UNIVERSITÀ DELLA CALABRIA

Department of Mathematics and Computer Science

Master's degree in Mathematics

Neural Networks in reaction-diffusion models

SUPERVISORS

Prof. Giovanni Mascali

Prof. Gennaro Infante

CANDIDATE

Riccardo Scida

M. N. 235617

Academic year 2023-2024

Table of Contents

Introduction

In recent years, the development of Neural Networks (NN), inspired by the structure and functioning of the human brain, has been a major revolution in the fields of Artificial Intelligence (AI) and Machine Learning (ML). These disciplines represent the most revolutionary tools in the contemporary scientific and technological landscape. They are concerned with building algorithms capable of performing very complex tasks, such as image recognition, sound recognition, and various autonomous decision-making processes. In doing so, Artificial Intelligence has found application in a wide range of fields, transforming the way many industries operate and improving the quality of people's lives. Some examples are:

1. in the field of health care, AI systems have been designed that can analyse medical images and clinical data so as to support doctors in diagnosing diseases such as cancer or certain heart diseases;
2. in the field of finance, AI systems capable of monitoring financial transactions have been designed to identify suspicious activities and prevent fraud;
3. in the field of transportation, some companies, such as Tesla, are using AI to develop self-driving cars that can sense their surroundings and make driving decisions, including optimizing routes and thus, for example, reducing delivery times and costs;
4. in the field of E-commerce, some platforms, such as Amazon and Netflix, use AI to analyse user preferences and recommend relevant products or content;
5. in the field of video games, AI is used to create "intelligent" and dynamic non-player characters (NPCs), making the gaming experience more realistic.

These examples are just a few of the many applications of Artificial Intelligence, and they demonstrate how this technology is becoming, more and more, a part of our daily lives and work.

In our work we will exploit a very important property of Neural Networks, that is their ability to approximate, under appropriate assumptions, any continuous function defined on $\mathbb{R}^n$. This led to the idea of using Neural Networks to numerically approximate the solutions of partial differential equations (PDEs). In particular, we will consider reaction-diffusion equations. PDEs occupy a central role in the creation of models describing the evolution of complex systems, as a function of space and time, such as physical, engineering, biological systems, etc., and have various applications, including fluid dynamics, wave propagation, heat diffusion, and material deformation. This combination of Neural Networks and PDEs represents a promising research frontier and opens up new possibilities for tackling problems previously considered "inaccessible" or excessively expensive to solve by traditional methods. Our goal, in this thesis, precisely, will be to analyse what has already been studied in the field of Neural Networks to solve ordinary differential equations (ODEs) and some PDEs, and then to deepen these studies also in the case of reaction-diffusion models. In particular, after developing these concepts from the theoretical point of view, we will try to build a "bridge" that will allow us to implement in Matlab algorithms to approximate, via Neural Networks, some specific models, such as the Fisher-Kolmogorov-Petrovsky-Piskunov model. The advantages and disadvantages of these tools will be discussed, with the aim of providing an overview of possible future research directions. Thus, an attempt will be made to demonstrate how Neural Networks can be an effective and innovative tool to address the study of reaction-diffusion patterns, paving the way for new applications and improvements in various fields of science.

Chapter 1

Partial Differential Equations (PDEs)

1.1 Introduction on PDEs

1.1.1 Definition of PDEs

Let $k \geq 1$ be a non-negative integer and let $U \subseteq \mathbb{R}^n$ be an open set, a *partial differential equation* [2] of order k is an equation of the form

$$\textbf{(1.1)} \quad F\left(D^k u(x), D^{k-1} u(x), \ldots, u(x), x\right) = 0,$$

where

- $x \in U$;
- $F : \mathbb{R}^{n^k} \times \mathbb{R}^{n^{k-1}} \times \ldots \times \mathbb{R}^n \times \mathbb{R} \times U \to \mathbb{R}$ is a given function;
- $u : U \to \mathbb{R}$ is the unknown.

In this definition we have used the

1.1.2 Multi-index Notation

A vector of the form $\alpha = (\alpha_1, \ldots, \alpha_n)$, where each component α_i is a non-negative integer, is called a *multi-index* of order

$$|\alpha| = \alpha_1 + \ldots + \alpha_n.$$

Given a multi-index $\alpha = (\alpha_1, \ldots, \alpha_n)$, define

$$D^\alpha u(x) := \frac{\partial^{|\alpha|} u(x)}{\partial x_1^{\alpha_1} \ldots \partial x_n^{\alpha_n}} = \partial x_1^{\alpha_1} \ldots \partial x_n^{\alpha_n} u.$$

If k is a non-negative integer, we can define with

$$D^k u(x) := \{ D^\alpha u(x) \,|\, |\alpha| = k \}$$

the set of all partial derivatives of order k.

1.1.3 Types of PDEs

A PDE is called:

1. *linear*, if it's of the form
$$\sum_{|\alpha|\le k} a_\alpha(x) D^\alpha u = f(x).$$
This PDE is said to be *homogeneous* if $f \equiv 0$;
2. *semi-linear*, if it's of the form
$$\sum_{|\alpha|=k} a_\alpha(x) D^\alpha u + a_0(D^{k-1}u, \dots, Du, u, x) = 0;$$
3. *quasilinear*, if it's of the form
$$\sum_{|\alpha|=k} a_\alpha(D^{k-1}u, \dots, Du, u, x) D^\alpha u + a_0(D^{k-1}u, \dots, Du, u, x) = 0;$$
4. *fully not linear*, if it depends non-linearly upon the highest order derivatives.

1.1.4 Definition of system of PDEs

Let $k \ge 1$ be an integer and let $U \subseteq \mathbb{R}^n$ be an open set, a *system of partial differential equations* of order k is an equation of the form

$$\mathbf{(1.2)} \quad \boldsymbol{F}(D^k \boldsymbol{u}(x), D^{k-1}\boldsymbol{u}(x), \dots, \boldsymbol{u}(x), x) = \boldsymbol{0},$$

where:

- $x \in U$;
- $\boldsymbol{F} : \mathbb{R}^{mn^k} \times \mathbb{R}^{mn^{k-1}} \times \dots \times \mathbb{R}^{mn} \times \mathbb{R}^m \times U \to \mathbb{R}^m$ is a given function;
- $\boldsymbol{u} : U \to \mathbb{R}^m, \boldsymbol{u} = (u^1, \dots, u^m)$ is the unknown.

1.1.5 Examples of PDEs

The following equations are linear:

- *Laplace's equation*
$$\Delta u = 0.$$
- *Eigenvalue equation*
$$-\Delta u = \lambda u.$$
- *Linear transport equation*
$$u_t + \sum_{i=1}^{n} b^i u_{x_i} = 0.$$
- *Liouville's equation*

$$u_t - \sum_{i=1}^{n} (b^i u)_{x_i} = 0.$$

- *Heat equation*

$$u_t - \Delta u = 0.$$

- *Wave equation*

$$u_{tt} - \Delta u = 0.$$

The following equations are nonlinear:

- *Eikonal equation*

$$|Du| = 1.$$

- *Nonlinear Poisson equation*

$$-\Delta u = f(u).$$

- *Scalar reaction-diffusion equation*

$$u_t - \Delta u = f(u).$$

- *Nonlinear wave equation*

$$u_{tt} - \Delta u = f(u).$$

The following are linear systems:

- *Equilibrium equations of linear elasticity*

$$\mu \Delta \boldsymbol{u} + (\lambda + \mu) D(div\, \boldsymbol{u}) = \boldsymbol{0}.$$

- *Evolution equations of linear elasticity*

$$\boldsymbol{u}_{tt} - \mu \Delta \boldsymbol{u} - (\lambda + \mu) D(div\, \boldsymbol{u}) = \boldsymbol{0}.$$

The following are nonlinear systems:

- *System of conservation laws*

$$\boldsymbol{u}_t + div\, \boldsymbol{F}(\boldsymbol{u}) = \boldsymbol{0}.$$

- *reaction-diffusion system*

$$\boldsymbol{u}_t - \Delta(\boldsymbol{u}) = \boldsymbol{f}(\boldsymbol{u}).$$

1.2 Linear differential operators

1.2.1 Definition of linear differential operators

A linear *differential operator* [7] of order n is a linear operator A acting on functions $u \in C^n$ in the ordinary way:

$$\textbf{(1.3)} \quad Au = \sum_{j=0}^{n} a_j u^{(j)},$$

where

- $u^{(j)} = \dfrac{d^j u}{d x^j}$, $u^{(0)} = u(x)$, $x \in I \subseteq \mathbb{R}$;
- a_j are real-complex valued functions.

If $n = 2$ we have operators of the type

$$Au = au'' + bu' + cu,$$

where every function is regular on $[0,1]$ e $a(x) > 0, \forall x \in [0,1]$, where, without loss of generality, we have taken $I = [0,1]$.

1.2.2 Boundary conditions

We will study problems of the type

$$\textbf{(1.4)} \quad Au = f,$$

with boundary conditions. The most common are

- *Dirichlet conditions* $\begin{cases} u(0) = a, \\ u(1) = b, \end{cases}$ $a, b \in \mathbb{R}$;
- *Neumann conditions* $\begin{cases} u'(0) = c, \\ u'(1) = d, \end{cases}$ $c, d \in \mathbb{R}$;
- *periodic conditions* $\begin{cases} u(0) = u(1), \\ u'(0) = u'(1), \end{cases}$
- *mixed conditions* $\begin{cases} a_0 u(0) + b_0 u'(0) = \alpha, \\ a_1 u(1) + b_1 u'(1) = \beta, \end{cases}$ $\alpha, \beta \in \mathbb{R}$.

1.2.3 Dirac Delta

Roughly speaking, the Dirac Delta δ is a "function" on $\mathbb{R}$ with the following formal properties:

- $\delta(x) = 0, \forall x \neq 0$;
- $\int_{-\infty}^{+\infty} \delta(x) dx = 1$;
- $\int_{-\infty}^{+\infty} \delta(y - x) f(y) dy = f(x)$;

- $\int_{-\infty}^{y} \delta(x)dx = \begin{cases} 0, if\ y<0, \\ 1, if\ y \geq 0. \end{cases}$

Of course, precise meaning can be given to these results only within the framework of the theory of distributions, for example, see [4]. In this section, we have limited ourselves only to a heuristic discussion suitable for motivating the classical definition of

1.2.4 Definition of Green function

In the framework of the classical theory, a function $g:[0,1]\times[0,1]\rightarrow\mathbb{C}$ is said to be a *Green function* if:

- g is a continuous function in the box $[0,1]\times[0,1]$ and it is twice differentiable with respect to x in the triangles $0\leq x\leq y\leq 1$ and $0\leq y\leq x\leq 1$;
- $Ag=0$ in $0<x<y<1$ and $0<y<x<1$;
- $g(0,y)=g(1,y)=0$;
- the "jump" of g_x across the line $x=y$ is given by

$$g_x(y^+,y)-g_x(y^-,y)=\frac{1}{a(y)}.$$

Remark

Let $A=a\frac{d^2}{dx^2}+b\frac{d}{dx}+c$ be a linear differential operator of order 2 and, without loss of generality, let's consider the homogeneous Dirichlet problem

$$\textbf{(1.5)} \quad \begin{cases} Au=f, \\ u(0)=u(1)=0, \end{cases}$$

with $f:[0,1]\rightarrow\mathbb{C}$ continuous function. We are looking for solutions, if they exist, such as

$$\textbf{(1.6)} \quad u(x)=\int_0^1 g(x,y)f(y)dy,$$

where $g:[0,1]\times[0,1]\rightarrow\mathbb{C}$ is called the *Green function*. Let us define, if it exists,

$$G:C([0,1])\rightarrow D(A),$$

$$\textbf{(1.7)} \quad G(f(x))=\int_0^1 g(x,y)f(y)dy,$$

as the inverse of A, such that to a function f it associates the function u which is solution of the Dirichlet problem. The function $g(x,y)$ is also called the *kernel* of G. The Green function is the formal solution of the problem

$$\textbf{(1.8)} \quad \begin{cases} Ag(x,y)=\delta(x-y), \\ g(0,y)=g(1,y)=0, \end{cases}$$

indeed, formally we have:

$$Au(x)=\int_0^1 Ag(x,y)f(y)dy=\int_0^1 \delta(x-y)f(y)dy=f(x),$$
$$u(0)=\int_0^1 g(0,y)f(y)dy=0, \quad u(1)=\int_0^1 g(1,y)f(y)dy=0.$$

1.2.5 Proposition (Green function as solution of the Dirichlet problem)

Let $A=a\frac{d^2}{dx^2}+b\frac{d}{dx}+c$ a linear differential operator with $a,b,c\in C([0,1]), a\neq 0,$

If g is a Green function and $f\in C([0,1])$, then

$$\textbf{(1.9)} \quad u(x)=G(f(x))=\int_0^1 g(x,y)f(y)dy$$

is the solution of the Dirichlet problem

$$\begin{cases} Au=f, \\ u(0)=u(1)=0. \end{cases}$$

1.2.6 Proposition (Building a Green function)

To construct the Green's function, we can choose two nonzero solutions v_1, v_2 of the homogeneous case $Au=0$, such that

$$v_1(0)=v_2(1)=0.$$

There exists two solutions that are linearly independent if and only if the problem

$$\begin{cases} Au=0, \\ u(0)=u(1)=0, \end{cases}$$

has the unique solution $u\equiv 0$. In this case we can define a Green function as

$$\textbf{(1.10)} \quad g(x,y)=\begin{cases} C(y)v_1(x)v_2(y), & for \quad 0\le x\le y\le 1, \\ C(y)v_1(y)v_2(x), & for \quad 0\le y\le x\le 1, \end{cases}$$

where

- $C(y)=\frac{1}{a(y)W(y)}$;

- $W(y)=\begin{vmatrix} v_1 & v_2 \\ v_1' & v_2' \end{vmatrix}=v_1v_2'-v_2v_1'\neq 0$, is said the *Wronskian* of v_1 and v_2.

Let us note that

$$\begin{cases} Ag=0, \\ g(0,y)=g(1,y)=0, \forall\, y\in[0,1], \end{cases}$$

and moreover

$$\begin{aligned} & g_x(y^+,y)-g_x(y^-,y) \\ & \quad =C(y)v_1(y^+)v_2'(y)-C(y)v_1'(y^-)v_2(y) \\ & \quad =C(y)[v_1(y)v_2'(y)-v_1'(y)v_2(y)] \\ & \quad =C(y)W(y)=\frac{1}{a(y)}. \end{aligned}$$

1.3 Laplace-Poisson equations

1.3.1 Definition of Laplace and Poisson equations

Let $U\subseteq\mathbb{R}^n$ be a connected, open and bounded set, $u:\bar{U}\to\mathbb{R}$ be the unknown and $f:U\to\mathbb{R}$ be a given function. Let us define the *Laplace equation* [7] as

$$\textbf{(1.11)}\quad \Delta u=0,$$

and the *Poisson equation* as

$$\textbf{(1.12)}\quad -\Delta u=f,$$

where the *Laplacian* of u is given by

$$\Delta u=\sum_{i=1}^{n}\frac{\partial^2 u}{\partial x_i^2}.$$

1.3.2 Definition of harmonic function

A function $u\in C^2$ such that $\Delta u=0$ is called a *harmonic function*.

1.3.3 Physical interpretation

Laplace's equation is found in many physical contexts. Suppose that u is the density of a certain quantity in equilibrium and let $\boldsymbol{F}:\bar{U}\to\mathbb{R}^n$ be a differentiable vector field, representing the flux of u, then by Gauss's Theorem:

$$\textbf{(1.13)}\quad \int_U \nabla\cdot\boldsymbol{F}\,d\boldsymbol{x}=\int_{\partial U}\boldsymbol{F}\cdot\boldsymbol{n}\,d\sigma.$$

We assume that the flux of u through ∂U is zero, i.e.

$$\int_{\partial U} \boldsymbol{F}\cdot\boldsymbol{n}\, d\sigma = 0,$$

where

- $\boldsymbol{n}$ is the normal versor to the surface ∂u oriented outward;
- $d\sigma$ is the area element of the surface.

hence

$$\nabla\cdot\boldsymbol{F} = 0, \text{ in } U.$$

In some physical examples, we can suppose that $\boldsymbol{F}$ is proportional to the *gradient* ∇u, but with opposite direction, i.e.

$$\boldsymbol{F} = -a\nabla u, \quad a>0,$$

then substituting yields

$$\nabla\cdot\nabla u = \Delta u = 0.$$

1.3.4 Boundary conditions

For the Laplace and Poisson equations, the most common boundary conditions are

- $u = h$, on ∂U, (*Dirichlet condition*);
- $\frac{\partial u}{\partial n} := \boldsymbol{n}\cdot\nabla u = h$, on ∂U, (*Neumann condition*).

1.3.5 Theorem (Green)

Let's take $u, v \in C^2(\overline{U})$, then

$$\textbf{(1.14)} \quad \int_U (u\Delta v - v\Delta u)\, d\boldsymbol{x} = \int_{\partial U} \left(u\frac{\partial v}{\partial n} - v\frac{\partial u}{\partial n}\right) d\sigma.$$

1.3.6 n-dimensional Dirac delta

We can define the *n-dimensional Dirac delta* as a "function" with the following formal properties:

- $\delta(\boldsymbol{x}) = 0, \forall\, \boldsymbol{x}\neq 0;$
- $\int_{\mathbb{R}^n} \delta(\boldsymbol{x})\, d\boldsymbol{x} = 1;$
- $\int_{\mathbb{R}^n} \delta(\boldsymbol{x}-\boldsymbol{y})\, f(\boldsymbol{y})\, d\boldsymbol{y} = f(\boldsymbol{x}), \forall\, f\in C(\mathbb{R}^n).$

The Green's function $g(\boldsymbol{x},\boldsymbol{y})$ associated with the Laplacian problem, with Dirichlet homogeneous boundary conditions, is solution of the problem

$$\textbf{(1.15)} \quad \begin{cases} -\Delta g(\boldsymbol{x},\boldsymbol{y}) = \delta(\boldsymbol{x}-\boldsymbol{y}), \ \boldsymbol{x}\in U, \ \boldsymbol{y}\in U, \\ g(\boldsymbol{x},\boldsymbol{y}) = 0, \ \boldsymbol{x}\in \partial U, \ \boldsymbol{y}\in U. \end{cases}$$

1.3.7 Solution of the Dirichlet problem of the Laplacian

Let's consider the problem

$$\textbf{(1.16)} \quad \begin{cases} -\Delta u(\boldsymbol{x}) = f(\boldsymbol{x}), \ \boldsymbol{x}\in U, \\ u(\boldsymbol{x}) = h(\boldsymbol{x}), \ \boldsymbol{x}\in \partial U, \end{cases}$$

and let $g(\boldsymbol{x},\boldsymbol{y})$ be the solution of the problem **(1.15)**, then by the Green formula one gets

$$\int_U (g\Delta u - u\Delta g)\, d\boldsymbol{x} = \int_{\partial U} \left(g\frac{\partial u}{\partial n} - u\frac{\partial g}{\partial n}\right) d\sigma = \int_{\partial U} -u\frac{\partial g}{\partial n}\, d\sigma .$$

By substituting we obtain

$$\int_U (-g(\boldsymbol{x},\boldsymbol{y}) f(\boldsymbol{x}) + u(\boldsymbol{x})\delta(\boldsymbol{x}-\boldsymbol{y}))\, d\boldsymbol{x} = \int_{\partial U} -h(\boldsymbol{x})\frac{\partial g(\boldsymbol{x},\boldsymbol{y})}{\partial n}\, d\sigma(\boldsymbol{x}),$$

and then

$$u(\boldsymbol{y}) = \int_U g(\boldsymbol{x},\boldsymbol{y}) f(\boldsymbol{x})\, d\boldsymbol{x} - \int_{\partial U} h(\boldsymbol{x})\frac{\partial g(\boldsymbol{x},\boldsymbol{y})}{\partial n}\, d\sigma(\boldsymbol{x}).$$

By renaming the variables and using the symmetry of g, which is due to the self-adjointness of $-\Delta$, see [2], we have that

$$\textbf{(1.17)} \quad u(\boldsymbol{x}) = \int_U g(\boldsymbol{x},\boldsymbol{y}) f(\boldsymbol{y})\, d\boldsymbol{y} - \int_{\partial U} h(\boldsymbol{y})\frac{\partial g(\boldsymbol{x},\boldsymbol{y})}{\partial n_y}\, d\sigma(\boldsymbol{y})$$

is solution of Dirichlet's Laplacian problem.

Proof

For the rigorous proof see [2], for example.

1.4 Heat equation

1.4.1 Definition of Heat equation

Let $U \subseteq \mathbb{R}^n$ be a connected, open and bounded set, $u : \overline{U} \times [0, \bar{t}] \to \mathbb{R}$ be the unknown, $f : U \to \mathbb{R}$ and $g : \partial U \times [0, \bar{t}] \to \mathbb{R}$ be given functions, $K > 0$ be the thermal diffusion. Let us define the Heat *equation* [7] as

$$\textbf{(1.18)} \quad \begin{cases} \dfrac{\partial u}{\partial t} - K \Delta u = 0, \ (\boldsymbol{x}, t) \in U \times (0, \bar{t}), \\ u(\boldsymbol{x}, 0) = f(\boldsymbol{x}), \ \boldsymbol{x} \in U, \\ u(\boldsymbol{x}, t) = g(\boldsymbol{x}, t), \ (\boldsymbol{x}, t) \in \partial U \times [0, \bar{t}]. \end{cases}$$

1.4.2 Definition of parabolic boundary

Let $U \subseteq \mathbb{R}^n$ be a connected, open and bounded set and let's take $t \in [0, \bar{t}]$, we can define the *parabolic boundary* of the cylinder $Q_T := U \times (0, \bar{t})$ as the set

$$\partial_p Q_T := (\overline{U} \times \{t = 0\}) \cup S_T,$$

where $S_T = \partial U \times (0, \bar{t}]$ is the *lateral surface* of the cylinder.

1.4.3 Proposition 1 (about the Weak Maximum principle)

Let $u \in C^{2,1}(Q_T) \cap C(\overline{Q_T})$ satisfy

$$\textbf{(1.19)} \quad \frac{\partial u}{\partial t} - K \Delta u < 0, \ (\boldsymbol{x}, t) \in U \times (0, \bar{t}), K > 0.$$

If $u \le M$ in $U \times \{t = 0\}$ and $u \le M$ in $\partial U \times [0, \bar{t}]$, then $u \le M$ in $\overline{U} \times [0, \bar{t}]$.

