$ with $\Omega = [0, 1]$, we can simply choose the surrogate model as $\hat{u}(x) = x(x-1)\mathcal{N}(x)$ to satisfy the boundary condition automatically, where $\mathcal{N}(x)$. is a neural network.

Step 4: Training the neural network to find the best parameters by minimizing the loss function

.

This is the training. Considering the fact that the loss is highly nonlinear and non-convex with respect to θ, the loss function is usually minimized by gradient-based optimizers, such as gradient descent, Adam, and L-BFGS. It was remarked that, for smooth PDE solutions L-BFGS can find a good solution with less iterations than Adam, because L-BFGS uses second- order derivatives of the loss function, while Adam only relies on first-order derivatives. However, for stiff solutions L-BFGS is more likely to be stuck at a bad local minimum.

The required number of iterations highly depends on the problem (e.g., the smoothness of the solution), and to check whether the network converges or not, we can monitor the loss function or the PDE residual using callback functions. PINNs can help achieving acceleration of training by using adaptive activation function that may remove bad local minima. Unlike traditional numerical methods, with PINNs there is no guarantee of unique solutions, because PINN solutions are obtained by solving non-convex optimization problems, which in general do not have a unique solution.

It is also noted that PINNs may converge to different solutions from different network initial values, and thus a common strategy is to train PINNs from random initialization for a few times (e.g., 10 independent runs) and choose the network with the smallest training loss as the final solution.

The procedure has been nicely described in this graphic of a PINN for solving the diffusion equation $\frac{\partial u}{\partial t} = \lambda \frac{\partial^2 u}{\partial x^2}$ with mixed boundary conditions (BC) $u(x,t) = g_D(x,t)$ on $\Gamma_D \subset \partial\Omega$ and $\frac{\partial u}{\partial n}(x,t) = g_R(u,x,t)$ on $\Gamma_R \subset \partial\Omega$. The initial condition (IC) is treated as a special type of boundary conditions. $\mathcal{T}_f$ and $\mathcal{T}_b$ denote the two sets of residual points for the equation and BC/IC.

4.3 Adam optimization

Most of the applications work very well using the "Adam" optimization algorithm.

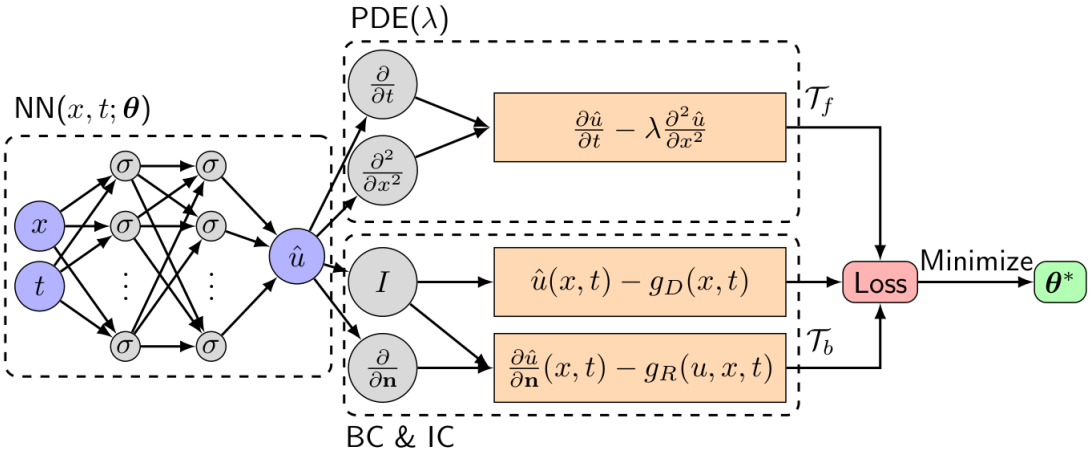

FIGURE 4.4: Schematic of a PINN for solving a PDE with BC

Adaptive Moment Estimation (Adam) is a gradient descent method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of *past squared* gradients v_t, it also keeps an exponentially decaying average of *past gradients* m_t, similar to momentum:

$$m_t = \beta_1 m_t - 1 + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_t - 1 + (1 - \beta_2) g_t^2$$

with $g_t = \nabla_\theta J(\theta_t)$ the gradient of the objective function w.r.t. to the parameter θ_t at time step t, and β_i being the decay rates.

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. Since m_t and v_t are initialized as vectors of 0s, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

They then use these to update the parameters, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

The authors propose default values of 0.9 for β_1 , 0.999 for β_2 , and 10^{-8} for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

4.4 Usage of DeepXDE

4.4.1 Installation

DeepXDE requires TensorFlow to be installed. Both TensorFlow 1 and TensorFlow 2 can be used as the DeepXDE backend, but TensorFlow 1 is recommended, because according to the developer's tests TensorFlow 2 is 2x-3x slower than TensorFlow 1.

- Install the stable version with `pip` :

```
$ pip install deepxde
```

- Install the stable version with `conda`:

```
$ conda install -c conda-forge deepxde
```

- For developers, clone the folder to your local machine and put it along with your project scripts:

```
$ git clone https://github.com/lululxvi/deepxde.git
```

- **Dependencies**:
  - Matplotlib
  - NumPy
  - SALib
  - scikit-learn
  - SciPy
  - TensorFlow>=1.14.0

4.4.2 Procedure

Solving differential equations in DeepXDE consists of simply specifying the problem using the built-in modules, including computational domain (geometry and time), PDE equations, boundary/initial conditions, constraints, training data, neural network architecture, and training hyperparameters.

The workflow is as follows:

Algorithm 2: Usage of DeepXDE to solve a differential equation

1. Specify the computational domain using the `geometry` module.
2. Specify the PDE using the grammar of `TensorFlow` .
3. Specify the boundary and initial conditions.
4. Combine the geometry, PDE and boundary/initial conditions together into `data.PDE` or `data.TimePDE` for time-independent problems or for time- dependent problems, respectively. To specify training data, we can either set the specific point locations, or only set the number of points and then DeepXDE will sample the required number of points on a grid or randomly.
5. Construct a neural network using the `maps` module.
6. Define a `Model` by combining the PDE problem in Step 4 and the neural net in Step 5.
7. Call `Model.compile` to set the optimization hyperparameters, such as optimizer and learning rate. The weights in the loss function ($\mathcal{L}$) can be set here by `loss_weights`
8. Call `Model.train` to train the network from random initialization or a pre- trained model using the argument `model_restore_path` . It is extremely flexible to monitor and modify the training behavior using `callbacks`.
9. Call `Model.predict` to predict the PDE solution at different locations.