Proof

Let's suppose by contradiction that $(\boldsymbol{x_0}, t_0)$ is a maximum point, with $\boldsymbol{x_0} \in U$, $t_0 \in (0, \bar{t})$, then

$$\frac{\partial}{\partial t} u(\boldsymbol{x_0}, t_0) = 0,$$

$$\Delta u(\boldsymbol{x_0}, t_0) \le 0,$$

and this is a contradiction because one would have

$$\frac{\partial u}{\partial t} - K \Delta u \ge 0.$$

Then we have shown that

$$\textbf{(1.20)} \qquad \max_{U\times(0,\bar{t})} u \le \max_{\partial U\times(0,\bar{t}]\cup U\times\{0\}} u .$$

Let us suppose that there exists a maximum point $(x_0,\bar{t})$, with $x_0\in U$, such that

$$u(x_0,\bar{t})>M .$$

By the continuity of u, there exists $\varepsilon>0$, such that $u(x_0,\bar{t}-\varepsilon)>M$, but $(x_0,\bar{t}-\varepsilon)\in U\times(0,\bar{t})$, whence it's a contradiction with **(1.20)**.

■

1.4.4 Proposition 2 (about the Weak Maximum principle)

Let us suppose that

$$\textbf{(1.21)} \qquad \frac{\partial u}{\partial t}-K\Delta u=0,\ (\boldsymbol{x},t)\in U\times(0,\bar{t}),K>0,$$

with $u\in C^{2,1}(Q_T)\cap C(\overline{Q_T})$. If $u\le M$ in $U\times\{t=0\}$ and in $\partial U\times[0,\bar{t}]$, then $u\le M$ in $\overline{U}\times[0,\bar{t}]$.

Proof

Let us define

$$v=u+\varepsilon\left(\sum_{i=1}^{n} x_i^2\right),$$

and we observe that

$$\frac{\partial v}{\partial t}=\frac{\partial u}{\partial t},$$

$$\Delta v=\Delta u+2n\varepsilon .$$

Then we obtain

$$\frac{\partial v}{\partial t}-K\Delta v=\frac{\partial u}{\partial t}-K(\Delta u+2n\varepsilon)=-K2n\varepsilon<0,$$

that is v satisfies **(1.19)**. Moreover, since U is bounded, there exists $R>0$ such that $U\subseteq B(0,R)$, whence

$$v\le M+\varepsilon R^2 \text{ in } U\times\{t=0\} \text{ and in } \partial U\times[0,\bar{t}].$$

From the previous proposition, we have that

$$v\le M+\varepsilon R^2 \text{ in } \overline{U}\times[0,\bar{t}].$$

The thesis follows taking the limit for ε going to zero.

■

By **Proposition 1** and **Proposition 2**, we obtain the following

1.4.5 Theorem (Weak Maximum principle)

Let us suppose that

$$\textbf{(1.22)} \qquad \frac{\partial u}{\partial t} - K\Delta u \le 0,\ (\boldsymbol{x},t) \in U \times (0,\bar{t}),\ K > 0,$$

with $u \in C^{2,1}(Q_T) \cap C(\overline{Q_T})$. If $u \le M$ in $U \times \{t=0\}$ and in $\partial U \times [0,\bar{t}]$, then $u \le M$ in $\overline{U} \times [0,\bar{t}]$, that is, the maximum is assumed on the parabolic boundary.

1.4.6 Definition of subsolutions and supersolutions

The functions u such that $\frac{\partial u}{\partial t} - K\Delta u \le 0$ $\left(\frac{\partial u}{\partial t} - K\Delta u \ge 0\right)$ are called *subsolutions* (*supersolutions*) of the heat equation.

1.4.7 Theorem (uniqueness of the solution)

If there exists a solution for the problem

$$\textbf{(1.23)} \qquad \begin{cases} \frac{\partial u}{\partial t} - K\Delta u = 0,\ (\boldsymbol{x},t) \in U \times (0,\bar{t}), \\ u(\boldsymbol{x},0) = f(\boldsymbol{x}),\ \boldsymbol{x} \in U, \\ u(\boldsymbol{x},t) = g(t),\ (\boldsymbol{x},t) \in \partial U \times [0,\bar{t}], \end{cases}$$

then this solution is unique.

Proof

Let's suppose by contradiction that there exist u_1, u_2 both solutions of **(1.23)**. Let's define $u := u_1 - u_2$, and observe that it's solution of

$$\begin{cases} \frac{\partial u}{\partial t} - K\Delta u = 0,\ (\boldsymbol{x},t) \in U \times (0,\bar{t}), \\ u(\boldsymbol{x},0) = 0,\ \boldsymbol{x} \in U, \\ u(\boldsymbol{x},t) = 0,\ (\boldsymbol{x},t) \in \partial U \times [0,\bar{t}], \end{cases}$$

that is $u = 0$ in $\{U \times \{t=0\}\} \cup \{\partial U \times [0,\bar{t}]\}$. By the *Weak Maximum principle theorem*, we obtain

$$u = 0 \text{ in } \overline{U} \times [0,\bar{t}],$$

that is $u_1 = u_2$ in $\overline{U} \times [0,\bar{t}]$.

■

1.4.8 Theorem (Strong Maximum principle)

Let us suppose that

$$\frac{\partial u}{\partial t} - K\Delta u \le 0, \quad (\boldsymbol{x}, t) \in U \times (0, \bar{t}), K > 0,$$

with $u \in C^{2,1}(Q_T) \cap C(\overline{Q_T})$. If u has maximum M in $(\boldsymbol{x}_1, t_1) \in U \times (0, \bar{t}]$, then $u \equiv M$ in $\overline{U} \times [0, t_1]$.

1.4.9 Fundamental solution of the heat equation (n=1)

We want to study the propagation, in an infinite homogeneous metal bar placed along the real axis, of the heat produced by an instantaneous source in the origin of the axes at time $t = 0$. We want to find the solution u^* of

$$u_t - D u_{xx} = 0, \quad t > 0.$$

The law of conservation of energy implies that

$$\textbf{(1.24)} \qquad \rho c_v \int_{\mathbb{R}} u^*(x, t)\, dx = E,$$

where

- ρ is the mass density of the bar;
- c_v is the specific heat;
- E is the total energy.

STEP 1

Let us define $Q = \frac{E}{\rho c_v}$, we want u^* to be an even function of x and depend also on t, D, Q, i.e.

$$u^* = U(x, t, D, Q) \text{ and } U(x) = U(-x).$$

In order to non-dimensionalize the variables, we list the physical dimensions of each argument of U, in terms of the three dimensional units: Θ (temperature), L (length) and T (time):

$$[x] = L, [t] = T, [D] = L^2 T^{-1}, [Q] = \Theta L,$$

or similarly

$$[x] = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, [t] = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, [D] = \begin{bmatrix} 0 \\ 2 \\ -1 \end{bmatrix}, [Q] = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

We observe that $[t],[D],[Q]$ are linearly independent and thus form a basis for $\mathbb{R}^3$. It follows from these observations that

- $[u^*]=\Theta=\frac{[Q]}{L}=\frac{[Q]}{[D]^{1/2}[t]^{1/2}}$;
- $[x]=L=[D]^{1/2}[t]^{1/2}$.

Let's define the non-dimensional quantities

- $\pi=\frac{u^*\sqrt{Dt}}{Q}$;
- $\pi_1=\frac{x}{\sqrt{Dt}}$.

From these definitions and from that of u^*, one obtains

$$\pi=\frac{\sqrt{Dt}}{Q}U(\pi_1)=\widetilde{U}(\pi_1,t,D,Q)=\widetilde{U}(\pi_1),$$

since π and $\widetilde{U}$ are dimensionless. Then

$$\textbf{(1.25)}\qquad u^*(x,t)=\frac{Q}{\sqrt{Dt}}\cdot\widetilde{U}\left(\frac{x}{\sqrt{Dt}}\right).$$

STEP 2

By the law of conservation of energy and making the change of variables $y=\frac{x}{\sqrt{Dt}}$, we have that

$$Q=\int_{\mathbb{R}}u^*(x,t)\,dx=\int_{\mathbb{R}}\frac{Q}{\sqrt{Dt}}\widetilde{U}\left(\frac{x}{\sqrt{Dt}}\right)dx=Q\int_{\mathbb{R}}\widetilde{U}(y)\,dy,$$

which implies

$$\textbf{(1.26)}\qquad \int_{\mathbb{R}}\widetilde{U}(y)dy=1.$$

Calculating the derivatives u_t^* and u_{xx}^*, it is derived that

$$u_t^*-D\,u_{xx}^*=\frac{-Q}{t\sqrt{Dt}}\left[\widetilde{U}''(y)+\frac{1}{2}\widetilde{U}'(y)+\frac{1}{2}\widetilde{U}(y)\right]=0.$$

Then u^* is solution of the problem if and only if

$$\widetilde{U}''(y)+\frac{1}{2}\widetilde{U}'(y)+\frac{1}{2}\widetilde{U}(y)=0,$$

that is

$$\frac{\partial}{\partial y}\left(\widetilde{U}'(y)+\frac{1}{2}y\,\widetilde{U}(y)\right)=0,$$

then

$$\widetilde{U}'(y)+\frac{1}{2}y\,\widetilde{U}(y)=C.$$

Since u^* is an even function, we have that $\widetilde{U}'(0)=0.$ Evaluating in zero gives $C=0.$ Thus, we obtain the equation

$$\widetilde{U}'(y)+\frac{1}{2}y\,\widetilde{U}(y)=0,$$

which has solution

$$\widetilde{U}(y)=c_0 e^{-y^2/4}.$$

From **(1.26)** we get

$$1=\int_{\mathbb{R}}\widetilde{U}(y)\,dy=\int_{\mathbb{R}}c_0 e^{-y^2/4}\,dy=c_0\int_{\mathbb{R}}e^{-z^2}\,2\,dz=2c_0\sqrt{\pi},$$

which implies

$$c_0=\frac{1}{\sqrt{4\pi}}.$$

Then we obtain

$$u^*(x,t)=\frac{Q}{\sqrt{4\pi Dt}}e^{\frac{-x^2}{4Dt}}.$$

The function

$$\textbf{(1.27)}\qquad \Gamma_D(x,t)=\frac{1}{\sqrt{4\pi Dt}}e^{\frac{-x^2}{4Dt}},\quad x\in\mathbb{R},\quad t>0,$$

is called the *fundamental solution of the heat equation* in one dimension. It is a *self-similar* solution in the sense that it keeps spatially "similar" to itself as time changes.

We observe that it possesses the following properties:

1. if $x\neq 0$, then $\lim_{t\to 0^+}\Gamma_D(x,t)=\lim_{t\to 0^+}\frac{1}{\sqrt{4\pi Dt}}e^{\frac{-x^2}{4Dt}}=0;$
2. if $x=0$, then $\lim_{t\to 0^+}\Gamma_D(x,t)=\lim_{t\to 0^+}\frac{1}{\sqrt{4\pi Dt}}e^{\frac{-x^2}{4Dt}}=+\infty;$
3. $\lim_{t\to 0^+}\int_{\mathbb{R}\setminus[-\delta,\delta]}\Gamma_D(x,t)\,dx=0,\ \forall\,\delta>0.$

From these properties it follows that the fundamental solution is concentrated around the origin. Furthermore, for $t\to 0^+$, we have that $\Gamma_D(x,t)$ satisfies all the

properties of Dirac's Delta. The function $\Gamma_D(x-y,t)$ can be interpreted as the solution of the unitary heat equation when the heat source is concentrated in y at $t=0$, that is it satisfies the initial condition

$$\Gamma_D(x-y,0)=\delta(x-y).$$

As soon as $t>0$, then $\Gamma_D(x,t)>0$ over all $\mathbb{R}$, that is, heat diffuses instantaneously therefore with infinite speed, which is a limit of the model of heat propagation.

1.4.10 Fundamental solution of the heat equation (n>1)

Let's study the diffusion, over all $\mathbb{R}^n$, of heat produced by an instantaneous source placed in the origin of axes at $t=0$. Similarly to the $n=1$ case, it is possible to show that the solution is

$$\textbf{(1.28)} \qquad u^*(\boldsymbol{x},t)=\frac{Q}{(4\pi Dt)^{n/2}}e^{\frac{-|\boldsymbol{x}|^2}{4Dt}}, \quad \boldsymbol{x}\in\mathbb{R}^n, \quad t>0.$$

The function

$$\textbf{(1.29)} \qquad \Gamma_D(\boldsymbol{x},t)=\frac{1}{(4\pi Dt)^{n/2}}e^{\frac{-|\boldsymbol{x}|^2}{4Dt}},$$

is called the *fundamental solution of the heat equation* in dimension n.

The function $\Gamma_D(\boldsymbol{x}-\boldsymbol{y},t)$ can be interpreted as the solution of the unitary heat equation when the heat source is concentrated in $\boldsymbol{y}$ and it's the unique solution of the problem

$$\begin{cases} u_t-D\Delta u=0, & \boldsymbol{x}\in\mathbb{R}^n, \quad t>0, \\ u(\boldsymbol{x},0)=\delta(\boldsymbol{x}-\boldsymbol{y}), & \boldsymbol{x}\in\mathbb{R}^n. \end{cases}$$

1.4.11 The global Cauchy problem (n=1)

Homogeneous case

Let's consider the homogeneous problem

$$\begin{cases} u_t-Du_{xx}=0, & x\in\mathbb{R}, \quad t>0, \\ u(x,0)=g(x), & x\in\mathbb{R}, \end{cases}$$

where g is a given function. The idea is to determine, for example, the concentration $u(x,t)$ of a substance whose initial concentration is $g(x)$. $g(y)dy$ represents the quantity contained in the interval $(y, y+dy)$, and

$$\Gamma_D(x-y,t)g(y)dy$$

represents the corresponding mass concentration at point x at time t.

It's possible to prove that the solution is given by

$$\textbf{(1.30)} \quad u(x,t)=\int_{\mathbb{R}} \Gamma_D(x-y,t)g(y)dy=\frac{1}{\sqrt{4\pi Dt}}\int_{\mathbb{R}} g(y)e^{\frac{-(x-y)^2}{4Dt}}dy,$$

which is called the *convolution of the initial value* with the fundamental solution. Of course, these are heuristic considerations, and it must be shown that it is indeed solution of the Cauchy problem under consideration.

1.4.12 Theorem (Existence of Solution)

Let g be a function with countable points of discontinuity in $\mathbb{R}$, such that

$$|g(x)|\leq c\,e^{ax^2}, \quad \forall\, x\in\mathbb{R},$$

with $a, c>0$, and let $u(x,t)$ be defined as in **(1.30)**, then

1. $u(x,t)$ is well defined and differentiable infinitely many times in $\mathbb{R}$, $\forall\, t<T$, with $T<\frac{1}{4Da}$, and we have that
$$u_t-Du_{xx}=0, \text{ in } \mathbb{R}\times(0,T);$$
2. if g is continuous in x_0, then
$$\lim_{(x,t)\to(x_0,0)} u(x,t)=g(x_0), \quad t>0;$$
3. there exist A, C constants such that
$$|u(x,t)|\leq C\,e^{Ax^2}, \text{ in } \mathbb{R}\times(0,T).$$

This theorem states that **(1.30)** is solution of the homogeneous Cauchy problem in a finite time interval.

Non-homogeneous case

Let's consider the problem

$$\begin{cases} v_t-Dv_{xx}=\delta(x-y,t-s), & s>0, \\ v(x,0)=0, & x\in\mathbb{R}. \end{cases}$$

Since up to the time $t=s>0$ nothing happens and after that time $\delta(x-y,t-s)=0$, we can alternatively consider the problem

$$\begin{cases} u_t - D u_{xx} = 0, & x \in \mathbb{R}, \quad t > s, \\ u(x,s) = \delta(x-y), & s \geq 0. \end{cases}$$

If $s=0$, the solution is

$$u(x,t) = \Gamma_D(x-y,t),$$

instead, if s is generic, the solution becomes

$$u(x,t) = \Gamma_D(x-y,t-s).$$

Now consider a source distributed in the strip $\mathbb{R}\times(0,T)$ that produces mass density at the rate $f(x,t)$, then

$$f(x,t)\,dx\,dt$$

is the mass produced in the spatial region $(x, x+dx)$ in the time interval $(t, t+dt)$.

If no mass is initially present, the problem becomes

$$\textbf{(1.31)} \qquad \begin{cases} v_t - D v_{xx} = f(x,t), & in \ \mathbb{R}\times(0,T), \\ v(x,0) = 0, & x \in \mathbb{R}. \end{cases}$$

STEP 1

We construct a family $w(x,t,s)$ of solutions of the homogeneous Cauchy problem, in which the initial time, instead of being $t=0$, fixed, is a time $0<s<t$, variable, and the initial datum is $f(x,s)$,i.e. we consider the problem:

$$\begin{cases} w_t - D w_{xx} = 0, & x \in \mathbb{R}, \quad t > s, \\ w(x,s,s) = f(x,s), & x \in \mathbb{R}. \end{cases}$$

The solution is given by

$$w(x,t,s) = \int_{\mathbb{R}} \Gamma_D(x-y,t-s)\, f(y,s)\, dy.$$

STEP 2

By integrating, we have

$$v(x,t) = \int_0^t w(x,t,s) = \int_0^t \int_{\mathbb{R}} \Gamma_D(x-y,t-s)\, f(y,s)\, dy\, ds,$$

which is solution of **(1.31)**.

1.4.13 Theorem (existence for the non-homogeneous Cauchy problem)

Let g be a function with numerable points of discontinuity in $\mathbb{R}$, such that

$$|g(x)| \leq c\, e^{ax^2}, \quad \forall\, x \in \mathbb{R},$$

with $a, c>0$, and let f, f_t, f_x, f_{xx} be continuous and bounded functions in $\mathbb{R}\times[0,\bar{t})$, then a solution for the problem

$$\textbf{(1.32)} \quad \begin{cases} z_t - D z_{xx} = f, & in \ \mathbb{R}\times(0,T), \\ z(x,0) = g, & in \ \mathbb{R}, \end{cases}$$

is given by

$$\textbf{(1.33)} \quad \begin{aligned} z(x,t) &= u(x,t) + v(x,t) \\ &= \int_{\mathbb{R}} \Gamma_D(x-y,t)\, g(y)\, dy + \int_0^t \int_{\mathbb{R}} \Gamma_D(x-y,t-s)\, f(y,s)\, dy\, ds. \end{aligned}$$

Furthermore, (1.33) is the only solution of (1.32)

1.4.14 Theorem (principle of global maximum)

Let's consider $z(x,t)\in C(\mathbb{R}\times[0,T])$, with $z_t, z_x, z_{xx}\in C(\mathbb{R}\times(0,T))$, such that

$$\textbf{(1.34)} \quad \begin{cases} z_t - D z_{xx} \le 0, & (\ge 0) \\ z(x,t) \le C e^{Ax^2}, & (\ge -C e^{Ax^2}) \end{cases}$$

with $C, A>0$. Then

$$\textbf{(1.35)} \quad \sup_{\mathbb{R}\times[0,T]} z(x,t) \le \sup_{\mathbb{R}} z(x,0), \quad (\inf_{\mathbb{R}\times[0,T]} z(x,t) \ge \inf_{\mathbb{R}} z(x,0)).$$

Corollary

Let u be a solution of

$$\begin{cases} u_t - D u_{xx} = 0, & on \ \mathbb{R}\times(0,T), \\ u(x,0) = 0, & on \ \mathbb{R}, \end{cases}$$

where u, u_t, u_x, u_{xx} are continuous function. If $|u| \le C^{Ax^2}$, with $C, A>0$, then $u \equiv 0$.

Proof

From the *principle of global maximum,* we have

$$0 \le \inf_{\mathbb{R}} u(x,0) \le \inf_{\mathbb{R}\times[0,T]} u(x,t) \le \sup_{\mathbb{R}\times[0,T]} u(x,t) \le \sup_{\mathbb{R}} u(x,0) = 0,$$

then $u \equiv 0$.

■

1.5 Scalar reaction-diffusion equations

1.5.1 Introduction to scalar reaction-diffusion equations

Reaction-diffusion models [7] are concerned with the evolution over time of a population of individuals residing within an open spatial region $\Omega \subset \mathbb{R}^d$. Let $\boldsymbol{x} \in \Omega$ be a point and let $\{O_n\}_{n \in \mathbb{N}} \subset \Omega$ be a decreasing sequence of spatial regions containing $\boldsymbol{x}$, such that $|O_n| \to 0$, for $n \to +\infty$. Then, we can define the *density function of individuals* as

$$\textbf{(1.36)} \quad u(\boldsymbol{x}, t) = \frac{number\ of\ individuals\ within\ O_n}{|O_n|},$$

if we take n such that O_n is small relative to Ω, but large enough to contain a significant number of individuals. Then the *total population* present in a measurable sub-region $O \subset \Omega$ is given by

$$\textbf{(1.37)} \quad \int_O u(\boldsymbol{x}, t)\, d\boldsymbol{x}.$$

The population within a region may vary due to a number of factors, including:

- people entering or leaving the region (for example for a *diffusion process*);
- births or deaths caused by physical, biological, chemical reasons and so on (*reaction process*).

Historically, for example, people have always moved from highly populated regions to less populated ones, so the flow $\boldsymbol{J}$ has opposite direction to that in which u grows, i.e., *Flick's law* can be assumed to apply

$$\textbf{(1.38)} \quad \boldsymbol{J}(\boldsymbol{x}, t) = -D(\boldsymbol{x}) \nabla u(\boldsymbol{x}, t),$$

where $D(\boldsymbol{x}) \geq 0$ is said the *diffusion coefficient*.

In addition, we can denote by $f(u, \boldsymbol{x}, t)$ the *reaction rate*, which is the rate of change in population density due to biological, physical and chemical reasons.