To better accentuate the above procedures, here's the example of using DeepXDE to solve the same problem that we used in the end of Chapter 2:

$$(1): \begin{cases} y' + 2y = x^3 e^{-2x}, \\ y(0) = 1 \end{cases}$$

Which has an exact solution: $y(x) = \frac{1}{4}e^{-2x}(x^4 + 4)$

1.Geometry

In DeepXDE, the built-in primitive geometries include `interval`, `triangle`, `rectangle`, `polygon`, `disk`, `cuboid` and `sphere`. Other geometries can be constructed from these primitive geometries using constructive solid geometry (CSG) through the three boolean operations: union (`|`), difference (-) and intersection (&).

Implementation for (1):

```
geom = dde.geometry.TimeDomain(0,10)
```

We the `geometry.TimeDomain` module to define an interval, in our case it's $[0, 10]$

2.Defining the differential problem

We use TensorFlow's `tf.gradients` module, and return the implicit form of the equation :

```
def diffeq(t, y):
        dy_t = tf.gradients(y, t)[0]
        return dy_t+2*y-(t**3)*tf.exp(-2*t)
```

Note that we must not mix the symbolic types, that's why we use `tf.` modules to define particular mathematical functions.

3.Specifying the boundary/initial conditions

DeepXDE supports four standard boundary conditions, including Dirichlet (`DirichletBC`), Neumann (`NeumannBC`), Robin (`RobinBC`), and periodic (`PeriodicBC`), and a more general BC can be defined using `OperatorBC`. The initial condition can be defined using `IC`.

So first we use the `boundary` function, which returns a boolean function that "confirms" that we're on the initial data (or boundary, it depends on the problem at hand)

```
def boundary(t, on_initial):
        return on_initial
```

Then we define the initial condition $y(0) = 1$ using the `dde.IC` module,

```
ic = dde.IC(geom, lambda t: 1 * np.ones(t.shape), boundary)
```

for which the arguments work this way: `boundary` is used to specify that on the initial point of `geom` y satisfies `lambda t: 1 * np.ones(t.shape)`. We use a `lamda` function that returns `np.ones(t.shape)` multiplied by whatever scalar value we need, in this case 1.

- `lambda` function: In Python, a lambda function is a single-line function declared with no name, which can have any number of arguments, but it can only have one expression. Such a function is capable of behaving similarly to a regular function declared using the Python's def keyword. Its use in DeepXDE allows us the flexibilty is provides. We can consider it is a "cupholder" or a "passepartout".
- `np.ones(t.shape)`: Return an array of ones with shape and type of input, which we multiply by the scalar we would like to use as an IC. Reminder that TensorFLow variables are by default tensors, that's why DeepXDE uses the NumPy library because it helps with the value affectations. There many other NumPy functions that we will see later on.

4.The data module

This module is basically the problem, in which we regroup all of the above definitions.

```
data = dde.data.PDE(geom,diffeq,ic, num_domain=16,
                        num_boundary=1, solution=func, num_test=100)
```

In addition to `ic`, `geom`, `diffeq`, we also added a comparison function `func`, which is the exact solution to the problem, and added training points such as: `num_domain` is the number of training points, `num_boundary` the number of boundary condition points on the geometry boundary and `num_test` the number of test points. By default, train points are sampled randomly, and test points are sampled uniformly. But sometimes it is hard to sample uniformly. There are two cases:

• Sometimes it is impossible to sample uniformly, e.g., there isn't any way to sample uniformly in a rectangle. In these cases, we switch to random points.

• Sometimes it is impossible to sample exactly the required number of points. For example, we have a square $[0, 1]^2$, and want to sample 50 points inside the domain, but $\sqrt{50}$ is not an integer. Then we compute $ceil(sqrt50) = 8$, and sample $8^2 = 64$ points.

Obviously the higher the numbers the more accurate the prediction is, but the longer it would take to run an epoch. So for a relatively easy problem, such as an ODE for example, the numbers wouldn't make that much of difference.

What DeepXDE does is:

Constructing the neural network

Contrary to the code we've seen in the previous chapter, the construction of a neural network is already pre-defined in the library, and it is done by using the `maps` module:

```
net = dde.maps.FNN([1] + [50] * 3 + [1], "tanh", "Glorot uniform")
```

We just defined `net`, a feed-forward neural network with 1 inputs, 3 hidden layers of size 50 and 1 output. The activation function is *tanh* and we use Glorot uniform which is a kernel initializer that generates random weights.

Defining the model and compiling it

Like the graph in Figure 4.4, we have both our neural network and our problem, and thus we have defined our model.

```
model = dde.Model(data, net)
```

Next is defining the hyperparamaters of the learning process, in this case it's the optimizer `adam` and a learning rate of 0.001. And since we already have an exact solution, we can use `metrics=["l2 relative error"]` to calculate the error on the residuals.

```
model.compile("adam", lr=0.001,metrics=["l2 relative error"])
```

And finally we train the model to predict the solution at different locations.

```
losshistory, train_state = model.train(epochs=10000)
```

We have instructed the library to both train the model and to save the taining data in `train_state`, and also save the loss history in `losshistory`

```
dde.saveplot(losshistory, train_state, issave=True, isplot=True)
```

We can use the `saveplot` module to draw a plot of both the solution and the loss history. Since we provided an exact solution, the library automatically plots the residuals history.

Here's the final output:

```
Best model at step 10000:
  train loss: 8.80e-07
  test loss: 2.07e-04
  test metric: [2.23e-02]
'train' took 16.327880 s
```

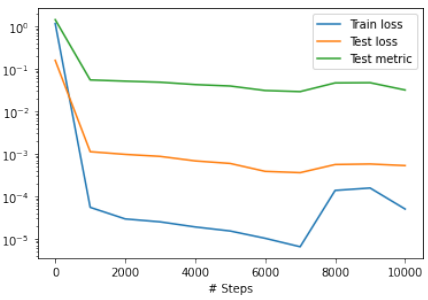

FIGURE 4.5: Loss history

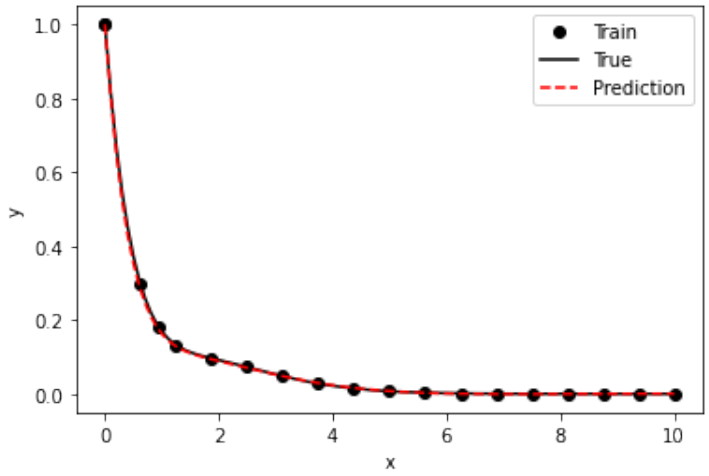

FIGURE 4.6: Predicted solution, compared to the exact solution. Note the high accuracy.

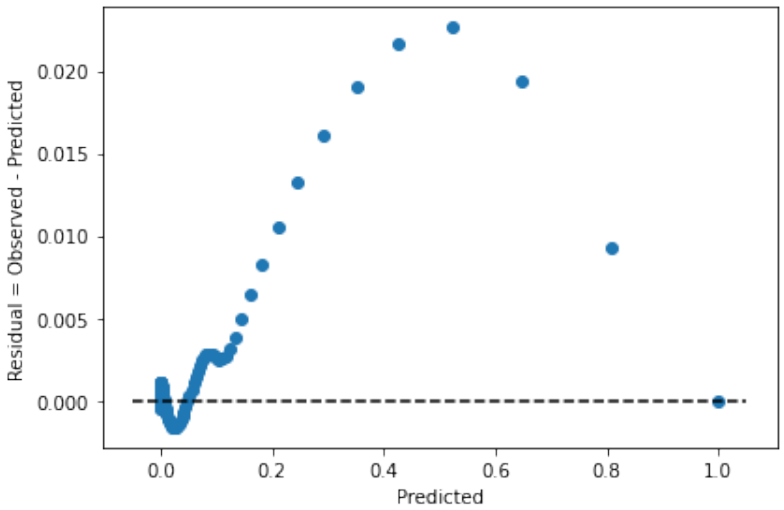

FIGURE 4.7: Residuals

Note that every module is highly customizable with many different arguments and therefore usage can vary depending on the problem at hand. In the next chapter we cover the most important and essential functionalities.

Applications

Like we emphasized throughout this paper, using DeepXDE is only a matter of defining the problem using TensorFlow and thus Python and NumPy syntax and functions, and knowing how to choose the hyperparameters for the learning process, although they work similarly well for the majority of problems. We will give further details on any new function used for the definitions.

Since we're using a Google Colaboratory notebook, we start with installing the library using:

```
$ pip install deepxde
```

As of today, it installs DeepXDE 0.7.0 and its dependencies.

Next we import the essential libraries :

```
from __future__ import absolute_import
from __future__ import division from __future__ import print_function

import numpy as np
import tensorflow as tf
import deepxde as dde
```

And that's it, we can now defined our problem and our training network inside the main function.

5.1 ODE system

Consider the following system:

$$\begin{cases} \frac{du}{dt} = -uv^2 \\ \frac{dv}{dt} = uv^2 \\ u(0) = 1 \\ v(0) = 2 \end{cases}$$

This is a rather simple case because the initial conditions for ODEs are pretty straightforward.

- **Defining the system:**

```
def system(t, y):
        u, v = y[:, 0:1], y[:, 1:]
        du_t = tf.gradients(u, t)[0]
        dv_t = tf.gradients(v, t)[0]
        return [du_t + u*v*v, dv_t - u*v*v]
```

- **Defining the initial conditions:**

```
def boundary(_, on_initial):
        return on_initial


geom = dde.geometry.TimeDomain(0,1)
ic1 = dde.IC(geom, lambda t: 1 * np.ones(t.shape), boundary, component=0)
ic2 = dde.IC(geom, lambda t: 2 * np.ones(t.shape), boundary, component=1)
```

As you can see, we defined the initial conditions using the IC module. Its arguments are `geom`, the [0,1] interval (or time domain), .

Now to define u(0)=1 and v(0)=2, we first need to specify which function we're on, that's why we the function's component index: 0 being for u and 1 being for v, as we defined them in `ode_system`.

Next the values 1 and 2. For that we used a `lamda` function that returns `np.ones(t.shape)` multiplied by whatever scalar value we need.

- **The neural network**

```
layer_size = [1] + [50] * 3 + [2]
activation = "tanh"
initializer = "Glorot uniform"
net = dde.maps.FNN(layer_size, activation, initializer)
```

This is very much like the example in the previous chapter, and requires no further explanation.

- **Combining the data, compiling the model and training the network**

```
data = dde.data.PDE(geom,system, [ic1, ic2], 35, 2, num_test=100)
model = dde.Model(data, net)
model.compile("adam", lr=0.001)
losshistory, train_state = model.train(epochs=20000)
```

Since our problem is but a simple ODE, we only used 35 training points, 2 boundary condition points and 100 test points. We use adam optimization, a learning rate of 0.001 and 20000 rounds of training.

- **The results :**

```
dde.saveplot(losshistory, train_state, issave=True, isplot=True)
```

Calling the module `saveplot` to plot both the loss history and the predicted outcome.