From the above considerations, it follows

$$\frac{\partial}{\partial t} \int_O u(\boldsymbol{x}, t)\, d\boldsymbol{x} = -\int_{\partial O} \boldsymbol{J}(\boldsymbol{x}, t) \cdot \boldsymbol{n}\, d\sigma + \int_O f(u, \boldsymbol{x}, t)\, d\boldsymbol{x}.$$

Applying Gauss's theorem, we obtain

$$\int_O \frac{\partial}{\partial t} u(\boldsymbol{x}, t)\, d\boldsymbol{x} = \int_O \nabla \cdot (D(\boldsymbol{x}) \nabla u(\boldsymbol{x}, t))\, d\boldsymbol{x} + \int_O f(u, \boldsymbol{x}, t)\, d\boldsymbol{x},$$

from which we get the *reaction-diffusion equation*

$$\frac{\partial}{\partial t}u(\boldsymbol{x},t)=\nabla\cdot(D(\boldsymbol{x})\nabla u(\boldsymbol{x},t))+f(u,\boldsymbol{x},t).$$

In the case when the spatial region is *homogeneous*, the equation becomes

$$\textbf{(1.39)}\quad \frac{\partial}{\partial t}u(\boldsymbol{x},t)=D\Delta u(\boldsymbol{x},t)+f(u,\boldsymbol{x},t).$$

1.5.2 Theorem (maximum principle)

Let's take $u(\boldsymbol{x},t)\in C^2(Q_T)\cap C(\overline{Q_T})$ and let $C=C(\boldsymbol{x},t)$ be a bounded function on Q_T. If

$$u_t-D\Delta u\le Cu,\ \ in\ \ Q_T,$$

then we have that

$$\textbf{(1.40)}\quad \max_{\partial_p Q_T} u\le 0\ \Rightarrow\ \max_{\overline{Q_T}} u\le 0.$$

1.5.3 Theorem (of comparison)

Let $u,v\in C^2(Q_T)\cap C(\overline{Q_T})$ be functions such that

$$u_t-D\Delta u\ge f(u) \text{ in } Q_T,$$
$$v_t-D\Delta v\le f(v) \text{ in } Q_T.$$

Then

$$u\ge v \text{ on } \partial_p Q_T \Rightarrow u\ge v \text{ in } \overline{Q_T}.$$

1.5.4 Global existence for the Cauchy problem

For simplicity, let us suppose that $u_o\in C(\mathbb{R})$. Consider the spatially one-dimensional Cauchy problem

$$\textbf{(1.41)}\quad \begin{cases} u_t-Du_{xx}=f(u),\ in\ \mathbb{R},\\ u(x,0)=u_0(x)\in L^\infty(\mathbb{R}),\end{cases}$$

where f is a globally Lipschitzian function, i.e.

$$\exists L>0 \text{ s.t. } |f(u)-f(v)|\le L|u-v|,\ \forall u,v\in\mathbb{R}.$$

A solution for this problem, if it exists, must satisfy the *Duhamel's formula*:

$$\textbf{(1.42)}\quad u(x,t)=\int_{\mathbb{R}}\Gamma_D(x-y,t)u_0(y)\,dy+\int_0^t\int_{\mathbb{R}}\Gamma_D(x-y,t-s)f(u(y,s))\,dy\,ds.$$

Let's define the operator

$$u \to A(u) = \int_{\mathbb{R}} \Gamma_D(x-y,t)\,u_0(y)\,dy + \int_0^t \int_{\mathbb{R}} \Gamma_D(x-y,t-s)\,f(u(y,s))\,dy\,ds.$$

Let us look for a fixed point of A, i.e., a function u such that $u = A(u)$. Let's take $T > 0$, and define the Banach space

$$X = C(\mathbb{R}\times[0,T]) \cap L^\infty(\mathbb{R}\times[0,T]),$$

endowed with the norm

$$\|u\|_X = \sup_{\mathbb{R}\times[0,T]} |u(x,t)|.$$

Proceeding with the calculations we have

$$\begin{aligned}
|A(u)-A(v)| &\le \int_0^t \int_{\mathbb{R}} \Gamma_D(x-y,t-s)\,|f(u)-f(v)|\,dy\,ds \\
&\le L\int_0^t \int_{\mathbb{R}} \Gamma_D(x-y,t-s)\,|u(y,s)-v(y,s)|\,dy\,ds \\
&\le L\|u-v\|_X \int_0^t \int_{\mathbb{R}} \Gamma_D(x-y,t-s)\,dy\,ds \\
&\le LT\|u-v\|_X.
\end{aligned}$$

By choosing T such that $LT < 1$ we have that A is a contraction, then by the *Banach-Caccioppoli Theorem* we have the local existence of the solution in the interval $[0,T]$. To go to the global existence of the solution, one must iterate the procedure in the interval $[T, 2T]$ with initial datum given by $u(x,T)$ and so on.

Remark

The result can also be proved in the case when the function f is locally Lipschitzian, if one has a priori estimates for the solutions.

1.5.5 Global existence in the case of systems

Suppose we are in the case where m chemicals interact with each other with vector source term $\boldsymbol{f} = (f_1, \dots, f_m)$. We have a vector of *unknown functions* of components $u_i(\boldsymbol{x}, t)$, $i = 1, \dots, m$, each endowed with its own *diffusion coefficient* D_i, $i = 1, \dots, m$. The Cauchy problem becomes

$$\textbf{(1.43)} \quad \begin{cases} \boldsymbol{u}_t = \boldsymbol{D}\Delta\boldsymbol{u} + \boldsymbol{f}(\boldsymbol{u}), \\ \boldsymbol{u}(\boldsymbol{x},0) = \boldsymbol{u_0}(\boldsymbol{x}) \in L^\infty(\mathbb{R}^d), \end{cases}$$

where

- $\boldsymbol{u} \in \mathbb{R}^m$;
- $\boldsymbol{D}$ is the *diffusivity matrix*, which is positive semidefinite;
- $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^m$.

If $\boldsymbol{D} = diag(D_1, \dots, D_m)$, with $D_i \geq 0$, and if $\boldsymbol{f}$ is globally Lipschitzian, we can rewrite the system component by component as

$$\frac{\partial u_i}{\partial t} = D_i \Delta u_i + f_i(u_1, \dots, u_m), \quad i = 1, \dots, m.$$

Under these assumptions, the existence of a solution is guaranteed as in the scalar case.

1.6 Reaction-diffusion models for a biological species

1.6.1 Reaction rate

Several choices can be made for the *reaction rate* [7] in these models:

1. *Malthaus linear growth*, in which

   $$f = r u, \tag{1.44}$$

   where

   - $r = b - d$ is called the *growth rate*;
   - b is called the *birth rate* (constant);
   - d is called *death rate* (constant).

   In the spatially homogeneous case we have $u = u(t)$ and the equation becomes

   $$u_t = r u,$$

   which has solution

   $$u(t) = u_0 e^{rt},$$

   where u_0 represents the *initial population*. If $r > 0$ there is an exponential growth, otherwise there is an exponential decrease.

2. *logistic growth*, in which

   $$f = r u \left(1 - \frac{u}{k}\right), \tag{1.45}$$

where k is the *carrying capacity* of the environment, accounting for resource limitation and competition among individuals.

In the spatially homogeneous case we have that

$$u(t)=u_0\frac{k\,e^{rt}}{k+u_0(e^{rt}-1)}.$$

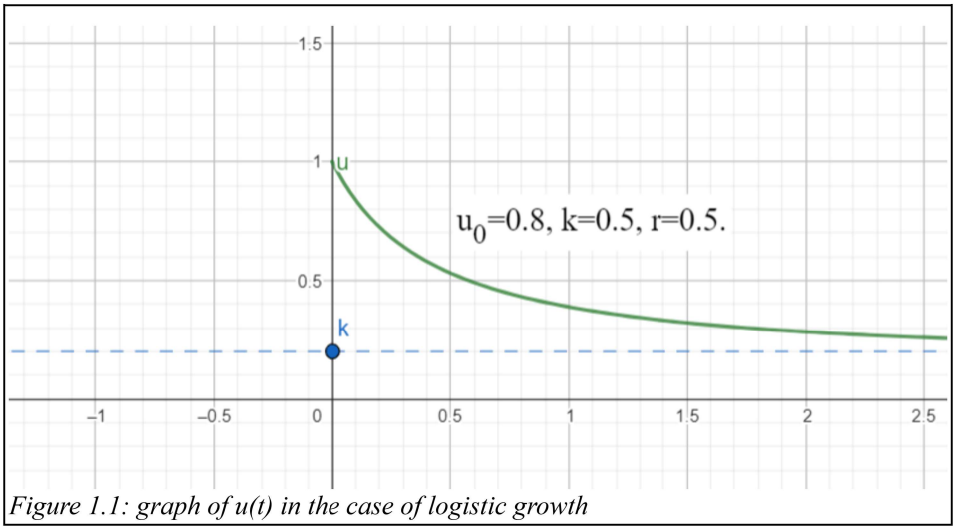

Figure 1.1: graph of u(t) in the case of logistic growth

Therefore, if $r>0$, whatever its initial value, the population tends to reach the value of k, see Figure 1.1.

We observe that there are two equilibrium states:

- $u=0$, which is unstable;
- $u=k$, which is stable and attractive.

3. *growth with an Allee effect*, which takes into account that the presence of a small population can be a disadvantage because, for example, mating may be difficult. A typical example is given by

$$\textbf{(1.46)}\quad f=r\,u\left(1-\frac{u}{k}\right)(u-c),$$

where c is the *constant of deficiency*.

1.6.2 Fisher–Kolmogorov–Petrovsky–Piskunov Equation

It is a reaction-diffusion equation in which the logistic growth rate is used as the reaction rate:

$$\textbf{(1.47)}\quad u_t=r\,u(1-u)+D\,u_{xx},$$

where $r, D>0$, $k=1$.

The importance of this equation is primarily didactic in that it represents the prototype of equations that admit travelling wave solutions and is useful for showing many standard techniques that are used for the analysis of models with diffusive scattering. By taking $t^*=r\,t$, $x^*=x\sqrt{rD}$, and omitting the asterisks, **(1.47)** can be rewritten as

$$\textbf{(1.48)} \quad u_t=u(1-u)+u_{xx}.$$

1.6.3 Definition of monostable function

A reaction function $f\in C^1(\mathbb{R})$ is said to be *monostable* if

- $f(u)>0$, *if* $u\in(0,1)$,
- $f(u)<0$, *if* $u\notin(0,1)$.

Remark

The function $f(u)=u(1-u)$ is monostable.

1.6.4 Proposition (reaction-diffusion equation with monostable reaction rate)

For a reaction-diffusion equation with monostable reaction rate, we have that every constant $M>1$ is supersolution. Moreover $u\equiv 0$ and $u\equiv 1$ are solutions.

Proof

Since $M>1$, through a direct calculation we have

$$\frac{\partial u}{\partial t}-f(u)-\frac{\partial^2 u}{\partial x^2}=\frac{\partial M}{\partial t}-f(M)-\frac{\partial^2 M}{\partial x^2}=-f(M)>0.$$

■

1.6.5 Properties (boundedness of the solution)

Let us consider the problem

$$\begin{cases} u_t-Du_{xx}=f(u), \\ u(x,0)=u_0(x). \end{cases}$$

If $0\le u_0(x)\le M$, with $M>1$, then $0\le u(x,t)\le M$, $\forall\, x\in\mathbb{R}$, $\forall\, t>0$.

1.6.6 Stationary solutions

We want to start studying *stationary solutions*, that is, solutions such that $u_t = 0$, from which we get

$$u_{xx} + f(u) = 0.$$

Multiplying by u_x we get

$$u_{xx} u_x + f(u) u_x = 0.$$

We now define the potential energy function F, such that

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

from which

$$\frac{\partial F}{\partial x} = \frac{\partial F}{\partial u} \cdot \frac{\partial u}{\partial x} = f(u) u_x.$$

Integrating $u_{xx} u_x + f(u) u_x = 0$ we have

$$\int_{\mathbb{R}} u_{xx} u_x + f(u) u_x \, dx = \int_{\mathbb{R}} 0 \, dx,$$

that is, $\frac{1}{2}(u_x)^2 + F(u) = K$, with $K \in \mathbb{R}$, from which we get

$$\left| \frac{\partial u}{\partial x} \right| = \sqrt{2(K - F(u))}.$$

Then

$$\pm \int_{u(x_0)}^{u(x)} \frac{1}{\sqrt{2(K - F(u))}} du = \int_{x_0}^{x} dx = x - x_0.$$

Drawing the graph of $F(u)$, for example in the case of the FKPP equation, we see that the stationary solutions are in biunivocal correspondence, up to transations and reflections, with the horizontal segments $v = K$ in the plane (u, v). At the values $K_0 = F(0)$ and $K_1 = F(1)$ we have the equilibrium states $u = 0$ and $u = 1$. As we can see in Figure 1.2, the solutions corresponding to segments A, C and E are unbounded, while those corresponding to segments B and D are finite. In addition, the solutions corresponding to segments A and E have minima given by the point of intersection with the graph of the function F, while, for $x \to \pm\infty$, they diverge. The solution corresponding to segment C tends to 1 for $x \to -\infty$, and diverges for $x \to +\infty$, or viceversa. Segment B has negative minimum and tends to 1 for $x \to \pm\infty$. While D has minimum and maximum.

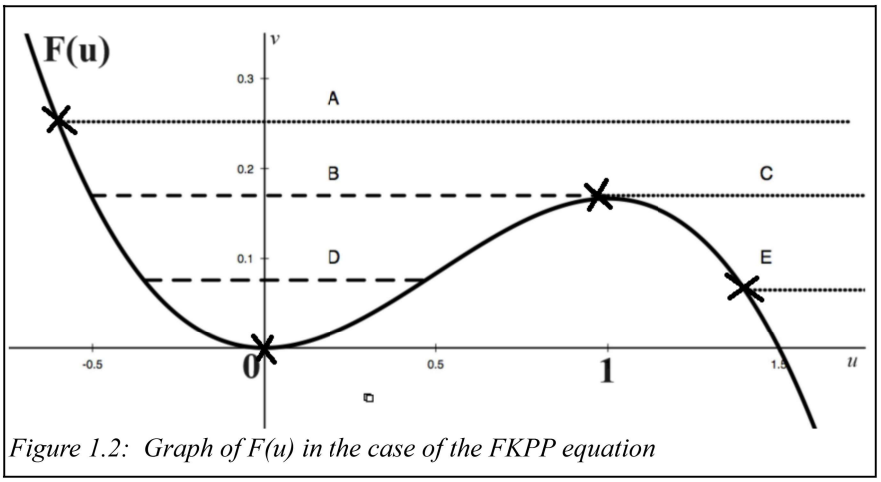

Figure 1.2: Graph of F(u) in the case of the FKPP equation

We remark that the only bounded non-negative solutions are $u \equiv 0$ and $u \equiv 1$.

1.7 Travelling waves

1.7.1 Definition of travelling waves

A solution $u = u(x,t)$ of an evolution equation is called a *travelling wave* [7] if there exists a function U, called the *wave profile*, and a constant c called the *wave velocity*, such that $u(x,t) = U(x - ct)$. We want to determine if there exist travelling waves of reaction-diffusion equations, with asymptotic conditions of the type:

$$U(+\infty) = u^+, \quad U(-\infty) = u^-, \text{ with } u^+, u^- \text{ such that } f(u^+) = f(u^-) = 0.$$

In this regard, one has the

1.7.2 Theorem (reaction-diffusion equation with monostable reaction rate admits travelling waves)

Let f be a monostable function such that

$$f(u) < f'(0)u, \quad u \in (0,1).$$

Then the equation

$$u_t = u_{xx} + f(u)$$

admits travelling waves such that

$$U(-\infty) = 1, \quad U(+\infty) = 0, \quad U(z) \geq 0, \forall z \in \mathbb{R},$$

if and only if

$$\textbf{(1.49)} \quad c \geq c_{min} := 2\sqrt{f'(0)}.$$

1.7.3 Logistic growth term

It is easy to see that the logistic growth term satisfies the required condition. In this case, in terms of dimensional quantities, we have that

$$\textbf{(1.50)} \quad c_{min} = \sqrt{r\,D}.$$

In addition, the conditions of $u(x,0)$ for the FKPP solution to evolve into a travelling wave solution have been widely studied. It was shown that if $\exists\, x_0, x_1 \in \mathbb{R}$, with $x_0 < x_1$, such that

$$u(x,0) = u_0(x) = \begin{cases} 1, & if \quad x \leq x_0, \\ 0, & if \quad x \geq x_1, \end{cases}$$

with $u_0(x)$ continuous function, then $u(x,t)$ evolves to a travelling wave solution $U(z)$ such that

$$\textbf{(1.51)} \quad z = x - 2t, \quad c_{min} = 2.$$

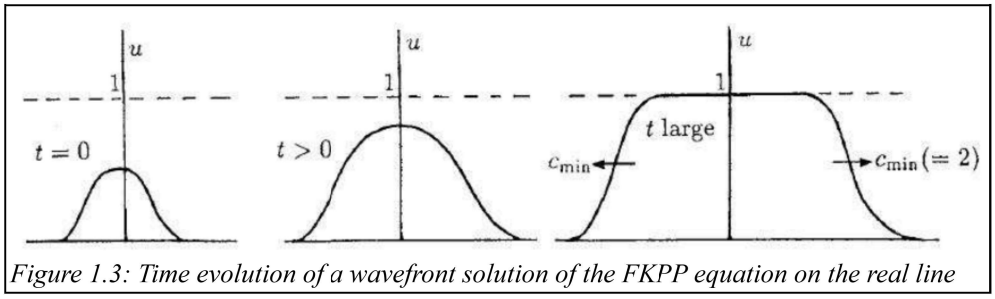

Figure 1.3: Time evolution of a wavefront solution of the FKPP equation on the real line

1.7.4 Asymptotic approximation

Let us consider the reaction-diffusion equation

$$\textbf{(1.52)} \quad u_t = u_{xx} + u(1-u).$$

By imposing $u = U(z)$, with $z = x - ct$, we obtain

$$\textbf{(1.53)} \quad U'' + c\,U' + U(1-U) = 0.$$

By the previous theorem, it admits a monotone solution such that

$$U(-\infty) = 1, \quad U(+\infty) = 0, \quad U(z) \geq 0, \forall\, z \in \mathbb{R}.$$

Furthermore, it can be shown that an approximation to this solution, in the smallness parameter $\epsilon = \frac{1}{c^2}$, is of the form

$$\textbf{(1.54)} \quad U(z)=\frac{1}{1+e^{z/c}}+\frac{1}{c^2}\cdot\frac{e^{z/c}}{1+e^{z/c}}\log\left[\frac{4\,e^{z/c}}{(1+e^{z/c})^2}\right]+O\left(\frac{1}{c^2}\right),$$

with $c\geq c_{min}=2$.

If we want to give an estimate of the slope of the wave, we must calculate

$$max|U'(z)|,$$

which is reached at the point where $U''=0$, that is, for $z=0$. Continuing with the calculations it is possible to obtain that an estimate for the slope is given by

$$\textbf{(1.55)} \quad s\approx U'(0)\approx-\frac{1}{4\,c}, \quad c\geq 2,$$

which shows that the higher the speed, the lower the wave front steepness will be. An estimate of the length of the wave is given by

$$\textbf{(1.56)} \quad L\approx\frac{1}{s}=4\,c.$$

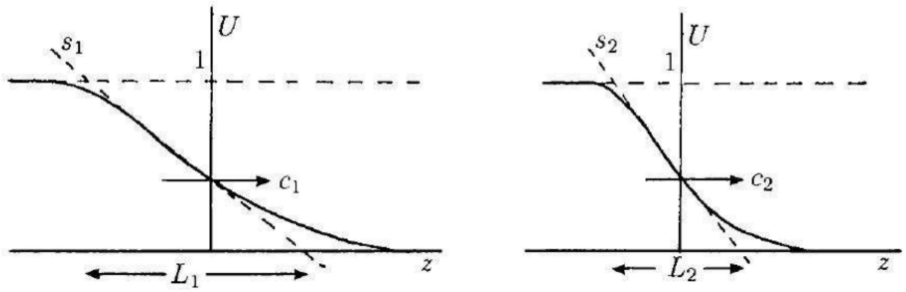

Figure 1.4: Representation of the slope of the wave and a measure of its width L for travelling wave solutions of the FKPP equation

1.7.5 Stability

We want to study how the travelling wave solutions behave with respect to small perturbations. Let us consider the equation

$$\textbf{(1.57)} \quad u_t=u_{xx}+u(1-u),$$

and we make the transformation to the system travelling with the wave

$$u(x,t)=u(z,t), \text{ with } z=x-c\,t,$$

with an abuse of notation. Substituting in **(1.57)** yields

$$\textbf{(1.58)} \quad u_t=c\,u_z+u_{ZZ}+u(1-u), \text{ with } c\geq 2.$$

Let $u_c(z)$ denote the travelling wave solution $U(z)$ of **(1.53)**, and let us consider a small compactly supported perturbation of it

$$u(z,t)=u_c(z)+\varepsilon\,v(z,t), \quad 0<\varepsilon\ll 1.$$

Substituting in **(1.58)** we get the equation

$$v_t = (1 - 2u_c)v + c\,v_z + v_{zz}.$$

It is possible to show all the solutions of the type

$$v(z,t) = g(z)e^{-\lambda t},$$

have λ positive, apart from the solution with $\lambda = 0$ which represents a small translation of u_c, and g is a continuous function. Therefore, for $\lambda > 0$,

$$\lim_{t \to +\infty} v(z,t) = g(z)e^{-\lambda t} = 0,$$

that is, $u_c(z)$ is stable for small perturbations of the type under consideration.

1.8 Systems of reaction-diffusion equations

1.8.1 Definition of conservative system

Given $\boldsymbol{u} : (0, T) \to \mathbb{R}^m$, the ordinary system of differential operations

$$\textbf{(1.59)} \qquad \dot{\boldsymbol{u}}(t) = \boldsymbol{f}(\boldsymbol{u})$$

is said to be *conservative* [7] if there exists a real valued function $g(\boldsymbol{u})$ such that

$$\dot{g}(\boldsymbol{u}) = \frac{\partial g}{\partial u_i}\dot{u}_i = \frac{\partial g}{\partial u_i} f_i = 0$$

1.8.2 Lotka – Volterra system

Let us consider the prey-predator system

$$\textbf{(1.60)} \qquad \begin{cases} \dfrac{du}{dt} = u(a - bv), \\ \dfrac{dv}{dt} = v(cu - e), \end{cases}$$

where:

- u = prey population;
- v = predator population;
- a = preys growth rate;
- e = predators decrease rate;
- b = predation rate;
- c = coefficient of prey-predator conversion.

We can adimensionalize the system by placing

$$\hat{t}=at\,,\quad \hat{u}=\frac{c}{e}u\,,\quad \hat{v}=\frac{b}{a}v\,,\quad \alpha=\frac{e}{a},$$

and we obtain

$$\textbf{(1.61)}\qquad \begin{cases}\dfrac{du}{dt}=u(1-v)=f_1(u,v),\\ \dfrac{dv}{dt}=\alpha v(u-1)=f_2(u,v),\end{cases}$$

where, for simplicity, we have dropped the hats.

To study the equilibrium points, let us set

$$\frac{du}{dt}=\frac{dv}{dt}=0\,,$$

whence

$$\begin{cases}u(1-v)=0\,,\\ \alpha\, v(u-1)=0.\end{cases}$$

The equilibrium points are $O=(0,0),\ \ Q=(1,1).$

The Jacobian matrix is

$$J=\begin{pmatrix}\dfrac{\partial f_1}{\partial u} & \dfrac{\partial f_1}{\partial v}\\ \dfrac{\partial f_2}{\partial u} & \dfrac{\partial f_2}{\partial v}\end{pmatrix}=\begin{pmatrix}1-v & -u\\ \alpha v & \alpha(u-1)\end{pmatrix}$$

Evaluated in 0, it gives

$$J(0)=\begin{pmatrix}1 & 0\\ 0 & -\alpha\end{pmatrix},$$

which is diagonal, then the eigenvalues are $\lambda=-\alpha\,,\ \ \lambda=1\,,$ and thus it is an unstable equilibrium point (*saddle point*).

Evaluated in Q, the Jacobian is

$$J(Q)=\begin{pmatrix}0 & -1\\ \alpha & 0\end{pmatrix},$$

whence

$$P_\lambda(J)=|J-\lambda I|=\begin{vmatrix}-\lambda & -1\\ \alpha & -\lambda\end{vmatrix}=\lambda^2+\alpha.$$

The eigenvalues are $\pm i\sqrt{\alpha}\,,$ so we cannot apply the Hartman-Grobman Theorem, see [8].

Consider then the system

$$\begin{cases} \frac{du}{dt} = u(1-v) = f_1(u,v), \\ \frac{dv}{dt} = \alpha v(u-1) = f_2(u,v). \end{cases}$$

By formally dividing the first by the second we have

$$\frac{du}{dv} = \frac{u(1-v)}{\alpha v(u-1)},$$

from which

$$\alpha \frac{u-1}{u} du = \frac{1-v}{v} dv,$$

which when integrated gives us

$$g(u,v) := \alpha(u - \log u) + v - \log v = cost.$$

We note that $\dot{g} = 0$, so the system is conservative.

Since

$$H(g) = \begin{pmatrix} \frac{\partial^2 g}{\partial u^2} & \frac{\partial^2 g}{\partial u \partial v} \\ \frac{\partial^2 g}{\partial v \partial u} & \frac{\partial^2 g}{\partial v^2} \end{pmatrix} = \begin{pmatrix} \frac{\alpha}{u^2} & 0 \\ 0 & \frac{1}{v^2} \end{pmatrix}$$

and $|H(g)| = \frac{\alpha}{u^2 v^2} > 0$ then g is convex over all $\mathbb{R}^2 \setminus \{0\}$.

Moreover

$$\begin{cases} \frac{\partial g}{\partial u} = \alpha(1 - \frac{1}{u}) = 0 \\ \frac{\partial g}{\partial v} = (1 - \frac{1}{v}) = 0 \end{cases}$$

The system is zero only at the point $Q = (1,1)$, which is a global minimum point for g and it holds

$$g(1,1) = \alpha + 1$$

One has

$$\lim_{u \to +\infty} g(u,v) = +\infty,$$

$$\lim_{u \to 0} g(u,v) = +\infty,$$

similarly for $v \to +\infty, 0^+$.

It is obtained that for $k>\alpha+1$, the contour lines $\{(u,v)|g(u,v)=k\}$, are closed regular curves containing $(1,1)$. The solutions are then periodic and the period depends on k.

Furthermore, we have that, if $T(k)$ is the period of the orbit, then

$$\int_0^{T(k)} \frac{\dot{u}}{u} dt = \ln(u(t))\Big|_0^{T(k)} = 0,$$

$$\int_0^{T(k)} \frac{\dot{v}}{v} dt = \ln(v(t))\Big|_0^{T(k)} = 0,$$

from which

$$0 = \int_0^T \frac{f_1}{u} = \int_0^T (1-v)\,dt = T - \int_0^T v(t)\,dt,$$

$$0 = \int_0^T \frac{f_2}{v} = \int_0^T (u-1)\,dt = T - \int_0^T v(t)\,dt.$$

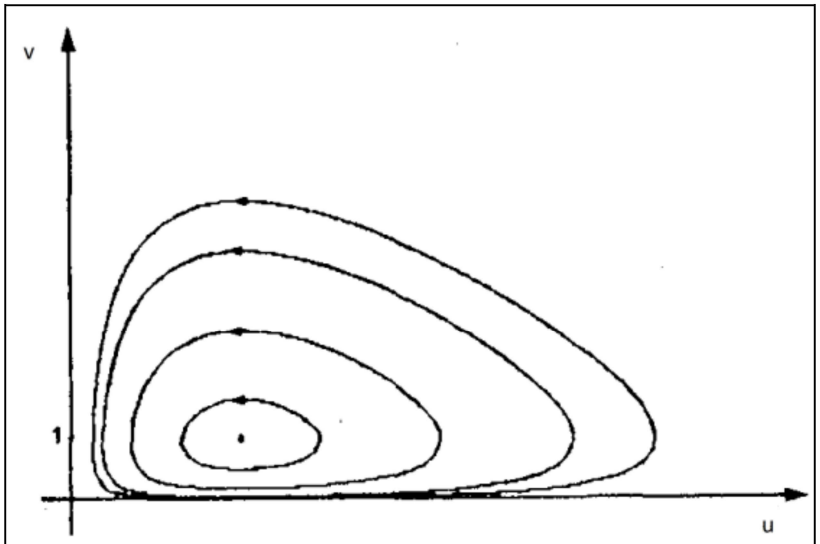

Figure 1.5: Phase portrait for the Lotka-Volterra system

1.8.3 Structural instability

Let us consider small perturbations of the type

$$\begin{cases} \frac{du}{dt} = u(1-v) - \varepsilon u^2, \\ \frac{dv}{dt} = \alpha v(u-1). \end{cases}$$

The equilibrium state, in which there is coexistence between the two species, is given by $(1,\ 1-\varepsilon)$which is a stable spiral. From a biological point of view, this means that a small perturbation can remove the oscillations, which is not very

realistic. This phenomenon, from a mathematical point of view, is called "structural instability".

1.8.4 Theorem (Diffusion)

Consider the Lotka-Volterra system with diffusion

$$\textbf{(1.62)} \qquad \begin{cases} u_t = u(1-v) - D\,\Delta u, \\ v_t = \alpha v(u-1) + D\,\Delta v, \\ \dfrac{\partial u}{\partial n} = \dfrac{\partial v}{\partial n} = 0, (\boldsymbol{x}, t) \in \partial\Omega\, x(0, T], \quad (Neumann) \end{cases}$$

with $\Omega \subseteq \mathbb{R}^n$ bounded, $\boldsymbol{n}$ external normal.

It can be shown that the system tends to a homogeneous spatial state for $t \to +\infty$.

Proof

Let us consider the initial conditions

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}), \quad v(\boldsymbol{x},0) = v_0(\boldsymbol{x}), \quad \boldsymbol{x} \in \overline{\Omega}$$

and let

$$g(u, v) = \alpha(u - \log u) + v - \log v$$

be the "energy" of system as before. Obviously we have

$$g_t = \alpha\left(u_t - \frac{u_t}{u}\right) + v_t - \frac{v_t}{v},$$

$$\Delta g = \alpha\left(\Delta u - \frac{\Delta u}{u} + \frac{|\nabla u|^2}{u^2}\right) + \Delta v - \frac{\Delta v}{v} + \frac{|\nabla v|^2}{v^2},$$

from which, by a direct calculation, one obtains

$$g_t - D\,\Delta g \le 0,$$

which means that the energy is dissipated by diffusion.

The boundary conditions for g are

$$\frac{\partial g}{\partial n} = \alpha\left(\frac{\partial u}{\partial n} - \frac{1}{u}\frac{\partial u}{\partial n}\right) + \frac{\partial v}{\partial n} - \frac{1}{v}\frac{\partial v}{\partial n} = 0$$

while the initial conditions read

$$g(\boldsymbol{x},0) = \alpha(u_0(\boldsymbol{x}) - \log u_0(\boldsymbol{x})) + v_0(\boldsymbol{x}) - \log v_0(\boldsymbol{x}) =: g_0(\boldsymbol{x}).$$

Let us consider now the total energy

$$G(t) = \int_\Omega g(\boldsymbol{x}, t)\, d\,\boldsymbol{x},$$

then

$$\dot{G}(t)=\int_{\Omega} g_t\, d\boldsymbol{x}=\int_{\Omega} D\Delta g-\alpha D\frac{|\nabla u|^2}{u^2}-D\frac{|\nabla v|^2}{v^2}\, d\boldsymbol{x}$$
$$\overset{Green}{=}\int_{\partial\Omega} D\frac{\partial g}{\partial n}-\int_{\Omega}\alpha D\frac{|\nabla u|^2}{u^2}+D\frac{|\nabla v|^2}{v^2}\, d\boldsymbol{x}\le 0,$$

so $G(t)$ is decreasing.

Furthermore, since $z-\log z\ge 1$, $\forall z>0$, we have

$$g(x,t)=\alpha(u-\log u)+v-\log v\ge\alpha+1.$$

Therefore

$$G(t)=\int_{\Omega} g(\boldsymbol{x},t)\, d\boldsymbol{x}\ge(\alpha+1)|\Omega|,$$

from which

$$\lim_{t\to+\infty} G(t)$$

is finite. Consequently

$$\lim_{t\to+\infty}\dot{G}(t)=0,$$

implies that

$$\nabla u\to 0,\nabla v\to 0,\ for\ t\to+\infty.$$

So the system goes toward a spatially homogeneous situation.

■

1.8.5 Definition (Boundary operator)

The operator $\boldsymbol{Bu}$ such that

$$\textbf{(1.63)}\quad (\boldsymbol{Bu})_i=c_i(\boldsymbol{x},t)u_i(\boldsymbol{x},t)+d_i(\boldsymbol{x},t)\frac{\partial u_i}{\partial n_i}(\boldsymbol{x},t),$$

is called the *boundary operator*, with $c_i\ge 0, d_i\ge 0, c_i^2+d_i^2>0$ and $\frac{\partial}{\partial n_i}$ a certain external derivative. We observe that $\boldsymbol{B}$ is a diagonal operator.

1.8.6 Definition (Invariant set)

An *invariant set* for systems of the type

$$\textbf{(1.64)}\quad \boldsymbol{u}_t=\boldsymbol{f}(\boldsymbol{u})+D\Delta\boldsymbol{u}$$

is a set $\Sigma\subseteq\mathbb{R}^m$ such that if $\boldsymbol{u}$ is solution of the system with initial values in Σ and boundary values in $\boldsymbol{Bu}\in\boldsymbol{c}\,\Sigma:=\{(c_1v_1,\dots,c_nv_n)^T|\boldsymbol{v}\in\Sigma\}$, then

$$\boldsymbol{u}(\boldsymbol{x},t)\in\Sigma,\ \forall\boldsymbol{x}\in\Omega,\ \forall t\in[0,T).$$

1.8.7 Definition (Almost monotonicity)

- A function $\boldsymbol{f}(\boldsymbol{u})$ is said to be *almost monotonic increasing* if every f_i is increasing with respect to every u_j, $j \neq i$.
- A function $\boldsymbol{f}(\boldsymbol{u})$ is said to be *mixed almost monotonic* if every f_i is monotonic with respect to every u_j, $j \neq i$.

Notation

In the following we will consider systems of the type

$$\textbf{(P)} \quad \begin{cases} N\boldsymbol{u} = L\boldsymbol{u} - \boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{u}_t - \boldsymbol{D}\Delta\boldsymbol{u} - \boldsymbol{f}(\boldsymbol{u}) = 0, \ \ in \ \ \Omega \times (0,T), \\ \boldsymbol{u}(\boldsymbol{x},0) = \boldsymbol{u}_0(\boldsymbol{x}), \ \ on \ \ \overline{\Omega} \times \{0\}, \\ \boldsymbol{B}\boldsymbol{u} = \boldsymbol{b}(\boldsymbol{x},t), \ \ on \ \ \partial\Omega \, x \, (0,T], \end{cases}$$

where

- **D** = diagonal diffusion matrix;
- **B** = diagonal boundary operator.

1.8.8 Theorem (Comparison generalized)

Let $\underline{\boldsymbol{u}}$ and $\overline{\boldsymbol{u}}$ be a subsolution and a supersolution of the system **(P)**. Let $\boldsymbol{f}$ be globally Lipschitz and almost monotonically increasing, and let Ω satisfy the inner sphere property.

1. The *weak comparison theorem* states that

   $$\textbf{(1.65)} \quad \underline{\boldsymbol{u}} \leq \overline{\boldsymbol{u}} \ \ in \ \ \overline{\Omega} \times [0,T],$$

   where the inequalities have to be intended componentwise.
2. The *strong comparison theorem* states that

   $$\textbf{(1.66)} \quad \underline{u}_i \leq \overline{u}_i, \forall i, \ \ in \ \ \Omega \times (0,T],$$

   or

   $$\underline{u}_i = \overline{u}_i, \ \ in \ \ \overline{\Omega} \times [0,t'] \text{ for a certain } t' \leq T.$$

Proof

1. Let y be dependent on the parameter ε, in such that $y \to 0$, if $\varepsilon \to 0$. Let $\boldsymbol{Y} = (y, \ \ldots, \ y)$ be a solution of the problem

   $$\begin{cases} \dot{y} = k\|\boldsymbol{Y}\| = k\sqrt{n}\, y \\ y(0) = \varepsilon, \end{cases}$$

   i.e. $y(t) = \varepsilon^{k\sqrt{m}t}$, where k is the Lipschitz constant of $\boldsymbol{f}$.

Let us consider the vector

$$\boldsymbol{v}=\underline{\boldsymbol{u}}-\boldsymbol{Y}$$

and show that it is a subsolution of problem $L\boldsymbol{u}=\boldsymbol{f}(\boldsymbol{u})$.

We have

$$\begin{aligned}
L_i v_i &\overset{\text{def}}{\equiv} \frac{\partial}{\partial t} v_i - D_i \Delta v_i = \frac{\partial}{\partial t}\underline{u_i} - \dot{y} - D_i \Delta \underline{u_i} \\
&\le f_i(\underline{\boldsymbol{u}}) - k\|\boldsymbol{Y}\| = f_i(\boldsymbol{v}+\boldsymbol{Y}) - f_i(\boldsymbol{v}) + f_i(\boldsymbol{v}) - k\|\boldsymbol{Y}\| \\
&\le k\|(\boldsymbol{v}+\boldsymbol{Y}-\boldsymbol{v})\| + f_i(\boldsymbol{v}) - k\|\boldsymbol{Y}\| \\
&= k\|\boldsymbol{Y}\| + f_i(\boldsymbol{v}) - k\|\boldsymbol{Y}\| = f_i(\boldsymbol{v}).
\end{aligned}$$

Now

$$\boldsymbol{v}(\boldsymbol{x},0)=\underline{\boldsymbol{u}}(\boldsymbol{x},0)-\boldsymbol{Y}(\boldsymbol{0})\le \boldsymbol{u_0}(\boldsymbol{x})-(\varepsilon,\dots,\varepsilon)^T \le \boldsymbol{u_0}(\boldsymbol{x}),$$

$$\boldsymbol{B}\boldsymbol{v}=\boldsymbol{B}(\underline{\boldsymbol{u}}-\boldsymbol{Y})=\boldsymbol{B}\underline{\boldsymbol{u}}-\boldsymbol{c}\,\boldsymbol{Y}\le \boldsymbol{b}(\boldsymbol{x},t).$$

Then $\boldsymbol{v}=\underline{\boldsymbol{u}}-\boldsymbol{Y}$ is a subsolution of $\boldsymbol{L}\boldsymbol{u}=\boldsymbol{f}(\boldsymbol{u})$.

Let now be

$$\boldsymbol{w}=\overline{\boldsymbol{u}}-\boldsymbol{v}=\overline{\boldsymbol{u}}-\underline{\boldsymbol{u}}+\boldsymbol{Y},$$

we have

$$\begin{aligned}
&\boldsymbol{w}(\boldsymbol{x},0)=\overline{\boldsymbol{u}}(x,0)-\underline{\boldsymbol{u}}(\boldsymbol{x},0)+\boldsymbol{Y}(0)\ge \boldsymbol{Y}(0)>0, \\
&\boldsymbol{B}\boldsymbol{w}=\boldsymbol{B}\overline{\boldsymbol{u}}-\boldsymbol{B}\boldsymbol{v}+\boldsymbol{c}\,\boldsymbol{Y}\ge \boldsymbol{b}-\boldsymbol{b}+\boldsymbol{c}\,\boldsymbol{Y}\ge 0.
\end{aligned}$$

Since $\boldsymbol{w}(\boldsymbol{x},0)>0$ then $\exists\,\tau\in[0,T]$ s.t.

$$\boldsymbol{w}(\boldsymbol{x},t)\ge 0 \quad in \quad [0,\tau],$$

then

$$\begin{aligned}
L_i w_i &= L_i\overline{u}_i - L_i\underline{u}_i + L_i y = L_i\overline{u}_i - L_i v_i \\
&\ge f_i(\overline{u}) - f_i(v) = f_i(\overline{u}_1 \dots \overline{u}_i \dots \overline{u}_m) - f_i(v_1 \dots v_i \dots v_m) \\
&\ge -K(\overline{u}_i - v_i) = -K w_i,
\end{aligned}$$

which means that $\boldsymbol{w}$ is a supersolution. Then, by the principle of the scalar maximum $w_i>0$. If $\tau\in[0,T]$ and $\boldsymbol{w}>0 \quad in \quad [0,\tau]$ then $\tau=T$, i. e.

$$\boldsymbol{w}>0 \quad in \quad [0,T],$$

from which

$$\overline{\boldsymbol{u}}>\underline{\boldsymbol{u}}-\boldsymbol{Y},$$

and, since $\boldsymbol{Y}\to 0,\ if\ \varepsilon\to 0$, we have

$$\overline{\boldsymbol{u}}\ge\underline{\boldsymbol{u}} \quad in \quad \overline{Q_T}.$$

2. By placing $N_i(\boldsymbol{u})=L_i u_i - f_i(\boldsymbol{u}),\ \ \boldsymbol{z}=\overline{\boldsymbol{u}}-\underline{\boldsymbol{u}}$, one obtains

$$
\begin{aligned}
0 \le N_i(\overline{\boldsymbol{u}}) - N_i(\underline{\boldsymbol{u}}) &= L_i(\overline{u_i}) - f_i(\overline{\boldsymbol{u}}) - L_i(\underline{u_i}) + f_i(\underline{\boldsymbol{u}}) \\
&= L_i z_i - [f_i(\overline{u_1} \dots \overline{u_m}) - f_i(\underline{u_1} \dots \underline{u_m})] \\
&\le L_i z_i - [f_i(\overline{u_1} \dots \overline{u_i} \dots \overline{u_m}) - f_i(\underline{u_1} \dots \underline{u_i} \dots \underline{u_m})] \\
&\le L_i z_i - h_i(\overline{u_i} - \underline{u_i}) = L_i z_i - h_i z_i .
\end{aligned}
$$

By the theorem of the strong scalar maximum it follows that the thesis holds true

■

1.8.9 Corollary (Uniqueness)

Under the assumptions of the comparison theorem, we have that the problem

$$
\textbf{(1.67)} \quad \begin{cases} \boldsymbol{N u = L u - f(u) = u_t - D \Delta u - f(u) = 0}, \ \ on \ \ \Omega \times (0,T), \\ \boldsymbol{u(x,0) = u_0(x)}, \ \ on \ \ \overline{\Omega} \times \{0\}, \\ \boldsymbol{Bu = b(x,t)}, \ \ on \ \ \partial\Omega \times [0,T], \end{cases}
$$

has a unique solution.

Proof

Suppose that $\boldsymbol{v}, \boldsymbol{w}$ are both solutions, then by the comparison theorem we have $\boldsymbol{v} \ge \boldsymbol{w}$ and $\boldsymbol{w} \ge \boldsymbol{v}$, i.e.

$$
\boldsymbol{v = w}.
$$

■

1.8.10 Corollary (Invariant set)

Under the assumptions of the comparison theorem, let $\boldsymbol{a}, \boldsymbol{b}$ be two vectors s.t. $-\infty < \boldsymbol{a} < \boldsymbol{b} < +\infty$ with $\boldsymbol{f(a)} \ge 0$, $\boldsymbol{f(b)} \le 0$, then

$$
\textbf{(1.68)} \quad \Sigma = \{\boldsymbol{u} \ | \boldsymbol{a} \le \boldsymbol{u} \le \boldsymbol{b}\}
$$

is an invariant set for **(1.67)**.

Proof

If $\boldsymbol{f(a)} \ge 0$, then $\boldsymbol{N(a)} = -\boldsymbol{f(a)} \le 0$, i. e. $\boldsymbol{a}$ is a subsolution of **(1.67)**.

If $\boldsymbol{f(b)} \le 0$, then $\boldsymbol{N(b)} = -\boldsymbol{f(b)} \ge 0$, i. e. $\boldsymbol{b}$ is a supersolution of **(1.67)**.

Let $\boldsymbol{u(x,t)}$ be a solution of **(1.67)** then we have that if

$$
\boldsymbol{a \le u(x,0) = u_0(x) \le b},
$$

then

$$\boldsymbol{a} \le \boldsymbol{u}(\boldsymbol{x}, t) \le \boldsymbol{b}, \quad \forall t \in (0, T),$$

hence the thesis.

■

1.8.11 Theorem (Global existence)

Under the assumptions of the comparison theorem, let $\boldsymbol{a}, \boldsymbol{b}$ s.t. $-\infty < \boldsymbol{a} < \boldsymbol{b} < +\infty$, $\boldsymbol{f}(\boldsymbol{a}) \ge 0$ and $\boldsymbol{f}(\boldsymbol{b}) \le 0$, then the problem

$$\textbf{(1.69)} \quad \begin{cases} \boldsymbol{N}\boldsymbol{u} = \boldsymbol{L}\boldsymbol{u} - \boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{u}_t - \boldsymbol{D}\Delta\boldsymbol{u} - \boldsymbol{f}(\boldsymbol{u}) = 0, & in \quad \Omega \times (0, T), \\ \boldsymbol{u}(\boldsymbol{x}, 0) = \boldsymbol{u}_0(x), & in \quad \overline{\Omega} \times 0, \\ \boldsymbol{B}\boldsymbol{u} = \boldsymbol{b}(\boldsymbol{x}, t), & in \quad \partial\Omega \times (0, T], \end{cases}$$

has a solution on $[0, T]$.