Here are the yielded results:

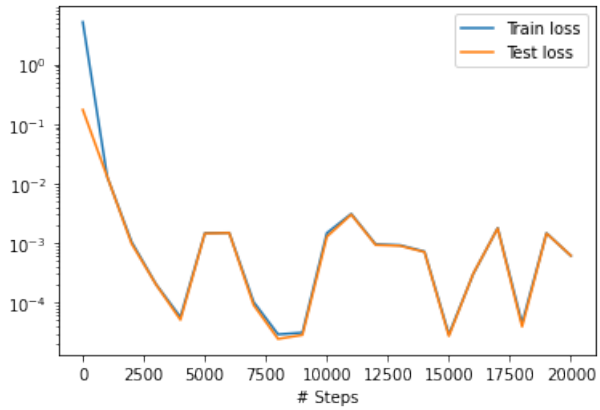

FIGURE 5.1: This is the loss history compared with the train history, and it shows a convergence in the first 1250 or so epochs

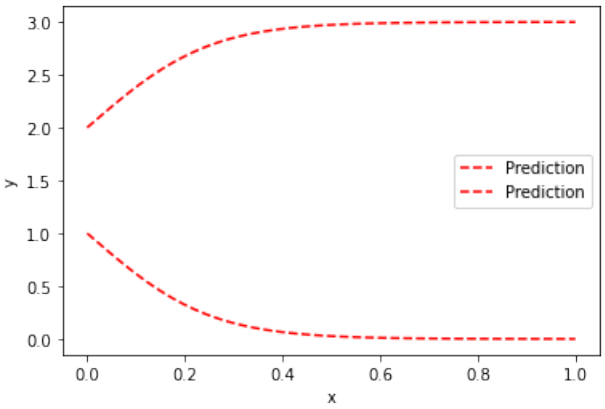

FIGURE 5.2: These are the solutions

5.2 Periodic solution

Let's consider the following differential equation:

$$\begin{cases} -\frac{du}{dt} = sin(t) - u^3 \\ u(0) = u(2\pi) \end{cases}$$

We will use a fixed point iteration to solve the differential equation.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf

import deepxde as dde

def main():
        u0=0.9
```

```
        epsilon=1e-8
        k=0
        err=1
        N=40
        T=np.array([[2*np.pi]])
        def equa(t, u):
                du_t = tf.gradients(u, t)[0]
                du_tt = tf.gradients(du_t, t)[0]
                return -du_t - tf.sin(t) + u**3

        def boundary(x, on_boundary):
                return on_boundary


        geom = dde.geometry.TimeDomain(0,2*np.pi)
        layer_size = [1] + [50] * 3 + [1]
        activation = "tanh"
        initializer = "Glorot uniform"
        net = dde.maps.FNN(layer_size, activation, initializer)



        while (err>epsilon) and (k<=40):
                ic1 = dde.IC(geom, lambda t: u0 * np.ones(t.shape), boundary)
                data = dde.data.PDE(geom,equa, ic1, num_domain=N, num_boundary=1)
                model = dde.Model(data, net)
                model.compile("adam", lr=0.001)
                losshistory, train_state =
                model.train(epochs=10000)
                err=(u0-model.predict(T))**2
                u0=model.predict(T)
                k=k+1


        dde.saveplot(losshistory, train_state, issave=True, isplot=True)

if __name__ == "__main__":
main()
```

We used a `while` loop to be able to use the fixed point iteration until convergence, with `model.predict(T)` The yielded results are:

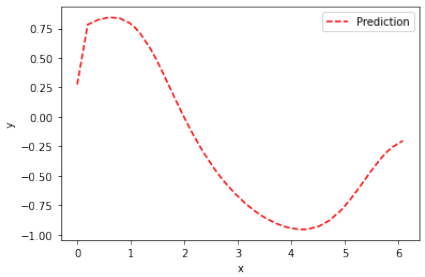

FIGURE 5.3: The solution, which we can see slightly satisfies the fixed point criterion, although the forced conditions on iterations (<40) affected the accuracy

5.3 Non linear equation

We will consider the following non-linear problem:

$$\frac{\partial^2 u}{\partial x^2} = 1 - u^3 \quad u(0) = u(1) = 0$$

```
def main():
        def ode_system(t, y):
                dy_t = tf.gradients(y, t)[0]
                dy_tt = tf.gradients(dy_t,t)[0]
                return dy_tt + 1 - y*y*y

        def boundary_l(t, on_boundary):
                return on_boundary and np.isclose(t[0], 0)

        def boundary_r(t, on_boundary):
                return on_boundary and np.isclose(t[0], 1)


        geom = dde.geometry.Interval(0, 1)
        bc1 = dde.DirichletBC(geom, lambda t: np.full_like(t,0), boundary_l)
        bc2 = dde.DirichletBC(geom, lambda t: np.full_like(t,0), boundary_r)
        data = dde.data.PDE(geom, ode_system, [bc1, bc2], 50, 2 , num_test=100)

        layer_size = [1] + [50] * 3 + [1]
        activation = "tanh"
        initializer = "Glorot uniform"
        net = dde.maps.FNN(layer_size, activation, initializer)

        model = dde.Model(data, net)
        model.compile("adam", lr=0.001)
        losshistory, train_state = model.train(epochs=40000)

        dde.saveplot(losshistory, train_state, issave=True, isplot=True)


if __name__ == "__main__":
main()
```

We mixed things up a little bit here, and used the Dirichlet boundary conditions on the 2 extremities of the interval, which still count as initial conditions.

- `lambda t: np.full_like(t,0)`: Returns a full array with the same shape and type as a given array, in this case: 0.

The results are:

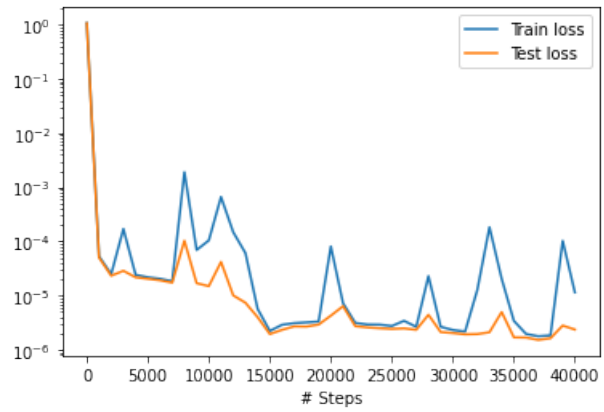

FIGURE 5.4: Very minimal loss

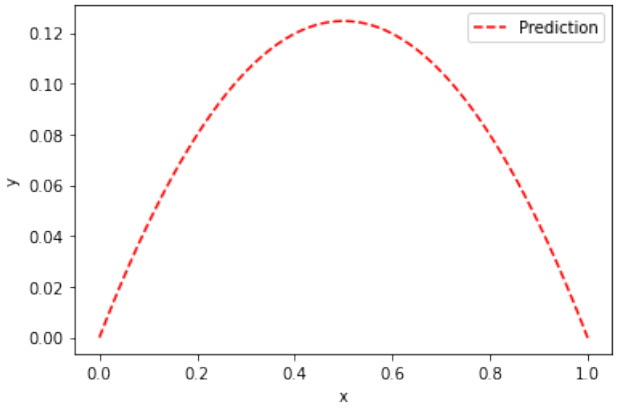

FIGURE 5.5: Predicted solution

5.4 2D equation on a rectangle

Consider the 2d equation:

$$-\frac{d^2u}{dx^2} - \frac{d^2u}{dy^2} = x^2 + y^2 \quad u(x,y) = 0 \text{ on } \backslash\Omega =]0,1[\times]0,1[$$

```
def main():
        def pde (x , y ) :
                dy_x = tf . gradients (y , x ) [0]
                dy_x , dy_y = dy_x [: , 0:1] , dy_x [: , 1:]
                dy_xx = tf . gradients ( dy_x , x ) [0][: , 0:1]
                dy_yy = tf . gradients ( dy_y , x ) [0][: , 1:]
                return dy_xx + dy_yy - x*x - y*y

        def boundary(_, on_boundary):
                return on_boundary
        def func (x) :
                return np.zeros([len(x),1])

        geom = dde.geometry.Rectangle([0,0],[1,1])
        bc = dde.DirichletBC(geom,func,boundary)
        data = dde.data.PDE(geom, pde , bc , num_domain =1200 , num_boundary =120)
        layer_size = [2] + [50] * 3 + [1]
        activation = "tanh"
```

```
        initializer = "Glorot uniform"
        net = dde.maps.FNN(layer_size, activation, initializer)

        model = dde.Model(data, net)
        model.compile("adam", lr=0.001)
        losshistory, train_state = model.train(epochs=20000)

        dde.saveplot(losshistory, train_state, issave=True, isplot=True)


if __name__ == "__main__":
main()
```

• Geometry: we're using the module `Rectangle` which takes as arguments the coordinates of the lowest point to the left and the highest point to the right; in our case they're [0,0] and [1,1].

Otherwise it's a pretty straightforward application.

The results:

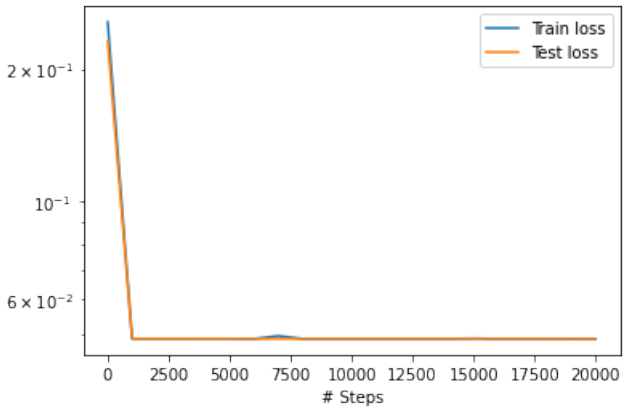

FIGURE 5.6: Loss history

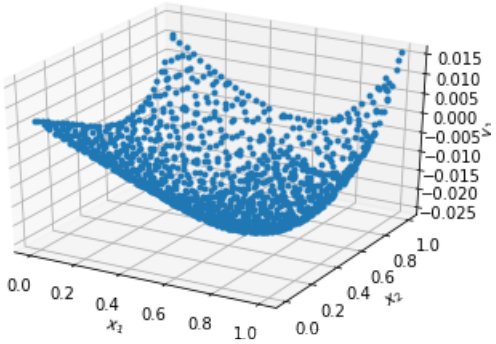

FIGURE 5.7: The predicted solution

5.5 Modeling of a real life phenomenon: COVID-19

The ongoing Covid-19 pandemic is a phenomenon that's affecting the world on many different aspects; many scientific institutions around the planet are allocating immense resources to better understand both the behaviour and the effects of this virus.

In this section, we present a mathematical model of the Covid-19 pandemic that highlights the ability and extreme importance of differential systems in the modeling of real life phenomena.

5.5.1 The problem's settings

The study we're referencing developed a Bats-Hosts-Reservoir- People (BHRP) transmission network model for simulating the potential transmission from the infection source (it's assumed to be bats) to the human infection. They subsequently simplified the model as Reservoir-People (RP) transmission network model, and R_0 was calculated based on the RP model to as- sess the transmissibility of the SARS-CoV-2.

Simply put, it's a compartment model that doesn't take into account the spatial variable, but instead only the temporal variable and time variation of the quantifiable entities that figure in the model.

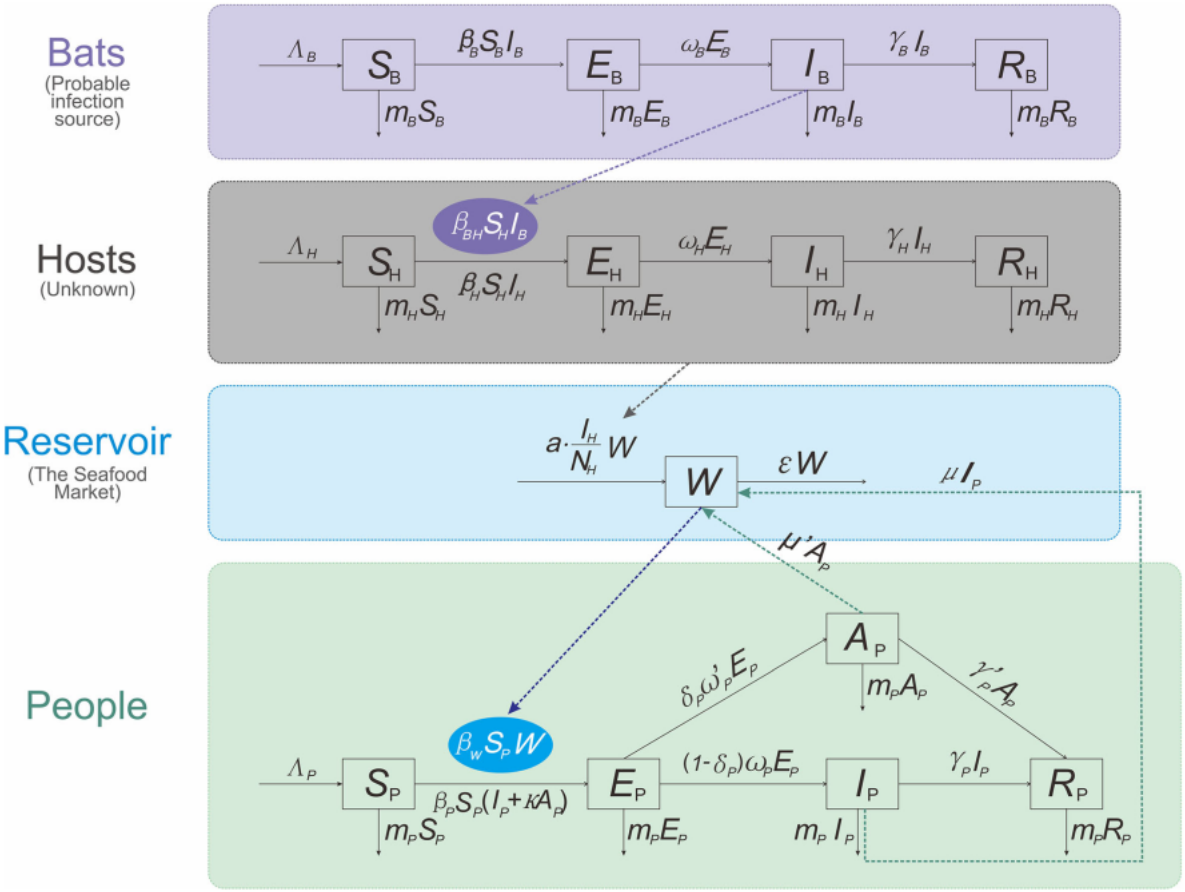

FIGURE 5.8: Flowchart describing the RBHP model

S_i: susceptible, E_i:exposed, I_i: infected, R_i:removed, n: birth rate, m: death rate, N_i: total number, $\Lambda_i = n_i \times N_i$, $1/\omega_i$: incubation period, $1/\gamma_i$: period of host infection, β_{ij}/β_i: transmission rate from i to j or in i, $1/\omega_i'$:period of latent incubation, $\{i, j\}$: compartment.

The flowchart describes the SARS-CoV-2 spread:

- The SARS-CoV-2 in reservoir (the seafood market) was denoted as W. It is assumed that the retail purchases rate of the hosts in the market was a, and that the prevalence of SARS-CoV-2 in the purchases was I_H/N_H , therefore, the rate of the SARS- CoV-2 in W imported form the hosts was $aWd\frac{I_H}{N_H}$.
- It is also assumed that symptomatic infected people and asymptomatic infected people could export the virus into W with the rate of μ_P and μ_P' , although this assumption might occur in a low probability.