Proof (Sketch)

Let $\{\boldsymbol{v}^n\}, \{\boldsymbol{w}^n\}$ be defined as

$$\boldsymbol{v}^0 = a, \quad (L_i + k) v_i^n = f_i(\boldsymbol{v}^{n-1}) + k\, v_i^{n-1},$$
$$\boldsymbol{w}^0 = b, \quad (L_i + k) w_i^n = f_i(\boldsymbol{w}^{n-1}) + k\, w_i^{n-1}.$$

By induction, it can be shown that $\boldsymbol{v}^{n+1} \ge \boldsymbol{v}^n$ and $\boldsymbol{w}^{n+1} \le \boldsymbol{w}^n$.

Then we have that $\{\boldsymbol{v}^n\}, \{\boldsymbol{w}^n\}$ are monotone sequences and are also bounded, so they converge; moreover, the limit is equal. We have constructed two monotone sequences that both converge to the solution of the problem, which can be calculated numerically.

■

1.8.12 Definition (Closed operator)

Let $A : D(A) \subseteq H \to H$ be an operator, with H a Hilbert space. It will be called a *closed operator* if

$$\forall x_n \in D(A) \quad s.t. \quad x_n \to x \quad and \quad A x_n \to y,$$

it holds that

$$x \in D(A) \quad and \quad A x_n \to y = A x.$$

1.8.13 Corollary (of the comparison theorem)

Let $\underline{\boldsymbol{f}} : \mathbb{R}^m \to \mathbb{R}^m$ be almost monotonically increasing and globally Lipschitz such that

$$\underline{f}(v) \le f(v), \quad \forall v \in \mathbb{R}^m.$$

Let $\underline{u}$ be a subsolution of **(P)**, with f substituted by $\underline{f}$ (let us recall this problem $(\underline{P})$), and $\bar{v}$ a supersolution of **(P)**, then

$$\underline{u} \le \bar{v} \quad in \quad \overline{Q_T}.$$

Similarly, one can define $\overline{f}$ for which there is corresponding property.

Proof

Since $\underline{u}$ is subsolution of **(P)**, we have that

$$L(\underline{u}) \le \underline{f}(\underline{u}) \le f(\underline{u}),$$

i.e. $\underline{u}$ is subsolution of $(\underline{P})$, then

$$\underline{u} \le \bar{v} \quad in \quad \overline{Q_T}.$$

■

1.8.14 Lemma (on the almost monotonicity)

Let $\underline{f}$ be defined by

$$\textbf{(1.71)} \quad \underline{f_i}(u) := \inf_{\{v | u \le v \le \bar{u},\, v_i = u_i\}} f_i(v),$$

with $\bar{u}$ supersolution of **(P)**.

Then

$$\underline{f}(u) \le f(u), \forall u \le \bar{u},$$

and $\underline{f}$ is almost monotonically increasing and is uniformly Lipschitzian with the same constant as f. Analogously, it is possible to construct

$$\overline{f}(u) \ge f(u), \forall u \ge \underline{u}.$$

1.8.15 Theorem (invariant set)

Let f be locally Lipschitzian and Q_T satisfy the inner sphere property. Let a, b be such that

$$-\infty < a < b < +\infty,$$

$$f_i(v) \ge 0, \text{ for } a \le v \le b, v_i = a_i,$$

$$f_i(v) \le 0, \text{ for } a \le v \le b, v_i = b_i.$$

Then $\Sigma := \{v | a \le v \le b\}$ is invariant for **(P)**.

1.8.16 Theorem (uniqueness of the solution)

Under the assumptions of the previous theorem, we can consider **(P)** with initial conditions $\boldsymbol{u}_0 \in \Sigma$ and boundary conditions $\boldsymbol{B}\boldsymbol{u} \in \Sigma$. Then (P) has only one global solution.

Example

Consider the system for two populations competing between them

$$\begin{cases} \dfrac{\partial u_1}{\partial t} = u_1(1-u_1-\alpha u_2) + D_1 \Delta u_1 = f_1(u_1,u_2) + D_1 \Delta u_1 \\ \dfrac{\partial u_2}{\partial t} = \vartheta u_2(1-\beta u_1-u_2) + D_2 \Delta u_2 = f_2(u_1,u_2) + D_2 \Delta u_2 \end{cases}$$

such that $\alpha, \vartheta, \beta > 0$ and with

$$\begin{cases} \dfrac{\partial u_1}{\partial n} = \dfrac{\partial u_2}{\partial n} = 0, \\ u_1(\boldsymbol{x},0) = u_1^0(\boldsymbol{x}), \\ u_2(\boldsymbol{x},0) = u_2^0(\boldsymbol{x}). \end{cases}$$

If $\boldsymbol{0} \le (u_1^0, u_2^0) \le \boldsymbol{M} = (M_1, M_2)$, then the system has a global in time solution that satisfies

$$\boldsymbol{0} \le (u_1, u_2) \le \boldsymbol{b} = (b_1, b_2),$$

where $b_1 = max\{M_1, 1\}, b_2 = max\{M_2, 1\}$.

Proof

The assumptions of theorem 1.8.13 are satisfied with $\boldsymbol{a} = (0,0)$ and $\boldsymbol{b} = (b_1, b_2)$. In fact

$$f_1(0,u_2) = 0 \Rightarrow f_1(a_1,u_2) \ge 0,$$
$$f_2(u_1,0) = 0 \Rightarrow f_2(u_1,a_2) \ge 0,$$
$$f_1(b_1,u_2) = b_1(1-b_1-\alpha u_2) \le 0 \Rightarrow f_1(b_1,u_2) \le 0,$$
$$f_2(u_1,b_2) = \vartheta b_2(1-bu_1-b_2) \le 0 \Rightarrow f_2(u_1,b_2) \le 0.$$

Furthermore, we have that

$\dfrac{\partial \boldsymbol{a}}{\partial n} \le \dfrac{\partial \boldsymbol{u}}{\partial n} \le \dfrac{\partial \boldsymbol{b}}{\partial n}$, with $\dfrac{\partial \boldsymbol{a}}{\partial n}, \dfrac{\partial \boldsymbol{u}}{\partial n}, \dfrac{\partial \boldsymbol{b}}{\partial n} = 0$, and $\boldsymbol{0} \le (u_1^0, u_2^0) \le \boldsymbol{M} \le \boldsymbol{b}$.

■

1.9 Compartmental models

1.9.1 Diffusion of rabies

Travelling wave solutions also occur for systems. Consider the case of the spread of rabies [7], which is a viral disease of the central nervous system that affects warm-blooded animals, including humans. There is no cure for rabies, but there are very effective vaccines. Let us consider a fox population consisting of

- S = susceptible foxes;
- I = infected foxes.

We get the equations

$$\textbf{(1.72)} \quad \begin{cases} \dfrac{\partial S}{\partial t} = -r S I, \\ \dfrac{\partial I}{\partial t} = r S I - a I + D \dfrac{\partial^2 I}{\partial x^2}, \end{cases}$$

where

- r: infection rate;
- a: mortality rate;
- D: diffusion coefficient of infected foxes.

By introducing the quantities

$$S^* = \frac{S}{S_0}, \quad I^* = \frac{I}{S_0}, \quad x^* = \sqrt{\frac{r S_0}{D}}\, x, \quad t^* = r S_0 t, \quad m = \frac{a}{r S_0},$$

we can adimensionalize the system and obtain

$$\textbf{(1.73)} \quad \begin{cases} \dfrac{\partial S}{\partial t} = -S I, \\ \dfrac{\partial I}{\partial t} = S I - m I + \dfrac{\partial^2 I}{\partial x^2}, \end{cases}$$

where the *'s have been omitted, for simplicity.

Looking for travelling wave solutions

$$\textbf{(1.74)} \quad \begin{cases} S = S(x - ct), \\ I = I(x - ct), \end{cases}$$

we obtain

$$\textbf{(1.75)} \quad \begin{cases} c S' = S I, \\ I'' + c I' + (S - m) I = 0. \end{cases}$$

Let us assume that before the epidemic there were no infected, that is

$$S(+\infty)=1\,,\ I(+\infty)=0.$$

It is also expected that after the passage of the wave

$$S'(-\infty)=0\,,\ I(-\infty)=0.$$

Since $c\,S'=S\,I\,,$ then $I=c\,\frac{S'}{S}$ and substituting in **(1.75)₂** we get

$$I''+c\,I'+c\,S'-m\,c\,\frac{S'}{S}=0.$$

By integrating, it is obtained

$$\textbf{(1.76)}\qquad I'+cI+c\,S-m\,c\log S=cost\,.$$

Applying the condition at $+\infty$ gives $cost=c$. Applying the condition at $-\infty$ gives

$$c\,S(-\infty)-c\,m\log\left(S(-\infty)\right)=c\,,$$

from which

$$\textbf{(1.77)}\qquad \frac{S(-\infty)-1}{\log[S(-\infty)]}=m=\frac{a}{r\,S_0}\,,$$

from which we derive the fraction of survivors, as we can see in Figure 1.6.

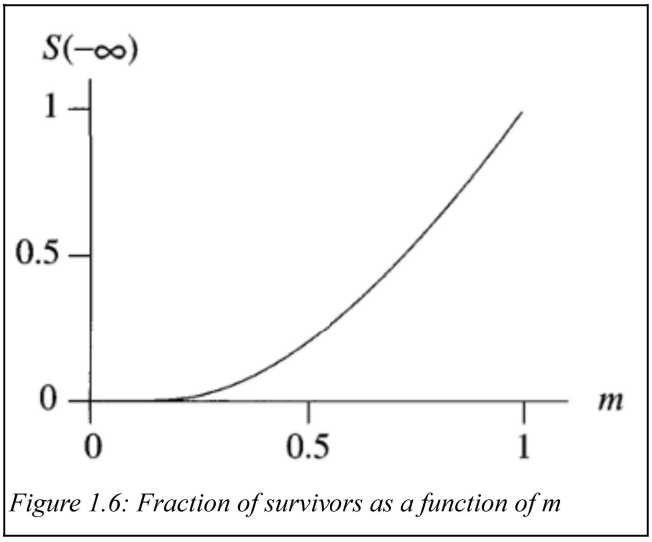

Figure 1.6: Fraction of survivors as a function of m

The parameter m is a measure of severity, the smaller it is, the smaller the number of survivors. We note that if $m>1\,,$ then $a>r\,S_0\,,$ that is, the foxes die before they can infect anyone, so the epidemic does not spread. Furthermore, the quantity $S_0=\frac{a}{r}$ also provides a minimum quantity of susceptibles for the epidemic to spread.

Let us now consider

$$\textbf{(1.78)} \qquad \begin{cases} S' = \frac{1}{c} S I, \\ I' = -c(S+I) + c\, m \log S + c, \end{cases}$$

and look for equilibrium states $S' = 0$, $I' = 0$, that is

$$\textbf{(1.79)} \qquad \begin{cases} \frac{1}{c} S I = 0, \\ -c(S+I) + c\, m \log S + c = 0, \end{cases}$$

which has solutions

$$P_1 = (S(-\infty), 0)^T, P_2 = (1,0)^T.$$

The Jacobian matrix is

$$J = \begin{pmatrix} \frac{1}{c} I & \frac{1}{c} S \\ -c + \frac{mc}{S} & -c \end{pmatrix}.$$

Evaluating in P_1 we have

$$J(P_1) = \begin{pmatrix} 0 & \frac{1}{c} S(-\infty) \\ -c + \frac{mc}{S(-\infty)} & -c \end{pmatrix},$$

whose eigenvalues are

$$\textbf{(1.80)} \qquad \lambda_{1,2} = \frac{-c \pm \sqrt{c^2 + (m - S(-\infty))}}{2}, \text{ with } m > S(-\infty),$$

therefore P_1 is a saddle. For $c < 2\sqrt{1-m}$, the point P_2 is a stable focus, so there can be no heterocline orbit for which S is positive. For $c \geq 2\sqrt{1-m}$, the point P_2 is a stable node and therefore one can have a heterocline orbit joining P_1 and P_2. Therefore

$$\textbf{(1.81)} \qquad c = 2\sqrt{1-m}$$

is the *minimum wave velocity*.

1.9.2 Model improvement

The model does not take into account growth terms in the equation for S. Therefore, the following modification should be made

$$\textbf{(1.82)} \qquad \begin{cases} \dfrac{\partial S}{\partial t} = -r\,S\,I + B\,S\left(1 - \dfrac{S}{S_0}\right), \\ \dfrac{\partial I}{\partial t} = r\,S\,I - a\,I + D\dfrac{\partial^2 I}{\partial x^2}, \end{cases}$$

where

- B: growth rate;
- S_0: carrying capacity.

We can adimensionalize the system, by posing

$$S^* = \frac{S}{S_0}, \quad I^* = \frac{r\,I}{B}, \quad x^* = \sqrt{\frac{B}{D}}\,x, \quad t^* = B\,t,$$

from which

$$\textbf{(1.83)} \qquad \begin{cases} \dfrac{\partial S}{\partial t} = -SI + S(1-S), \\ \dfrac{\partial I}{\partial t} = bI(S-m) + \dfrac{\partial^2 I}{\partial x^2}, \end{cases}$$

where $m = \dfrac{a}{rS_0}$, $b = \dfrac{rS_0}{B}$.

We now search for travelling wave solutions

$$\textbf{(1.84)} \qquad \begin{cases} S = S(z) = S(x - ct), \\ I = I(z) = I(x - ct), \end{cases}$$

We obtain the system

$$\textbf{(1.85)} \qquad \begin{cases} c\,S' = S\,I - S(1-S), \\ c\,I' = -I'' - b\,I(S-m), \end{cases}$$

and placing $V = I'$ we have

$$\textbf{(1.86)} \qquad \begin{cases} S' = \dfrac{1}{c}S(I - 1 + S), \\ I' = V, \\ V' = -c\,V - b\,I(S-m). \end{cases}$$

Obviously, it is expected that

$$(S, I, V)(+\infty) = (1,0,0)^T.$$

We note that the homogeneous system

$$\begin{cases} \frac{1}{c} S(I-1+S)=0, \\ V=0, \\ -cV-bI(S-m)=0, \end{cases}$$

has solutions

$$\begin{cases} S=1, \\ V=0, \\ I=0, \end{cases} \quad \text{and} \quad \begin{cases} I=1-m, \\ V=0, \\ S=m, \end{cases}$$

from which

$$(S,I,V)(-\infty)=(m,1-m,0)^T.$$

The Jacobian matrix is

$$J=\begin{pmatrix} -\frac{1-2S-I}{c} I & \frac{1}{c} S & 0 \\ 0 & 0 & 1 \\ -bI & -b(S-m) & -c \end{pmatrix}.$$

Evaluating in $P_1=(1,0,0)^T$ we have

$$J(P_1)=\begin{pmatrix} \frac{1}{c} & \frac{1}{c} & 0 \\ 0 & 0 & 1 \\ 0 & -b(1-m) & -c \end{pmatrix},$$

that has the following eigenvalues

$$\textbf{(1.87)} \qquad \lambda_1=\frac{1}{c}, \lambda_{2,3}=\frac{-c\pm\sqrt{c^2-4b(1-m)}}{2}.$$

One must have

$$c\geq 2\sqrt{b(1-m)}=:c.$$

Then the stable variety has dimension two, the unstable one.

Evaluating instead in $P_2=(m,1-m,0)^T$ we have

$$J(P_2)=\begin{pmatrix} \frac{m}{c} & \frac{m}{c} & 0 \\ 0 & 0 & 1 \\ -b(1-m) & 0 & -c \end{pmatrix}.$$

The eigenvalues are therefore given by the roots of the characteristic equation

$$P_b(\lambda)=\lambda^3+\left(c-\frac{m}{c}\right)\lambda^2-m\lambda+\frac{b}{c}m(1-m).$$

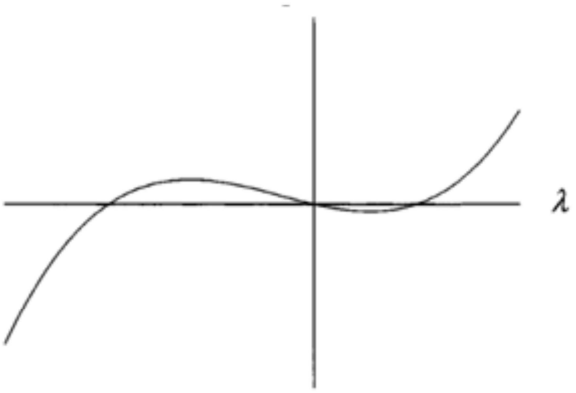

Figure 1.7:Graph of the characteristic polynomial for b = 0

In the case where $b=0$, it has eigenvalues

$$\textbf{(1.88)} \quad \lambda_1=0\,,\lambda_{2,3}=\frac{-(c-m/c)\pm\sqrt{(c-m/c)^2+4m}}{2}.$$

We want to find $b*$ such that P_{b*} has a local minimum equal to zero. Therefore, being

$$\textbf{(1.89)} \quad P_b{}'(\lambda)=3\lambda^2+2\left(c-\frac{m}{c}\right)\lambda-m\,,$$

one obtains

$$\textbf{(1.90)} \quad \lambda=\frac{m/c-c\pm\sqrt{(m/c-c)^2+3m}}{3}.$$

The minimum is in

$$\textbf{(1.91)} \quad \lambda*=\frac{m/c-c\pm\sqrt{(m/c-c)^2+3m}}{3},$$

and imposing $P_b(\lambda*)=0$, we have

$$\textbf{(1.92)} \quad b*=-\frac{c}{m(1-m)}P_0(\lambda*).$$

The two real roots $\lambda_{2,3}$ as b increases, will become complex with positive real part, then the point P_2 has an unstable manifold of dimension 2. The trajectory joining P_1 and P_2 is given by the intersection of the unstable manifold of P_2 and the stable manifold of P_1. The trajectory satisfies

$$\phi(z)\to(m\,,1-m\,,0),\quad \text{for}\quad z\to-\infty\,,$$
$$\phi(z)\to(1\,,0\,,0),\quad \text{for}\quad z\to+\infty.$$

1.9.3 Spatial distribution patterns of grey and red squirrels

In the early 20th century, North American gray squirrels were introduced to various parts of Great Britain, particularly the south-east, and they spread quickly. Concurrently, native red squirrels disappeared from these localities after only a few years of coexistence. Both red and gray squirrels can breed twice a year, but red squirrels, which are smaller, rarely have more than two or three offspring per litter, while gray squirrels often have four or five pups. In addition, gray squirrels have another competitive advantage in that they have a body weight that is almost twice as high as red squirrels. In separate environments, the two species of squirrels show quite similar social organization and feeding patterns, so it is natural that placed in the same habitat they would compete for resource exploitation and are unlikely to coexist.

Let us define

- $S_1(x,t)$: population density of grey squirrels;
- $S_2(x,t)$: population density of red squirrels.

A model describing the competition between the two species is given by

$$\textbf{(1.93)} \qquad \begin{cases} \dfrac{\partial S_1}{\partial t} = D_1 \Delta S_1 + a_1 S_1 (1 - b_1 S_1 - c_1 S_2), \\ \dfrac{\partial S_2}{\partial t} = D_2 \Delta S_2 + a_2 S_2 (1 - b_2 S_2 - c_2 S_1), \end{cases}$$

where

- a_i: growth rates;
- $\dfrac{1}{b_i}$: carrying capacity;
- c_i: coefficient of competition;
- D_i: diffusion coefficient.

As a first step, we switch to a dimensionless system, making a change of variables and defining new constants:

$$\textbf{(1.94)} \qquad \begin{aligned} &\theta_i = b_i S_i, \ \ i = 1,2, \ \ t^* = a_1 t, \ \ \boldsymbol{x}^* = \sqrt{a_1 / D}\, \boldsymbol{x}, \\ &\gamma_1 = c_1 / b_2, \ \ \gamma_2 = c_2 / b_1, \ \ k = D_2 / D_1, \ \ \alpha = a_2 / a_1, \end{aligned}$$

we obtain the system

$$\textbf{(1.95)} \quad \begin{cases} \dfrac{\partial \theta_1}{\partial t} = \Delta \theta_1 + \theta_1 (1 - \theta_1 - \gamma_1 \theta_2), \\ \dfrac{\partial \theta_2}{\partial t} = k \, \Delta \theta_2 + \alpha \theta_2 (1 - \theta_2 - \gamma_2 \theta_1), \end{cases}$$

where, as usual, we have omitted the asterisks for simplicity of notation.

In the absence of diffusion, the equilibrium states are (0,0), (1,0), (0,1). Considering diffusion, it can be shown that there is a trajectory joining (1,0) and (0,1), which corresponds to the situation in which grey squirrels drive red squirrels to extinction. Considering the one-dimensional case, we look for wave solutions of **(1.95)** of the form

$$\textbf{(1.96)} \quad \theta_i = \theta_i(z) = \theta_i(x - ct),$$

we obtain then

$$\textbf{(1.97)} \quad \begin{cases} \dfrac{\partial^2 \theta_1}{\partial z^2} + c \dfrac{\partial \theta_1}{\partial z} + \theta_1 (1 - \theta_1 - \gamma_1 \theta_2) = 0, \\ k \dfrac{\partial^2 \theta_2}{\partial z^2} + c \dfrac{\partial \theta_2}{\partial z} + \alpha \theta_2 (1 - \theta_2 - \gamma_2 \theta_1) = 0, \end{cases}$$

with conditions

$$\begin{cases} \theta_1(-\infty) = 1, \theta_2(-\infty) = 0, \\ \theta_1(+\infty) = 0, \theta_2(+\infty) = 1, \end{cases}$$

implying that the grey squirrels drive out the red ones. Analytical results for **(1.97)** can be obtained in the case $k = \alpha = 1, \gamma_1 + \gamma_2 = 2$.

Adding up the two equations and setting $\theta = \theta_1 + \theta_2$, we obtain that

$$\textbf{(1.98)} \quad \frac{\partial^2 \theta}{\partial z^2} + c \frac{\partial \theta}{\partial z} + \theta (1 - \theta) = 0,$$

which is a FKPP equation, but different boundary conditions

$$\theta(\pm\infty) = 1,$$

suggesting that

$$\theta(z) = \theta_1(z) + \theta_2(z) = 1, \forall z.$$

The first equation of **(1.97)** can then be rewritten as

$$\textbf{(1.99)} \quad \frac{\partial^2 \theta_1}{\partial z^2} + c \frac{\partial \theta_1}{\partial z} + \theta_1 (1 - \gamma_1)(1 - \theta_1) = 0.$$

It admits travelling wave solutions for θ_1 if

$$c \geq 2\sqrt{1-\gamma_1}.$$

Similarly, since $\gamma_1 = 2-\gamma_2$, we have that

$$c \geq 2\sqrt{\gamma_2 - 1}.$$

Returning to dimensional quantities we obtain

$$\textbf{(1.100)} \qquad c \geq c_{min} = 2\sqrt{a_1 D_1\left(1-\frac{c_1}{b_2}\right)}.$$

1.9.4 Spatial pattern formation

Alan Turing was the first to show that, under certain conditions, “individuals” of various nature can react and spread in such a way as to produce stable distributions that are spatially non-homogeneous (patterns). Let us consider two species with densities u, v respectively, then we have the system

$$\textbf{(1.101)} \qquad \begin{cases} \dfrac{\partial u}{\partial t} = D_u \Delta u + f(u,v), \\ \dfrac{\partial v}{\partial t} = D_v \Delta v + g(u,v), \end{cases}$$

Turing assumed that in the absence of diffusion $(D_u = D_v = 0)$, u and v apprach over time a spatially homogeneous steady state (u_0, v_0) that is asymptotically stable for the linearised system

$$\textbf{(1.102)} \qquad \begin{cases} \dfrac{\partial u}{\partial t} = f_u(u_0,v_0)(u-u_0) + f_v(u_0,v_0)(v-v_0), \\ \dfrac{\partial v}{\partial t} = g_u(u_0,v_0)(u-u_0) + g_v(u_0,v_0)(v-v_0). \end{cases}$$

Under appropriate conditions the state (u_0, v_0) becomes unstable due to diffusion and the solution evolves towards a spatially inhomogeneous stable state. Since diffusion is usually stabilising, Turing's demonstration was surprising.

Let us consider the case where

$$\textbf{(1.103)} \qquad \begin{cases} f(u,v) = k_1 - k_2 u + k_3 u^2 v, \\ g(u,v) = k_4 - k_3 u^2 v, \end{cases}$$

which is known as the Schnakenberg, with constants $k_j > 0$. This term describes a chemical reaction of the type

$$X \leftrightarrow A,$$
$$B \rightarrow Y,$$

$$2X + Y \rightarrow 3X,$$

in which the concentration of the substances A and B is kept constant. The associated mathematical model is obtained by the law of mass action.

A second system is given by the activation-inhibition mechanism introduced by Gierer-Meinhardt in 1972, in which the reaction term is

$$\textbf{(1.104)} \qquad \begin{cases} f(u,v) = k_1 - k_2 u + k_3 \dfrac{u^2}{v}, \\ g(u,v) = k_4 u^2 - k_5 v. \end{cases}$$

The model describes the evolution of the concentration, u, of a short-range autocatalytic substance, the activator, which regulates the production of the long-range antagonistic substance, the inhibitor, having concentration v.

Another system is that of Thomas, that is

$$\textbf{(1.105)} \qquad \begin{cases} f(u,v) = k_1 - k_2 u - R(u,v), \\ g(u,v) = k_3 - k_4 v - R(u,v), \end{cases}$$

where $R(u,v) = k_5 \dfrac{uv}{k_6 + k_7 u + k_8 u^2}$, which describes the reaction of the substrates oxygen, of concentration u, and uric acid, of concentration v, in the presence of the enzyme urease.

For example, we can adimensionalize the system **(1.101)** with the reaction terms **(1.103)**. Let L be a typical length and set

$$\textbf{(1.106)} \qquad \begin{aligned} & u^* = u\left(\frac{k_3}{k_2}\right)^{\frac{1}{2}}, \quad v^* = v\left(\frac{k_3}{k_2}\right)^{\frac{1}{2}}, \quad t^* = D_u \frac{t}{L^2}, \boldsymbol{x}^* = \frac{\boldsymbol{x}}{L}, \\ & d = \frac{D_v}{D_u}, \quad a = \frac{k_1}{k_2}\left(\frac{k_3}{k_2}\right)^{\frac{1}{2}}, \quad b = \frac{k_4}{k_2}\left(\frac{k_3}{k_2}\right)^{\frac{1}{2}}, \quad \gamma = \frac{L^2 k_2}{D_u}. \end{aligned}$$

Omitting the asterisks, the dimensionless equations are written as

$$\textbf{(1.107)} \qquad \begin{cases} u_t = \Delta_u + \gamma f(u,v), \\ v_t = d\,\Delta_v + \gamma g(u,v), \end{cases}$$

where $f(u,v) = a - u + u^2 v$ and $g(u,v) = b - u^2 v$.

Remark

The γ parameter is such that

1. $\gamma^{1/2} \propto L$, for $n = 1$;

2. $\gamma^{1/2} \propto L^2$, for $n=2$;
3. γ represents the relative strength of the reaction term, with respect to the diffusive one;
4. increasing γ, is equivalent to decreasing d.

1.9.5 Definition (Turing instability)

A system exhibits Turing instability if a steady state of the homogeneous system without diffusion is stable for small perturbations, but is unstable with respect to spatially non-homogeneous perturbations when diffusion is present.

1.9.6 Necessary and sufficient condition for Turing instability

Let us consider the system

$$\textbf{(1.108)} \qquad \begin{cases} \dfrac{\partial u}{\partial t} = \Delta u + \gamma f(u,v), \\ \dfrac{\partial v}{\partial t} = d\,\Delta v + \gamma g(u,v), \end{cases}$$

with conditions

$$\textbf{(1.109)} \qquad \begin{cases} u(\boldsymbol{x},0) = u_0(\boldsymbol{x}), \boldsymbol{x} \in \Omega, \\ v(\boldsymbol{x},0) = v_0(\boldsymbol{x}), \boldsymbol{x} \in \Omega, \\ \dfrac{\partial u}{\partial n} = \dfrac{\partial v}{\partial n} = 0, \boldsymbol{x} \in \partial\Omega. \end{cases}$$

In the stationary case without diffusion, we must have asymptotic stability. Let now be the Jacobian matrix

$$A = \left(\begin{matrix} f_u & f_v \\ g_u & g_v \end{matrix} \right) \Bigg|_{(u_0, v_0)}.$$

To have stability, we must impose

$$\textbf{(1.110)} \qquad \begin{cases} Tr(\boldsymbol{A}) = f_u(u_0, v_0) + g_v(u_0, v_0) < 0, \\ det(\boldsymbol{A}) = f_u(u_0, v_0)\, g_v(u_0, v_0) - f_v(u_0, v_0)\, g_u(u_0, v_0) > 0. \end{cases}$$

In fact, the eigenvalues of $\boldsymbol{A}$ are

$$\textbf{(1.111)} \qquad \lambda_{1,2} = \frac{Tr(\boldsymbol{A}) \pm \sqrt{(Tr(\boldsymbol{A}))^2 - 4\, det(\boldsymbol{A})}}{2},$$

which implies $Tr(\boldsymbol{A}) < 0$, $det(\boldsymbol{A}) > 0$. Now, let us define

$$\boldsymbol{w} = \left(\begin{matrix} u - u_0 \\ v - v_0 \end{matrix} \right),$$

and consider the complete linearized system

$$\mathbf{(1.112)} \qquad \boldsymbol{w}_t = \boldsymbol{D}\Delta \boldsymbol{w} + \gamma \boldsymbol{A}\boldsymbol{w},$$

with

$$\boldsymbol{D} = \begin{pmatrix} 1 & 0 \\ 0 & d \end{pmatrix}.$$

Let $\boldsymbol{W}(\boldsymbol{x})$ be a time-independent solution, that is such that

$$\mathbf{(1.113)} \qquad \begin{cases} \boldsymbol{D}\Delta \boldsymbol{W} + k^2 \boldsymbol{W} = 0, \\ \dfrac{\partial \boldsymbol{W}}{\partial n} = 0, \end{cases}$$

with k^2 eigenvalue of the Laplacian.

If the domain is $[0,a]\times[0,b]$, it can be seen that eigenvalues and eigenvectors are

$$\mathbf{(1.114)} \qquad k^2 = \lambda_{mn} = \pi^2 \left(\frac{n^2}{a^2} + \frac{m^2}{b^2} \right),$$

$$\mathbf{(1.115)} \qquad \boldsymbol{W}_{m,n} = \boldsymbol{C}_{m,n} \cos\left(\frac{\pi n x}{a} \right) \cos\left(\frac{\pi m x}{b} \right), \quad \boldsymbol{C} = \begin{pmatrix} C_1 \\ C_2 \end{pmatrix}.$$

We look for solutions such as

$$\mathbf{(1.116)} \qquad \boldsymbol{W} = \exp(\lambda t)\boldsymbol{W}_k(\boldsymbol{x}),$$

where $\boldsymbol{W}_k$ is the eigenfunction corresponding to the wave number k, and substituting in **(1.112)** we obtain

$$\lambda \boldsymbol{W}_k = \boldsymbol{D}\Delta \boldsymbol{W}_k + \gamma \boldsymbol{A}\boldsymbol{W}_k = -k^2 \boldsymbol{D}\boldsymbol{W}_k + \gamma \boldsymbol{A}\boldsymbol{W}_k,$$

which implies

$$\lambda \boldsymbol{W}_k + k^2 \boldsymbol{D}\boldsymbol{W}_k - \gamma \boldsymbol{A}\boldsymbol{W}_k = 0,$$

from which

$$|\lambda + k^2 \boldsymbol{D} - \gamma \boldsymbol{A}| = 0.$$

Calculating the determinant, we obtain

$$\mathbf{(1.117)} \qquad \lambda^2 + \lambda[(1+d)k^2 - \gamma(f_u + g_v)] + h(k^2) = 0,$$

where

$$\mathbf{(1.118)} \qquad h(k^2) = dk^4 - \gamma k^2 (df_u + g_v) + \gamma^2 det(\boldsymbol{A}).$$

Since **(1.112)** is linear, its solutions will be of the form

$$\mathbf{(1.119)} \qquad \boldsymbol{w}(\boldsymbol{x},t) = \sum_k c_k \exp(\lambda(k^2)t)\boldsymbol{W}_k(\boldsymbol{x}).$$

We have already imposed that the stationary state is stable in the absence of spatial effects, that is we have that $\mathrm{Re}(\lambda(k^2=0))<0$. Now, for the stationary state to be unstable for spatially non-homogeneous perturbations it must be

$$\mathrm{Re}(\lambda(k^2>0))>0, \text{ for some } k>0.$$

This can occur if and only if $h(k^2)<0$, that is, if and only if $d\,f_u+g_v>0$, but since $f_u+g_v=Tr(\boldsymbol{A})<0$, we need $d\neq 1$ and $f_u\cdot g_v<0$.

1.9.7 Theorem (necessary condition)

A necessary condition to have Turing instability is that

$$\textbf{(1.120)} \qquad D_u\neq D_v \text{ and } f_u\cdot g_v<0.$$

Remark

These conditions are necessary, but not sufficient to have $\mathrm{Re}\,\lambda>0$. To make sure that this happens, we must have that $\min_k\{h(k)\}<0$.

Let us introduce $s=k^2$, then, by considering the parabola,

$$\textbf{(1.121)} \qquad h(s)=d\,s^2-\gamma(d\,f_u+g_v)s+\gamma^2\,det(\boldsymbol{A}),$$

and calculating its derivative

$$h'(s)=2\,d\,s-\gamma(d\,f_u+g_v)=0,$$

we observe that the minimum is taken for

$$\textbf{(1.122)} \qquad s_{min}=\frac{\gamma(d\,f_u+g_v)}{2\,d},$$

and is equal to

$$\textbf{(1.123)} \qquad h(s_{min})=\gamma\left[det(\boldsymbol{A})-\frac{(d\,f_u+g_v)^2}{4\,d}\right].$$

Therefore there must be

$$\frac{(d\,f_u+g_v)^2}{4\,d}>det(\boldsymbol{A}),$$

The bifurcation (that is, the change in behaviour) occurs for

$$\textbf{(1.124)} \qquad \frac{(d\,f_u+g_v)^2}{4\,d}=det(\boldsymbol{A}),$$

which occurs for a certain critical coefficient $d_c>1$.

Correspondingly, in **(1.121)**, we derive the critical wave number

$$\textbf{(1.125)} \qquad k_c = \gamma^{\frac{1}{2}} \left[\frac{det(\boldsymbol{A})}{d_c} \right]^{\frac{1}{4}}$$

For $d > d_c$, $h(k^2)$ has two zeros:

- $\underline{k}^2 = \frac{\gamma}{2d}[(d f_u + g_v) - \sqrt{(d f_u + g_v)^2 - 4 d\, det(\boldsymbol{A})}]$;
- $\overline{k^2} = \frac{\gamma}{2d}[(d f_u + g_v) + \sqrt{(d f_u + g_v)^2 - 4 d\, det(\boldsymbol{A})}]$.

Remark

To have real solutions, one must have

$$\textbf{(1.126)} \qquad (d f_u + g_v)^2 - 4d(f_u g_v - f_v g_u) > 0.$$

1.9.8 Theorem (sufficient condition)

Conditions of the type

- $D_u \neq D_v$,
- $f_u \cdot g_v < 0$,
- $d f_u + g_v > 0$, $f_u + g_v < 0$,
- $(d f_u + g_v)^2 - 4d(f_u g_v - f_v g_u) > 0$,

are sufficient to ensure that there is an unstable range $[\underline{k}^2, \overline{k^2}]$ for the wave numbers. That is, for $k^2 \in [\underline{k}^2, \overline{k^2}]$, we have that $\mathrm{Re}(\lambda(k^2)) > 0$.

Consequences

For large t, we have that **(1.119)** becomes

$$\textbf{(1.127)} \qquad \boldsymbol{w}(\boldsymbol{x}, t) \sim \sum_{\underline{k} \le k \le \overline{k}} c_k e^{\lambda(k^2)t} \boldsymbol{W}_k(\boldsymbol{x}).$$

It is important to note that the exponential growth will be restrained by the non-linearity of the reaction terms and the solution will tend towards a non-homogeneous spatial equilibrium configuration.

1.9.9 Detailed analysis of pattern formation

Let us consider the system

$$\textbf{1.128)} \qquad \begin{cases} u_t = \gamma f(u, v) + u_{xx} = \gamma(a - u + u^2 v) + u_{xx}, \\ v_t = \gamma g(u, v) + d\, v_{xx} = \gamma(b - u^2 v) + d v_{xx}, \quad x \in (0, l_x), \end{cases}$$

in the one-dimensional case. In the absence of diffusion, the stationary state is given by

$$\textbf{(1.129)} \qquad (u_0, v_0)=\left(a+b, \frac{b}{(a+b)^2}\right),$$

and in correspondence of it, we have

$$\textbf{(1.130)} \qquad \begin{aligned} &f_u(u_0, v_0)=\frac{b-a}{a+b}, \quad f_v(u_0, v_0)=(a+b)^2>0, \\ &g_u=\frac{-2b}{a+b}<0, \quad g_v=-(a+b)^2<0. \end{aligned}$$

To have an unstable range, the following properties must hold

$$\textbf{(1.131)} \qquad \begin{cases} f_u \cdot g_v<0 \Rightarrow -b^2+a^2<0 \Rightarrow a<b, \\ f_u+g_v<0 \Rightarrow \dfrac{(b-a)}{(a+b)}-(a+b)^2<0 \Rightarrow b-a<(a+b)^3, \\ f_u g_v-f_v g_u>0 \Rightarrow (a+b)^2>0, \\ d f_u+g_v>0 \Rightarrow d(b-a)>(a+b)^3, \\ (d f_u+g_v)^2-4d(f_u g_v-f_v g_u)>0 \Rightarrow [d(b-a)+(a+b)^3]^2>4d(a+b)^4. \end{cases}$$

These inequalities define a domain in the parameter space (a, b, d), known as Turing space. Consider the eigenvalue problem

$$\textbf{(1.132)} \qquad \begin{cases} \boldsymbol{W}_{xx}+k^2 \boldsymbol{W}=0, \\ \left.\dfrac{\partial \boldsymbol{W}}{\partial x}\right|_{x=0}=\left.\dfrac{\partial \boldsymbol{W}}{\partial x}\right|_{x=l_x}=0, \end{cases}$$

whose solutions are

$$\textbf{(1.133)} \qquad \boldsymbol{W}_n(x)=\boldsymbol{C}_n \cos\left(\frac{n\pi x}{l_x}\right), n=0,1,2,\ldots,$$

where the $\boldsymbol{C}_n$ are arbitrary constant vectors. The eigenvalues are given by

$$\textbf{(1.134)} \qquad k_n=\frac{n\pi}{l_x}.$$

Once **(1.131)** are satisfied and there are wave numbers included in $[\underline{k}^2, \overline{k^2}]$, the related eigenfunctions $\boldsymbol{W}_n$ are linearly unstable. The explicit expressions of $\underline{k}$ and $\overline{k}$, in the case under consideration, are

$$\textbf{(1.135)} \qquad \begin{aligned} \underline{k}^2&=\frac{[d(b-a)-(a+b)^3]-\sqrt{[d(b-a)-(a+b)^3]^2-4d(a+b)^4}}{2d(a+b)}=:\gamma q_1(a,b,d), \\ \overline{k^2}&=\frac{[d(b-a)-(a+b)^3]+\sqrt{[d(b-a)-(a+b)^3]^2-4d(a+b)^4}}{2d(a+b)}=:\gamma q_2(a,b,d), \end{aligned}$$

such that in the interval $[\underline{k}^2, \overline{k^2}]$ we have unstability.

If γ is too small, there will be no eigenvalue in $[\underline{k}^2, \overline{k^2}]$ and thus the stationary state is stable. From **(1.127)** it follows that, as time passes, the non-homogeneous spatial solution is given by

$$\textbf{(1.136)} \qquad \boldsymbol{w}(\boldsymbol{x},t)=\sum_{n_1}^{n_2} \boldsymbol{c}_n \exp\left[\lambda\left(\frac{n^2\pi^2}{l_x^2}\right)\right]\cos\frac{n\pi x}{l_x},$$

where n_1 is the smallest value of n such that the corresponding eigenvalue is in $[\underline{k}^2, \overline{k^2}]$ and n_2 is the largest.

Let us now consider the two-dimensional case. Let $\Omega=(0,l_x)\times(0,l_y)$, we have

$$\textbf{(1.137)} \qquad \begin{cases} \left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)\boldsymbol{W}+k^2\boldsymbol{W}=0, \\ \frac{\partial \boldsymbol{W}}{\partial x}(0,y)=\frac{\partial \boldsymbol{W}}{\partial x}(l_x,y)=0, \\ \frac{\partial \boldsymbol{W}}{\partial y}(x,0)=\frac{\partial \boldsymbol{W}}{\partial y}(x,l_y)=0. \end{cases}$$

The eigenfunctions and the eigenvalues are

$$\textbf{(1.138)} \qquad \boldsymbol{W}_{nm}=\boldsymbol{C}_{nm}\cos\frac{n\pi x}{l_x}\cos\frac{m\pi y}{l_y},$$

$$\textbf{(1.139)} \qquad k_{nm}^2=\pi^2\left(\frac{n^2}{l_x^2}+\frac{m^2}{l_y^2}\right), \quad \underline{k}^2\le k_{nm}\le\overline{k^2}.$$

The emerging solution will be

$$\textbf{(1.140)} \qquad \sum_{n,m}\boldsymbol{C}_{nm}\exp\{\lambda(k_{nm}^2)t\}\cos\frac{n\pi x}{l_x}\cos\frac{m\pi y}{l_y},$$

with n, m such that $\gamma q_1(a,b,d)=\underline{k}^2\le k_{nm}^2\le\overline{k^2}=\gamma q_2(a,b,d)$.

Let us return to the one-dimensional case and assume that γ is such that in the range $[\underline{k},\overline{k}]$ only the mode with $n=1$ falls, then

$$\textbf{(1.141)} \qquad \boldsymbol{w}(\boldsymbol{x},t)\approx\boldsymbol{c}_1\exp\left\{\lambda\left(\frac{\pi^2}{l_x^2}\right)t\right\}\cos\frac{\pi x}{l_x},$$

since all other modes decay exponentially with the passage of time. Let us take $\boldsymbol{c}_1=\binom{\epsilon}{\epsilon}$, for some small $\epsilon>0$, and consider

$$\textbf{(1.142)} \qquad u(x,t)\approx u_0+\epsilon\exp\left\{\lambda\left(\frac{\pi^2}{l_x^2}\right)t\right\}\cos\frac{\pi x}{l_x}.$$

Clearly, if exponential growth continued over time we would have $|u| \to +\infty$, for $t \to +\infty$. However, it is hypothesizable that the solution eventually settles down to a spatial pattern which looks like the cosine function shown in figure 1.8. Similar considerations can be made if the unstable mode is the second one.

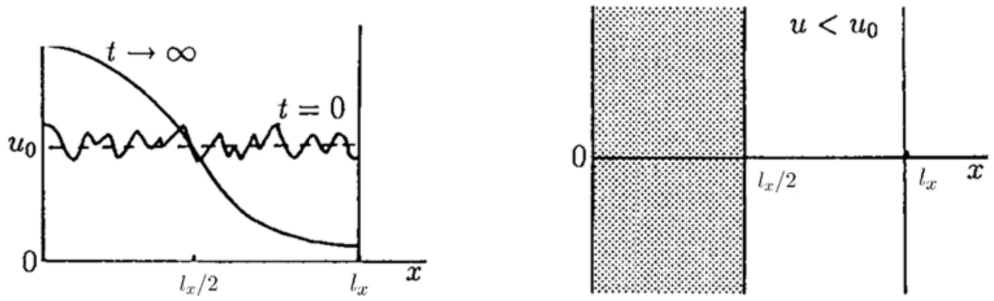

Figure 1.8:One-dimensional spatial pattern when the only unstable mode is the one with n =1

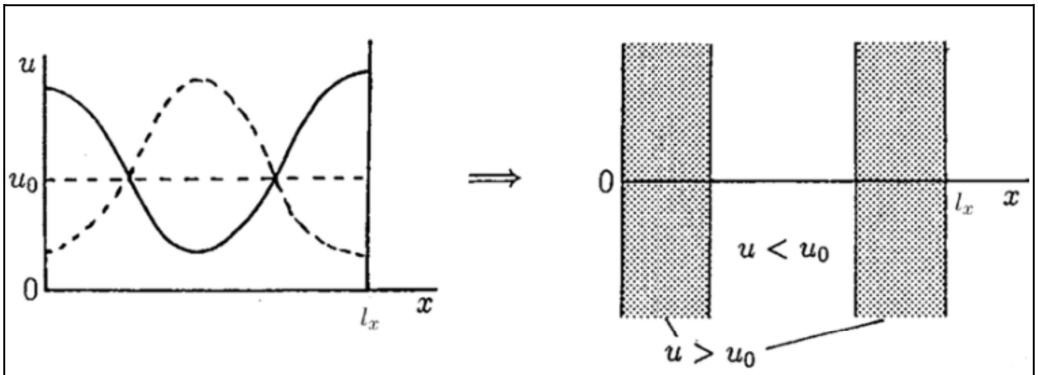

Figure 1.9: Emergent pattern when the only unstable mode is the second

The two-dimensional case, on the other hand, is much more complicated because the unstable modes depend on l_x and l_y. We have that

$$\gamma q_1(a,b,d) = \underline{k^2} \le k^2 = \pi^2 \left(\frac{n^2}{a^2} + \frac{m^2}{b^2} \right) \le \overline{k^2} = \gamma q_2(a,b,d).$$

If either dimension of the domain is too small, there is no unstable mode.