- The virus in W will subsequently leave the W compartment at a rate of ϵW, where $1/\epsilon$ is the lifetime of the virus.

- Symptomatic infected people are noted as I_P, and asymptomatic infected people A_P, while removed people (R_P) includes recovered and death people.
- The proportion of asymptomatic infection was defined as δP . The S_P will be infected through sufficient contact with W and I_P. It is also assumed that the transmissibility of A_P was κ times that of I_P ,where $0 \leq \kappa \leq 1$

And finally the mathematical model that describes the RPH chart is as follows:

$$\textbf{(BHRP)}: \begin{cases} \frac{dS_B}{dt} = \Lambda_B - m_b S_b - \beta_B S_B I_B \\ \frac{dE_B}{dt} = \beta_B S_B I_B - \omega_B E_B - m_B E_B \\ \frac{dI_B}{dt} = \omega_B E_B - (\gamma_B + m_B) I_B \\ \frac{dR_B}{dt} = \gamma_B I_B - m_B R_B \\ \frac{dS_H}{dt} = \Lambda_H - m_H S_H - \beta_B H S_H I_B - \beta_H S_H I_H \\ \frac{dE_H}{dt} = \beta_{BH} S_H I_B + \beta_H S_H I_H - \omega_H E_H - m_H E_H \\ \frac{dI_H}{dt} = \omega_H E_H - (\gamma_H + m_H) I_H \\ \frac{dR_H}{dt} = \gamma_H I_H - m_H R_H \\ \frac{dS_p}{dt} = \Lambda_P - m_P S_P - \beta_P S_P (I_P + \kappa A_P) - \beta_W S_P W \\ \frac{dE_P}{dt} = \beta_P S_P (I_P + \kappa A_p) + \beta_W S_P W - (1 - \delta_p)\omega_P E_P - \delta_p \omega'_p E_P - m_P E_P \\ \frac{dI_P}{dt} = (1 - \delta_P)\omega_P E_P - (\gamma_P + m_P) I_P \\ \frac{dA_P}{dt} = \delta_p \omega'_P I_P - (\gamma'_P + m_P) A_P \\ \frac{dR_P}{dt} = \gamma_P I_P + \gamma'_P A_P - m_P R_P \\ \frac{dW}{dt} = aW \frac{I_H}{N_H} + \mu_P I_P + \mu'_P A_P - \epsilon W \end{cases}$$

Ignoring the transmission network of Bats-Host led to a simplified model.

During the outbreak period, the natural birth rate and death rate in the population was in a relative low level. However, people would commonly travel into and out from Wuhan City mainly due to the Chinese New Year holiday. Therefore, n_P and m_P refer to the rate of people traveling into Wuhan City and traveling out from Wuhan City, respectively. In the model, people and viruses have different dimen- sions. Normalization of different compartment criteria was subsequently done. In the normalization, parameter c refers to the relative shedding coefficient of A_P compared to I_P. The normalized RP model is changed as follows:

$$\textbf{(RP)} \begin{cases} \frac{ds_p}{dt} = n_p - m_p s_p - b_p s_p (i_p + \kappa a_p) - b_W s_p w \\ \frac{de_p}{dt} = b_p s_p (i_p + \kappa a_p) + b_W s_p w - (1 - \delta_p)\omega_p e_p - \delta_p \omega'_p e_p - m_p e_p \\ \frac{di_p}{dt} = (1 - \delta_p)\omega_p e_p - (\gamma_p + m_p) i_p \\ \frac{da_p}{dt} = \delta_p \omega'_p e_p - (\gamma'_p + m_p) a_p \\ \frac{dr_p}{dt} = \gamma_p i_p + \gamma'_p a_p - m_p r_p \\ \frac{dw}{dt} = \epsilon (i_p + c a_p - w) \end{cases}$$

Solving the system (RP) will eventually describe the behaviour of the different criteria with respect to time.

We will use DeepXDE to solve it.

$$\textbf{(RP)}: \begin{cases} \frac{ds_p}{dt} = n_p - m_p s_p - b_p s_p (i_p + \kappa a_p) - b_W s_p w \\ \frac{de_p}{dt} = b_p s_p (i_p + \kappa a_p) + b_W s_p w - (1 - \delta_p)\omega_p e_p - \delta_p \omega'_p e_p - m_p e_p \\ \frac{di_p}{dt} = (1 - \delta_p)\omega_p e_p - (\gamma_p + m_p) i_p \\ \frac{da_p}{dt} = \delta_p \omega'_p e_p - (\gamma'_p + m_p) a_p \\ \frac{dr_p}{dt} = \gamma_p i_p + \gamma'_p a_p - m_p r_p \\ \frac{dw}{dt} = \epsilon (i_p + c a_p - w) \end{cases}$$

Since we're rather interested in the behaviour of the different criteria, will use simplified scaled initial values on top of the projected parameter values provided by the study.