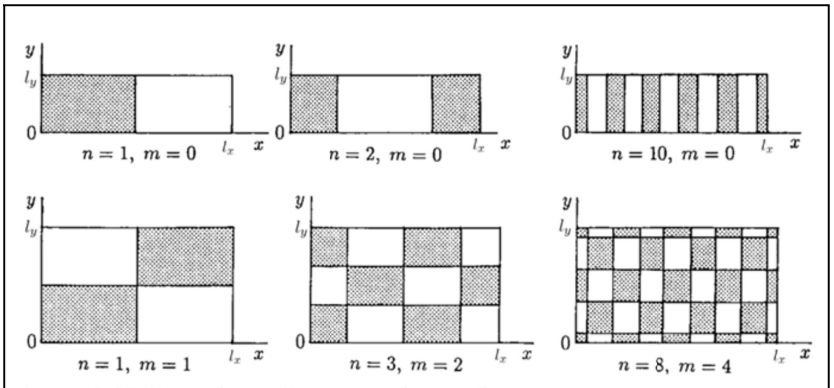

Figure 1.10:Typical two-dimensional spatial patterns

1.9.10 Pattern formation in growing domains

The evolution of patterns in growing domains can be quite complicated, but it has important biological implications. There are two cases:

1. as γ increases, the dispersion curve shifts along the wavelength axis, such that the unstable mode changes and becomes smaller ("discrete" pattern);
2. as γ increases, more modes are excited, that is the unstable band grows with gamma ("continuous" pattern)

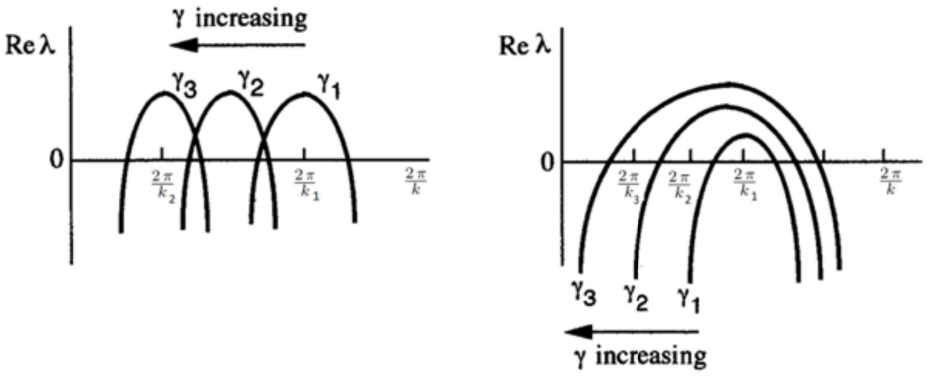

Figure 1.11:Possible behaviors of the dispersion relationship as γ increases

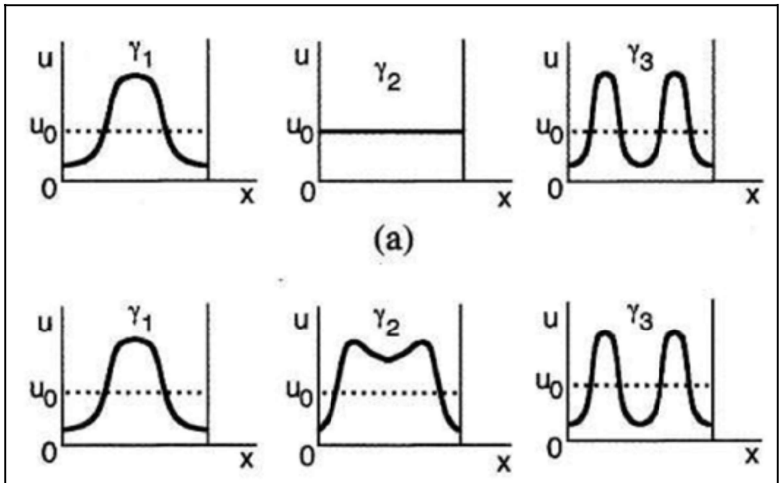

Figure 1.12: Evolution of spatial patterns for the two different types of dispersion relation behavior considered.

1.9.11 Hints of pattern formation on the coat of some mammals

Mammals exhibit a wide variety of patterns on their coats. According to J.D. Murray, a single mechanism may underlie the formation of the vast majority of commonly observed patterns: a diffusion reaction mechanism, characterized by diffusion instability. To create the colour patterns on animal skin, genetically

determined cells, melanoblasts, migrate to the surface of the embryo, where they transform into specialized cells, melanocytes, which are apt to produce melanin, a pigment responsible for skin, eye and hair colour. The basic assumption is that the movement and organization of melanoblasts reflects the spatial, prepattern distribution of as-yet unknown chemicals, the so-called morphogens. The formation of prepatterns occurs in the early stages of embryo development, while colour patterns form toward the end of embryo development. Because the mechanism of diffusion involved is not yet known, Murray in his studies used Thomas's, one both for illustrative purposes and because it describes truly observable kinetics with realistic parameters. As an example, one can consider the tail or paw of a feline. The spatial domain can then be thought of as rectangular, assuming that the skin covering the tail or paw is cut longitudinally and flattened. It is then natural to assume on the two edges at the lower and upper sides of the rectangle periodicity conditions for the concentrations of morphogens, u and v, and for their derivatives u_y and v_y, while considering homogeneous Neumann conditions on the other two sides:

$$\textbf{(1.143)} \qquad \begin{cases} u(x,0,t)=u(x,l_y,t), \\ v(x,0,t)=v(x,l_y,t), \\ u_y(x,0,t)=u_y(x,l_y,t), \\ v_y(x,0,t)=v_y(x,l_y,t), \\ u_x(0,y,t)=u_x(l_x,y,t)=0, \\ v_x(0,y,t)=v_x(l_x,y,t)=0. \end{cases}$$

Proceeding with the calculations we can obtain

$$\textbf{(1.144)} \qquad \boldsymbol{w}(x,y,t)\sim \sum_{k\le k_{nm}\le \bar{k}} \boldsymbol{C}_{nm}\exp\{\lambda_{nm}t\}\cos\left(\frac{n\pi x}{l_x}\right)\cos\left(\frac{m\pi y}{l_y}\right),$$

where the $\lambda's$ are solutions of

$$\lambda^2+\lambda[k_{nm}^2(1+d)-\gamma\, Tr(\boldsymbol{A})]+h(k_{nm}^2)=0,$$

where

$$k_{nm}^2=\pi^2\left(\frac{n^2}{l_x^2}+\frac{4m^2}{l_y^2}\right).$$

We can draw the following conclusions

- unstable modes determine the colouring of the mantle;
- instability occurs for modes with k in the instability range;

- pattern formation is only possible for $d>1$;
- if $l_y<l_x$, the formation of transverse stripes is probable, while if $l_x<l_y$, non-constant spots in the y-direction are possible.

1.9.12 Non-existence of spatial patterns in reaction-diffusion systems

We want to show how the process of pattern formation can be destroyed if diffusion is fast enough.

In the one-dimensional case we consider

$$\textbf{(1.145)} \qquad \begin{cases} u_t = f(u,v) + D_u u_{xx}, \\ v_t = g(u,v) + D_v v_{xx}, \end{cases}$$

with conditions

$$\textbf{(1.146)} \qquad \begin{cases} u_x(0,t) = u_x(1,t) = v_x(0,t) = v_x(1,t) = 0, \\ u(x,0) = u_0(x), \quad u_x(0,t) = u_x(1,t) = 0, \\ v(x,0) = v_0(x), \quad v_x(0,t) = v_x(1,t) = 0. \end{cases}$$

The integral of energy is defined as

$$\textbf{(1.147)} \qquad E(t) = \frac{1}{2}\int_0^1 (u_x^2 + v_x^2)\,dx,$$

which derived with respect to time gives us

$$\frac{dE}{dt} = \int_0^1 (u_x u_{xt} + v_x v_{xt})\,dx,$$

where

$$\begin{cases} u_{xt} = u_{tx} = f_u u_x + f_v v_x + (D_u u_{xx})_x, \\ v_{xt} = v_{tx} = g_u u_x + g_v v_x + (D_v v_{xx})_x. \end{cases}$$

Substituting in the integral, we obtain

$$\frac{dE}{dt} = \int_0^1 [u_x (D_u u_{xx})_x + u_x (f_u u_x + f_v v_x) + v_x (D_v v_{xx})_x + v_x (g_u u_x + g_v v_x)]\,dx$$

$$= [u_x D_u u_{xx} + v_x D_v v_{xx}]_0^1 - \int_0^1 (D_u u_{xx}^2 + D_v v_{xx}^2)\,dx + \int_0^1 [f_x u_x^2 + g_v v_x^2 + (f_u + g_v) v_x u_x]\,dx.$$

Let us define

$$d := min(D_u, D_v), \quad M = \max_{u,v} (f_u^2 + f_v^2 + g_u^2 + g_v^2)^{\frac{1}{2}},$$

where the maximum is taken over all values u and v that the solutions can take, which are assumed to be bounded. Let us observe that

$$(u_x - v_x)^2 \geq 0,$$

which implies

$$u_x^2 + v_x^2 \geq 2\, u_x\, v_x.$$

Then

$$\begin{aligned}\frac{dE}{dt} &\leq -d \int_0^1 (u_{xx}^2 + v_{xx}^2)\, dx + 2M \int_0^1 (u_x^2 + v_x^2)\, dx \\ &\leq -d\,\pi^2 \int_0^1 (u_x^2 + v_x^2)\, dx + 2M \int_0^1 (u_x^2 + v_x^2)\, dx = (4M - 2d\,\pi^2)\, E \ \leq 0,\end{aligned}$$

if d is sufficiently large, from which $E \to 0$ for $t \to +\infty$. Consequently

$$\textbf{(1.148)} \qquad u_x, v_x \to 0,$$

and this means that diffusion will cancel out any spatial inhomogeneity as time varies.

1.9.13 Numerical methods

Wewant to discretize the one-dimensional problem

$$\textbf{(1.149)} \qquad \begin{cases} u_t = u_{xx} + f(u), \\ u_x(0,t) = u_x(1,t) = 0, \\ u(x,0) = u^{(0)}(x). \end{cases}$$

We divide $[0,1]$ with $N+1$ equispaced nodes $x_i = i\,h$, $h = \frac{1}{N}$, $i = 0, \ldots, N$.

Approximations of the spatial derivatives can be calculated, such as:

- $u_{xx}(x_i, t) = \frac{u(x_{i+1}, t) - 2\,u(x_i, t) + u(x_{i-1}, t)}{h^2} + O(h^2)$,
- $u_x(x_0, t) = \frac{-3\,u(x_0, t) + 4\,u(x_1, t) - u(x_2, t)}{2h} + O(h^2)$,
- $u_x(x_N, t) = \frac{u(x_{N-2}, t) - 4\,u(x_N - 1, t) + 3\,u(x_N, t)}{2h} + O(h^2).$

If the $u_i(t)$ are appropriate approximations of the $u(x_i, t)$, using (4.61)-(4.62), and defining

- $\boldsymbol{U}(t) = (u_1(t), \ldots, u_{N-1}(t))^T$,
- $\boldsymbol{F}(\boldsymbol{U}(t)) = (f(u_1(t)), \ldots, f(u_{N-1}(t)))^T$,

we can discretize the problem

$$\textbf{(1.150)} \qquad \dot{\boldsymbol{U}}(t) = \boldsymbol{A}_h \boldsymbol{U}(t) + \boldsymbol{F}(\boldsymbol{U}(t)),$$

$$\textbf{(1.151)} \qquad \boldsymbol{U}(0)=(u_1(0),\dots,u_{N-1}(0))^T,$$

where $\boldsymbol{A}_h$ is an appropriate coefficient matrix. It is called the semi-discretized problem, since it is discretized only with respect to x. The main difficulty lies in choosing an appropriate scheme to discretize the time variable. In fact, the use of the simplest scheme, the Euler forward scheme, is affected by severe limitations concerning its stability and accuracy. However, an explicit third-order Runge-Kutta method can be resorted to.

Chapter 2

Neural Networks for solving reaction-diffusion equations

2.1 Machine Learning

2.1.1 ML algorithms

Sometimes it can be particularly difficult to write algorithms to solve certain types of problems, such as recognizing an image or driving a car. That's where Machine Learning (ML) algorithms [6] come in: they take advantage of a collection of examples (*training sets*), consisting of input and output, to produce a model (*target function*) for classifying new instances.

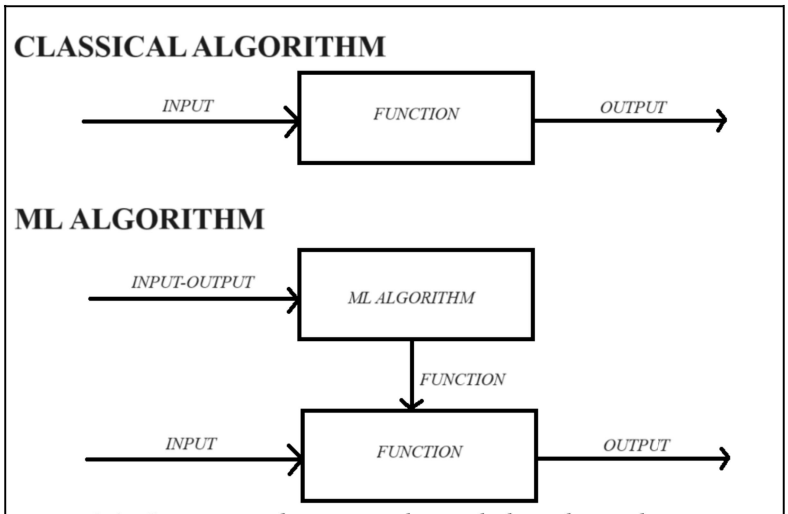

Figure 2.1: Comparison between a classical algorithm and an ML algorithm

A training set S is a set of examples (*instances*), each of which is characterized by a set of features (*attributes*) and a class label. The association between instances and the corresponding class is the target function.

2.1.2 Overfitting

One of the biggest problems of the training sets is the presence of noise, i.e., data with the wrong class label are present and that could adversely affect the creation of the model. Another problem is the lack of representative examples, i.e., the training set has too many parameters compared to the number of instances. As a result, one can fall into the phenomenon of overfitting, that is, one finds a model that works very well on the training set used, but has very poor generalization capabilities.

Example

Suppose we want to classify a set of objects into two distinct classes (represented in Figure 2.2 with a black ball and a white ball, respectively).

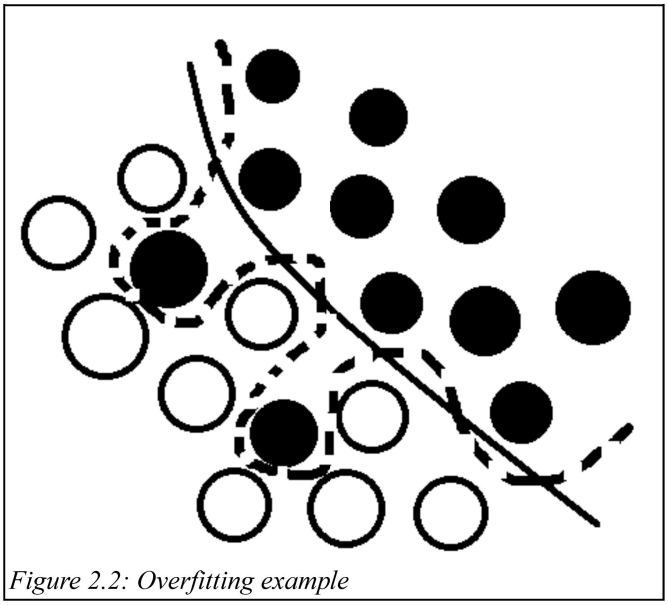

Figure 2.2: Overfitting example

It is possible to observe how the continuous line is constructed having a good accuracy on the training set, but without achieving 100% accuracy, in fact some black balls are considered as white balls. However, this allows us to have a better generalization ability since the constructed model is not too restrictive. Instead, the dashed line is constructed to present 100% accuracy on the training set (no black balls are classified as white balls, and vice versa). This results in a lower generalization ability, as the model turns out to be too restrictive and as a result some new instances may be misclassified. To prevent overfitting one should try to prefer simple models rather than complex ones.

2.2 Neural Networks

2.2.1 Definition of Neural Networks

Neural Networks are Machine Learning tools inspired by the way human brain works. They are composed of processing units called neurons (*perceptrons*) [1]. These neurons are organized into layers and connections, and the Neural Network is able to learn from the data through a training process. They have a wide range of applications, such as: Speech recognition, Autonomous driving, Image classification, etc..

2.2.2 Perceptrons

A *perceptron* is a binary classifier of the form

$$\textbf{(2.1)} \quad y=\sigma(Z)=\begin{cases} 0, & if \quad Z\leq 0, \\ 1, & if \quad Z>0, \end{cases}$$

where:

- $Z=\sum_{i=1,\dots,n} w_i x_i + w_0$;
- $x_1,\dots,x_n$ are the inputs, that is, real or boolean values;
- $w_1,\dots,w_n$ are the weights;
- w_0 is the bias (distortion);
- σ is the activation step function.

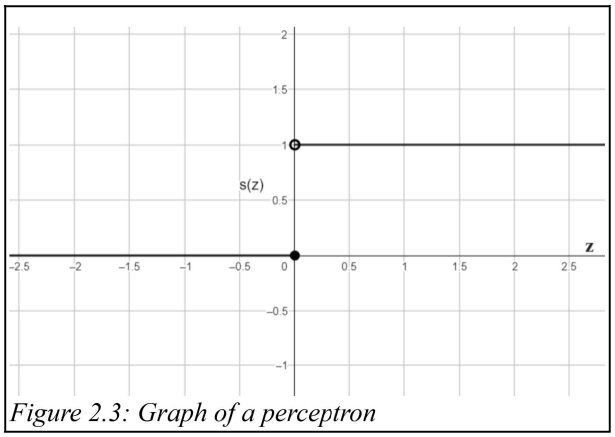

Figure 2.3: Graph of a perceptron

The "bias" is an additional parameter that represents the level of activation of a neuron when all its inputs are null. In simpler terms, bias allows the perceptron to learn even when all inputs are null. In a perceptron, each input is associated with a weight that indicates the importance of the input to the neuron's output. Bias, similarly, is an additional weight that is not associated with any specific input but still contributes to the determination of the neuron's output. Without bias, perceptron capacity would be limited to linear functions passing through the

origin. Too high or too low a bias could result in the perceptron's inability to adapt properly to the training data.

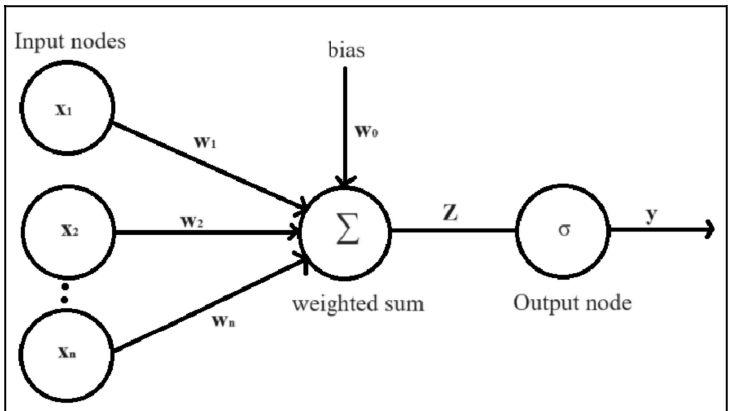

Figure 2.4: example of the structure of a Neural Network

By placing $t=-w_0$, we can observe that

$$\textbf{(2.2)} \quad y=\sigma(Z)=\begin{cases} 0, \text{ if } \sum_{i=1}^{n} w_i x_i < t, \\ 1, \text{ if } \sum_{i=1}^{n} w_i x_i > t, \end{cases}$$

that is, t represents the a*ctivation threshold*.

Example

Let's suppose to have two input nodes x_1, x_2, then

$$Z=w_0+w_1x_1+w_2x_2.$$

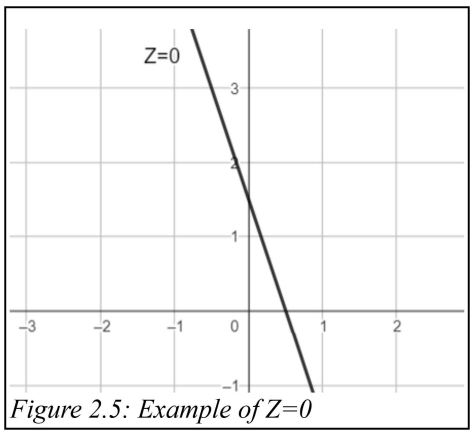

Figure 2.5: Example of Z=0

We can observe that $Z=0$ represents the *hyperplane* separating the input space into two regions:

- P, if $Z>0$;
- N, if $Z<0$.

So the perceptron will return 1 if $(x_1, x_2) \in P$, otherwise it will return 0 if $(x_1, x_2) \in N$.

Remark

Since our brain is composed of billions of neurons that are responsible for our logical thinking, it was natural to try to design a neuron that can represent Boolean functions such as "AND" and "OR." In fact, the perceptron is capable of computing any binary function that is *linearly separable*, that is, we can construct a perceptron from any Data Set that is linearly separable.