```
omegap=0.1923
gammap=0.1724
deltap=0.5
kappa=0.5
c=0.5
n=0.0018
mp=0.0018
eps=0.1
bp=3.58
bw=2.30

def main():
        def ode_system(t, y):
                sp, ep, ip, ap, rp, w = y[:, 0:1], y[:, 1:2], y[:, 2:3], y[:, 3:4],
                y[:, 4:5], y[:, 5:]
                dsp_t = tf.gradients(sp, t)[0]
                dep_t = tf.gradients(ep, t)[0]
                dip_t = tf.gradients(ip, t)[0]
                dap_t = tf.gradients(ap, t)[0]
                drp_t = tf.gradients(rp, t)[0]
                dw_t = tf.gradients(w, t)[0]

                return [
                dsp_t - n + mp * sp + bp * sp * ( ip + kappa * ap ) + bw * sp * w,
                dep_t - bp * sp * ( ip + kappa * ap ) - bw * sp * w + ( 1 - deltap )
                    * omegap * ep + deltap * omegap * ep + mp * ep,
                dip_t - ( 1 - deltap ) * omegap * ep + ( gammap + mp ) * ip,
                dap_t - deltap * omegap * ep + ( gammap + mp ) * ap,
                drp_t - gammap * ip - gammap * ap + mp * rp,
                dw_t - eps * ( ip + c * ap - w )
                ]

        def boundary(_, on_initial):
                return on_initial

```

```
        geom = dde.geometry.TimeDomain(0, 10)
        ic1 = dde.IC(geom, lambda X: 10 * np.ones(X.shape), boundary, component=0)
        ic2 = dde.IC(geom, lambda X: 8 * np.ones(X.shape), boundary, component=1)
        ic3 = dde.IC(geom, lambda X: 5 * np.ones(X.shape), boundary, component=2)
        ic4 = dde.IC(geom, lambda X: 3 * np.ones(X.shape), boundary, component=3)
        ic5 = dde.IC(geom, lambda X: 0.2 * np.ones(X.shape), boundary, component=4)
        ic6 = dde.IC(geom, lambda X: 1 * np.ones(X.shape), boundary, component=5)

        data = dde.data.PDE(
        geom, ode_system, [ic1, ic2, ic3, ic4, ic5, ic6], 500, 6, num_test=100
        )

        layer_size = [1] + [50] * 3 + [6]
        activation = "tanh"
        initializer = "Glorot uniform"
        net = dde.maps.FNN(layer_size, activation, initializer)

        model = dde.Model(data, net)
        model.compile("adam", lr=0.001)
        losshistory, train_state = model.train(epochs=30000)

        dde.saveplot(losshistory, train_state, issave=True, isplot=True)


if __name__ == "__main__":
        main()
```

The code is no different than the previous ODE systems we've treated, but it highlights how we can use DeepXDE for multiple solutions.

Note the declarations

```
sp, ep, ip, ap, rp, w =
y[:, 0:1], y[:, 1:2], y[:, 2:3], y[:, 3:4],
y[:, 4:5], y[:, 5:]
```

It shows how our function are just array components of the y variable.

We also used a relatively small Time Period, which forced the scaling, because as per the developer's words: *The training sometimes could be tricky. When the time domain is small, it is easy to train. But the training becomes hard for long time prediction, and hyperparameters have to be carefully tuned. On the other hand, for inverse problems, because we have extra points for the solution, which will make the training easier, although we have unknown coefficients. That is why in this case the inverse problem is easier to solve than the forward problem.*

Now the result of the above code is:

DeepXDE doesn't currently have built-in plotting customization, so to better highlight our solutions we must manually download and plot them using another tool.

To download the predicted data:

```
from google.colab import files
files.download( "test.dat" )
```

The file `"test.dat"` gives us a 7×100 matrix, containing the test points (our x axis) plus our 6 other solutions' corresponding values on the test points.

Using a spreadsheet, we obtain the following graph:

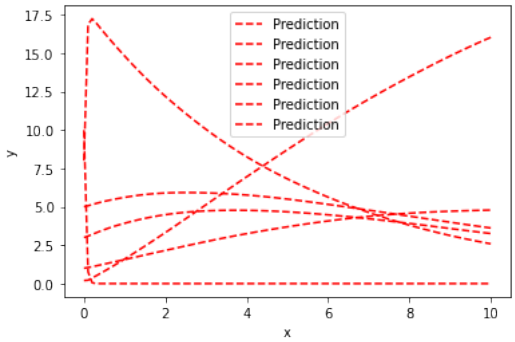

FIGURE 5.9: Solutions to the RP model

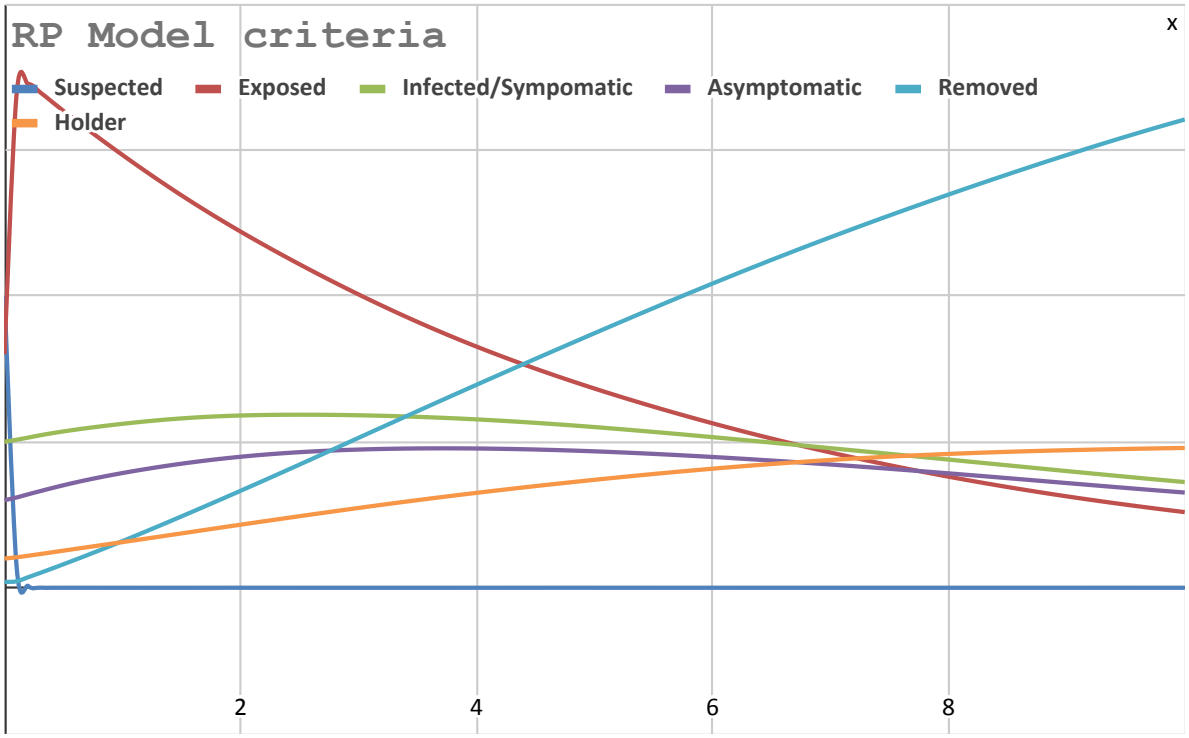

FIGURE 5.10: sds

Although we only used loosely-scaled initial conditions, the results are still pretty reliable. This emphasises the accuracy of both the model and DeepXDE's predictions.

As we can see, as time goes: suspected cases plummet, exposed cases steadily regress after an initial peak, symptomatic and asymptomatic carriers gradually diminish and evidently the removed cases, either dismissed, cured or deceased goes up. This is more or less how things developed in real life.

If we want to use the predictions for discrete points (e.g how many projected cases on day 20), we must use the `Predict` module. Since `Predict`, much like most of the other modules, deals in arrays, we must declare our as test point, let's consider it it $X = 20$, as `X = np.array([[20]])`.

Now to call onto the predicted solutions on $X = 20$, we use:

```
Y = model.predict(X)
```

Y is 1×6 matrix, i.e., $[[y1(x1), ..., y6(x1)]$

So our solutions are but components of Y. Getting the projected cases:

```
In[]: print('Y(20)= ',Y)

```

```
Out[]: Y(20)=  [[-0.6032586  6.3330913  5.148689   5.0569577  8.837534
  3.5146205]]
```

Epilogue

The main theme of this work was solving differential equations: we looked at the conditions for which there exist solutions, locally or globally, without the need to explicit them. Those are very important results that allow us to use the tool at our disposal to their full potential.

This here project highlighted the important advances in the application of neural networks to scientific computing. We have on our hands a very powerful tool that has the potential to solve any type of differential equations, the only constraints being the required knowledge of neural networks and deep learning, which is rather necessary to tune the hyperparamters and scale them in order to suit the problem in hand. While DeepXDE is still in its infancy, the feedback and discussions from various users are constantly helping it improve. This shows that the emerging technologies in scientific computing are a very welcome and in-demand tool, and thus extensive education in the field of Machine Learning should start being more prioritized.

Bibliography

Charu C., Aggarwal (2018). *Neural Networks and Deep Learning: A textbook*.

Chen et al. (2020). "A mathematical model for simulating the phase-based transmissibility of a novel coronavirus". In: *Infectious diseases of poverty*.

Corliss et al. (2001). *Automatic Differentiation of Algorithms: From Simulation to Opztimization*.

Csáji, Balázs Csanád (2001). *Approximation with Artificial Neural Networks*.

Cybenko, George (1989). *Approximation by superpositions of sigmoidal functions. Mathematics of Control, Signals, and Systems.*

DeepXDE documentation (n.d.). URL: https://deepxde.readthedocs.io/en/latest/.

Ganesh, S. Sivaji (2016). *Lecture notes on Ordinary Differential Equations, IIT Mombay*.

Grant, Christopher (n.d.). "Picard-Lindelof Theorem Lecture 4 Math 634".

Haykin, Simon (2008). *Neural Networks and Learning Machines, 3rd Edition*.

Hornik, Kurt (1991). *Approximation Capabilities of Multilayer Feedforward Networks. Neural Networks, vol. 4*.

Lorke et al. (2019). "Cybenko's Theorem and the capabilityof a neural networkas function approximator".

Lu, Lu et al. (2019). "DeepXDE: A deep learning library for solving differential equations". In: *arXiv preprint arXiv:1907.04502*.

luluxvi/deepxde (n.d.). URL: https://github.com/lululxvi/deepxde.

MATLAB Deep Learning toolbox (n.d.). URL: https://www.mathworks.com/help/deeplearning/.

missinglink.ai (n.d.). URL: https://missinglink.ai/guides/neural-network-concepts/complete-guide-artificial-neural-networks/.

Nielsen., Michael A. (2015). *Neural Networks and Deep Learning, Determination Press*.

Pinkus, Allan (1999). "Approximation theory of the MLP model in neural networks". In: *Acta Numerica*.

Raissi, Maziar, Paris Perdikaris, and George E Karniadakis (2019). "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378, pp. 686–707.

Raissi, Maziar, Paris Perdikaris, and George Em Karniadakis (2017a). "Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations". In: *arXiv preprint arXiv:1711.10561*.

— (2017b). "Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations". In: *arXiv preprint arXiv:1711.10566*.

Ruder, Sebastian (2015). *An overview of gradient descent optimization algorithms*.

Trench, William F. (2013). *"Elementary Differential Equations and Boundary Value Problems"*.

Umberto, Michelucci (2018). *Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks*.

Volt, Jan (2015). "Deep neural networks and their implementation".

Yannick Viossat, Daniela Tonon (2020). *Differential equations, Paris Dauphine MIDO L3*.

Yves Chauvin (ed.), David E. Rumelhart (ed.) (1995). *Backpropagation: Theory, Architectures, and Applications*.