Unfortunately, not all Data Sets are linearly separable, consequently it is not possible to use a single perceptron to implement a generic function, but we must combine several perceptrons to form the so-called

2.2.3 Multilayer perceptron (MLP)

A multilayer perceptron is an artificial Neural Network composed of multiple layers of neurons, usually organized into an input layer, one or more hidden layers, and an output layer. Each layer consists of a set of artificial neurons, each of which is connected to all the neurons in the previous and next layer. In general, for each function of the type

- $f : \{0,1\}^N \to \{0,1\}$, boolean function,
- $f : \mathbb{R}^N \to \{0,1\}$, real function,

there exists an MLP that implements it. As shown in Figure 2.6, by combining multiple perceptrons, it is possible to construct an MLP that returns 1 when the input is within the closed area and 0 otherwise. The greater the number of nodes, the greater the accuracy of the approximation.

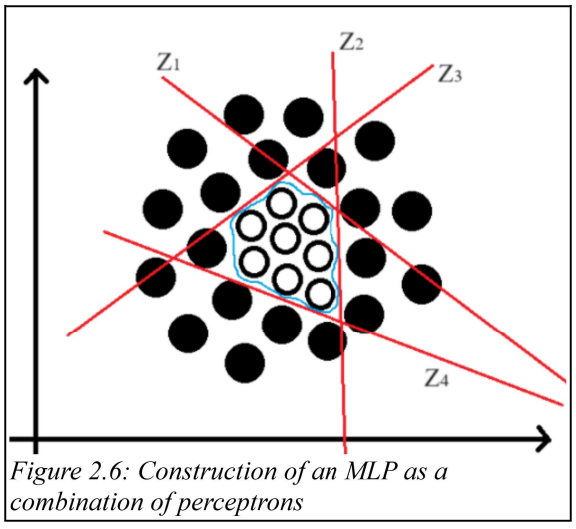

Figure 2.6: Construction of an MLP as a combination of perceptrons

Remark

Unfortunately, the step function is a discrete function and is discontinuous at the point $Z=0$, so it is not derivable. This is why it is useful to introduce new activation functions that are continuous and derivable over all $\mathbb{R}$.

2.2.4 Sigmoid function

The *Sigmoid function*, also known as the logistic function, is one of the most widely used activation functions in Neural Networks, especially in the early stages of development in the field of deep learning. It's a function that produces a sigmoid curve, that is, a curve having an "S" shape. A sigmoid function is of the form

$$\textbf{(2.3)} \quad y=\sigma(z)=\frac{1}{1+e^{-z}},$$

where

$$z=w_0+\sum_{i=1}^{n} w_i x_i.$$

A sigmoid function takes a real value and scales it between 0 and 1.

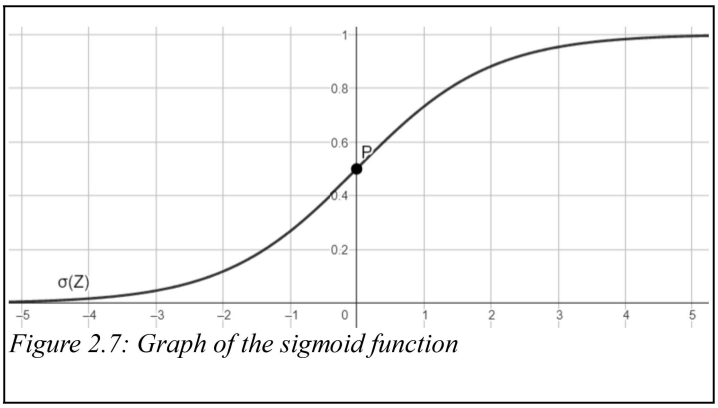

Figure 2.7: Graph of the sigmoid function

This makes the function particularly useful when you want to interpret the output as a probability. Very large negative numbers approach 0, while large positive numbers approach 1. The function returns 0.5 when the input is 0. It introduces nonlinearities into the model, allowing the network to learn complex, nonlinear representations of the data. We can observe that it is a continuous and derivable function over all $\mathbb{R}$ and its derivative is

$$\textbf{(2.4)} \quad \frac{d\sigma(z)}{dz}=\sigma(z)(1-\sigma(z)).$$

For very large or very small inputs, the derivative of the Sigmoid tends to zero, causing the vanishing gradient problem. This slows down the updating of weights in deep networks during training.

2.2.5 Hyperbolic Tangent function

The Tanh (*Hyperbolic Tangent*) activation function [5] is one of the most commonly used activation functions in Neural Networks**.** The Tanh function is defined as:

$$\textbf{(2.5)} \quad Tanh(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}.$$

Tanh maps any input value to a range between -1 and 1. This is advantageous compared to the Sigmoid function, whose output ranges between 0 and 1.

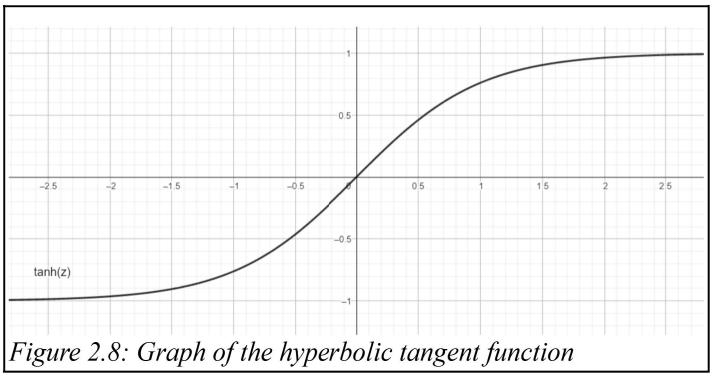

Figure 2.8: Graph of the hyperbolic tangent function

Unlike the Sigmoid, Tanh is zero-centered, meaning that negative and positive output values are balanced. This can improve convergence during training because the gradients are not all shifted in the same direction. As with the sigmoid function, the hyperbolic tangent also introduces nonlinearity into the model, allowing the network to learn complex, nonlinear representations of the data. Unfortunately, however, it can suffer from vanishing gradient problems (gradient becoming very small), slowing down the training of deep networks. Calculating the Tanh function requires the use of exponential functions, which are computationally more expensive compared to the basic operations used in other activation functions.

2.2.6 ReLU function

The ReLU (Rectified Linear Unit) activation function [5] is one of the most popular and widely used in modern Neural Networks, especially deep Neural

Networks. The ReLU function is defined as:

$$\textbf{(2.6)} \quad ReLu(z) = max(0, z).$$

It's very simple to implement and computationally efficient, since it requires only one comparison and does not involve exponential or trigonometric operations. As for the previous functions, the ReLU also introduces nonlinearity into the model, allowing the network to learn nonlinear representations of the data. The ReLU sets all negative input values to zero, which introduces sparsity into the model, that is, many nodes in the Neural Network will have value zero. Sparsity allows us to perform fewer calculations because neurons with an output of zero do not contribute further to calculations in later levels. This can make the network more efficient. Unlike activation functions such as Sigmoid and Tanh, ReLU does not suffer severely from the vanishing gradient problem, allowing gradients to remain meaningful even in deep networks. This promotes faster convergence during training. The ReLU is computationally very efficient because it requires only simple operations (compare and maximum). Because the ReLU sets negative values to zero, only a fraction of the neurons are activated at any instant, reducing the risk of overfitting and improving efficiency.

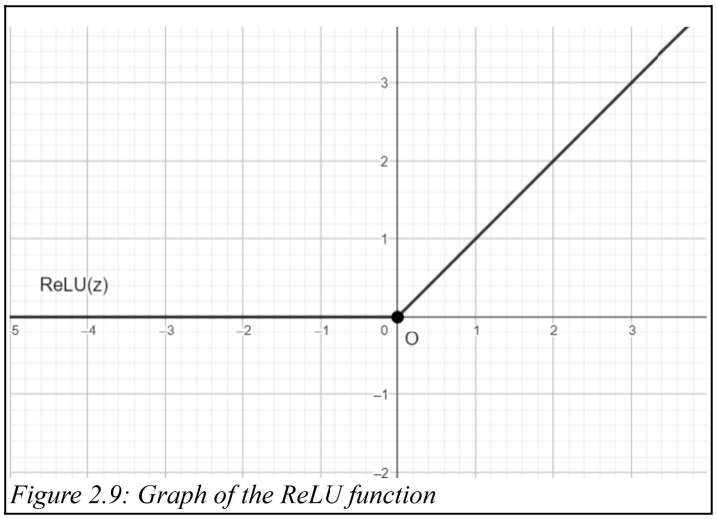

Figure 2.9: Graph of the ReLU function

A common problem with ReLU is that neurons can "die" during training. If a neuron receives negative input, its output is zero, and if this continues to happen, the neuron stops updating (its gradients become zero). This is known as the "dead neuron" problem. In some cases, very large gradients can cause training to diverge. This can be mitigated by using techniques such as gradient normalization or ReLU variants.

2.2.7 How MLP works

We have already mentioned that MLP consists of an input layer, one or more hidden layers, and an output layer. Each layer consists of a set of artificial neurons, each of which is connected to all the neurons in the previous layer and the next layer. An MLP uses a process called "feed-forward," which means that input is presented to the network and information flows forward through the hidden layers until it reaches the output layer. During this process, each node in each layer receives input from the nodes in the previous layer, applies the activation function, and sends the output to the nodes in the next layer.

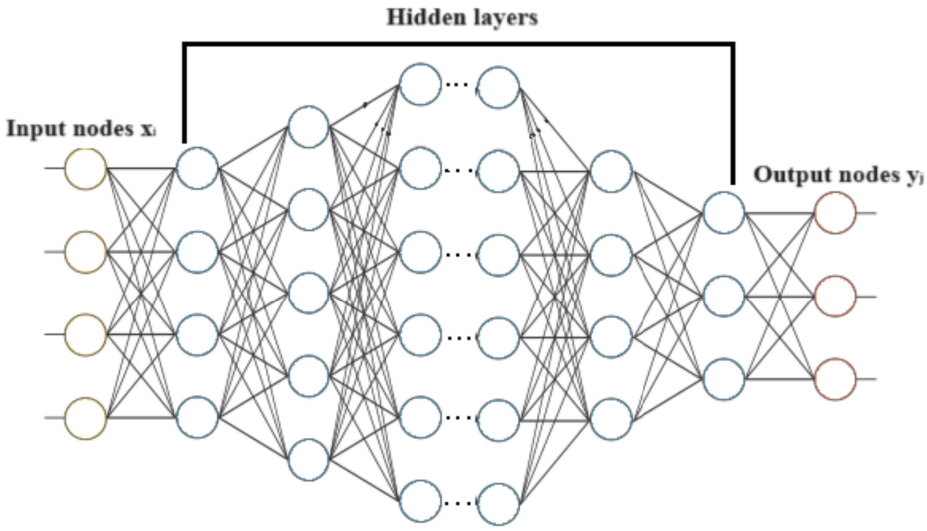

Figure 2.10: Example of MLP

Suppose we have a Neural Network defined by:

- m input nodes $x_1, \ldots, x_m$ (sometimes, we will denote by $\boldsymbol{x}^{(0)}$ or $\boldsymbol{x}$ the vector of input nodes);
- N hidden layers $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}$, consisting respectively of $d_1, \ldots, d_N$ nodes (in particular, we will denote by $x_i^{(j)}$ the i-th component of the vector $\boldsymbol{x}^{(j)}$);
- n output nodes $y_1, \ldots, y_n$ (sometimes, we will denote by $\boldsymbol{x}^{(N+1)}$ or $\boldsymbol{y}$ the vector of output nodes);
- $\boldsymbol{\omega}^{(j)}$ are the weights of the arches connecting $\boldsymbol{x}^{(j-1)}$ with $\boldsymbol{x}^{(j)}$ (we will denote by $\omega_{ki}^{(j)}$ the weight connecting $x_i^{(j-1)}$ with $x_k^{(j)}$);
- $\boldsymbol{\theta}^{(j)}$ are the biases of $\boldsymbol{x}^{(j)}$ (we will denote by $\theta_k^{(j)}$ the bias of $x_k^{(j)}$).

For the *j-th* layer we calculate

$$x_1^{(j)}=\sigma\left(\sum_{i=0}^{d_{j-1}}\omega_{1i}^{(j)}x_i^{(j-1)}\right),\ldots,\quad x_{d_j}^{(j)}=\sigma\left(\sum_{i=0}^{d_{j-1}}\omega_{d_j i}^{(j)}x_i^{(j-1)}\right),\quad j=1,..,N,$$

with $x_0^{(j)}=1,\ j=0,\ldots,N$. Instead, for the output layer, we have

$$y_1=\sigma\left(\sum_{i=0}^{d_N}\omega_{1i}^{(N+1)}x_i^{(N)}\right),\ldots,\quad y_n=\sigma\left(\sum_{i=0}^{d_N}\omega_{mi}^{(N+1)}x_i^{(N)}\right).$$

In the next subsection, we will rewrite everything in a compact form.

2.2.8 General description of a Neural Network

A Neural Network, with N hidden layers, can be seen as a map $N_\sigma:\mathbb{R}^m\to\mathbb{R}^n$, which depends on the parameters [3]

$$\boldsymbol{W}=\{(\boldsymbol{\omega}^{(1)},\boldsymbol{\theta}^{(1)}),\ldots,(\boldsymbol{\omega}^{(N+1)},\boldsymbol{\theta}^{(N+1)})\},$$

such that

$$\boldsymbol{y}=\boldsymbol{\omega}^{(N+1)}\sigma(\boldsymbol{\omega}^{(N)}\sigma(\ldots\boldsymbol{\omega}^{(2)}\sigma(\boldsymbol{\omega}^{(1)}\boldsymbol{x}+\boldsymbol{\theta}^{(1)})+\boldsymbol{\theta}^{(2)}\ldots)+\boldsymbol{\theta}^{(N)})+\boldsymbol{\theta}^{(N+1)},$$

where

- $\boldsymbol{\omega}^{(j)}\in\mathbb{R}^{d_j\times d_{j-1}},\ j=1,\ldots,N+1,$ is the matrix of weights connecting the nodes $\boldsymbol{x}^{(j-1)}$ with the nodes $\boldsymbol{x}^{(j)}$;
- $\boldsymbol{\theta}^{(j)}\in\mathbb{R}^{d_j},\ j=1,\ldots,N+1,$ is the biases vector of $\boldsymbol{x}^{(j)}$.

2.2.9 Definition (Neural Networks as composition of functions)

The Neural Network $N_\sigma:\mathbb{R}^m\to\mathbb{R}^n$ can then be seen as a composition [3] of functions

$$\textbf{(2.7)}\qquad N_\sigma=\boldsymbol{z}^{(k+1)}\circ\sigma\circ\boldsymbol{z}^{(k)}\circ\ldots\circ\sigma\circ\boldsymbol{z}^{(2)}\circ\sigma\circ\boldsymbol{z}^{(1)},$$

where the $\boldsymbol{z}^{(j)}:\mathbb{R}^{d_{j-1}}\to\mathbb{R}^{d_j}$, such that $\boldsymbol{z}^{(j)}\boldsymbol{x}=\boldsymbol{\omega}^{(j)}\boldsymbol{x}+\boldsymbol{\theta}^{(j)}$, are affine maps.

2.3 Neural Networks learning

2.3.1 Example (Error function)

Let $\boldsymbol{x}=\{x_1,\dots,x_m\}$ be the input, $\boldsymbol{W}=\{(\boldsymbol{\omega}^{(1)},\boldsymbol{\theta}^{(1)}),\dots,(\boldsymbol{\omega}^{(N+1)},\boldsymbol{\theta}^{(N+1)})\}$, be the set of weights and biases, and $\boldsymbol{y}=\{y_1,\dots,y_n\}$ be the output. We can define the *quadratic error* as

$$\varepsilon(\boldsymbol{W})=\frac{1}{2}\sum_{i=1}^{n}(y_i-t_i)^2,$$

where t_i is the target value of the output nodes y_i, $i=1,\dots,n$.

It is natural to think of a way to train the Neural Network by varying the weights and the biases in such a way that the quadratic error function is minimized.

To do this, we will use the

2.3.2 Steepest descent method (SDM)

The *steepest descent method* [10] is a widely used algorithm for finding the minimum of a scalar function $f(\boldsymbol{x})$, with $\boldsymbol{x}=(x_1,\dots,x_m)\in\mathbb{R}^m$. It is based on calculating the gradient of the objective function with respect to its arguments, and then deciding the direction in which to move to reach the minimum. You start with an initial point and move in the direction opposite to the gradient of the function at the current point. This means that you move along the direction in which the function is decreasing most rapidly. In mathematical terms, given a current point $\boldsymbol{x}^k$, it is possible to calculate the next point $\boldsymbol{x}^{k+1}$ in the following way:

$$\textbf{(2.8)}\quad \boldsymbol{x}^{k+1}=\boldsymbol{x}^k-\alpha_k\nabla f(\boldsymbol{x}^k),$$

where

- α_k is the *learning rate*, which can be fixed or variable;
- $\nabla f(\boldsymbol{x}^k)$ is the *gradient* of the function f evaluated at the point $\boldsymbol{x}^k$.

The process continues until a stop condition is reached, such as when $|\boldsymbol{x}^{k+1}-\boldsymbol{x}^k|<\epsilon$, with ϵ suitably chosen, or when the value of the function reaches a

desired threshold. This method is easy to implement and is very effective for smooth functions. However, it may converge slowly in some special cases that will not be studied. Another very important task is the choice of parameters α_k, indeed:

- if the steps are too small, convergence will be very slow;
- if the steps are too large, there is a risk of "jumping" beyond the minimum.

2.3.3 Learning scheme, the toy network

For simplicity, let us consider a perceptron consisting of 2 input nodes and a single output node.

Consider N examples of input $x_1^k, x_2^k,\ k=1,\dots,N,$ with target value

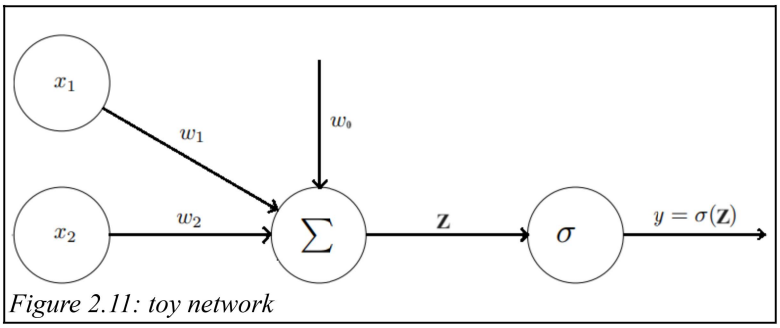

Figure 2.11: toy network

$t^k,\ k=1,\dots,N$. Since the perceptron has not yet learned, when it is applied on example k, it will produce an error $e^k=t^k-y^k,\ k=1,\dots,N$. This error will be used to improve the values of the weights. We can observe that, in k-th step, the goal is to minimize the squared error (*squared Loss Function*)

$$E^k(\omega_0,\omega_1,\omega_2)=\frac{1}{2}\left(t^k-\sigma\left(\sum_{j=0}^{2}\omega_j x_j^k\right)\right)^2.$$

The idea is to vary the weights by using the local steepest descent method in the opposite direction of the gradient of E, that is

$$\omega_j^{k+1}=\omega_j^k-\eta\frac{\partial E^k}{\partial \omega_j},\quad j=0,1,2,$$

where η, as we have said, is the *learning rate*.

Since the derivative of the sigmoid function is $\sigma'(z)=\sigma(z)(1-\sigma(z))$, we can calculate the gradient components of E^k as

$$\frac{\partial}{\partial \omega_i}E^k(\omega_0,\omega_1,\omega_2)=-e^k x_i^k\sigma'\left(\sum_{j=0}^{2}\omega_j x_j^k\right).$$

We obtain that

$$\omega_j^{k+1} = \omega_j^k + \eta \delta^k x_j^k, \quad j = 0, 1, 2,$$

where $\delta^k = e^k \sigma'(z^k)$, with $z^k = \sum_{j=0}^{2} \omega_j x_j^k$.

Due to the presence of δ^k, this procedure is also called the *delta rule* for the learning step with the sigmoid function. A gradient sweep over all N learning examples is called an epoch. Since η is usually taken small (unless a reliable estimate of its value is available), a large number of epochs may be required to approximately minimise the Loss Function for all k.

2.3.4 Learning scheme, the general case

Consider now, as it is explained in [1], that we have a MLP consisting of *m* input nodes, *n* output nodes, and a single hidden layer consisting of *N* nodes. For convenience, we consider the bias as if it were an input node with value $x_0 = 1$. We denote by:

- $\boldsymbol{\omega}^{(1)} = (\omega_{ij}^{(1)})_{i=1,\dots,N;\, j=0,1,\dots,m}$ the matrix of weights and biases of the input nodes;
- $\boldsymbol{x} = (x_0, x_1, x_2, \dots, x_m)^T$ the vector of input nodes;
- $\boldsymbol{z}^{(1)} = (z_1^{(1)}, \dots, z_N^{(1)})^T$ the vector of the weighted sums of the hidden nodes.

We observe that

$$\boldsymbol{z}^{(1)} = \boldsymbol{\omega}^{(1)} \boldsymbol{x},$$

and applying the activation function σ we get

$$\boldsymbol{x}^{(1)} = (\sigma(z_1^{(1)}), \dots, \sigma(z_N^{(1)}))^T.$$

We will also write $\boldsymbol{x}^{(1)} = \sigma(\boldsymbol{z}^{(1)}) = \sigma(\boldsymbol{\omega}^{(1)} \boldsymbol{x})$, where σ is applied componentwise.

Now we denote by:

- $\boldsymbol{\omega}^{(2)} = (\omega_{ij}^{(2)})_{i=1,\dots,n;\, j=0,1,\dots,N}$ the matrix of weights and biases of the hidden layer;
- $\boldsymbol{z}^{(2)} = (z_1^{(2)}, \dots, z_n^{(2)})^T$ the vector of the weighted sums of the output nodes.

We observe that

$$\boldsymbol{z}^{(2)} = \boldsymbol{\omega}^{(2)} \boldsymbol{x}^{(1)} = \boldsymbol{\omega}^{(2)} \sigma(\boldsymbol{\omega}^{(1)} \boldsymbol{x}),$$

and applying the activation function σ we get

$$\boldsymbol{y} = \sigma(\boldsymbol{z}^{(2)}) = \sigma(\boldsymbol{\omega}^{(2)} \boldsymbol{x}^{(1)}) = \sigma(\boldsymbol{\omega}^{(2)} \sigma(\boldsymbol{\omega}^{(1)} \boldsymbol{x})).$$

Consider K examples of input $(x_1,\dots,x_m)^k,\ k=1,\dots,K$, with target values $(t_1,\dots,t_n)^k,\ k=1,\dots,K$. Now we can define a learning process for a MLP. Obviously, using what we introduced above, we can update the $\boldsymbol{\omega}^{(2)}$ weights in the following way

$$\omega_{ij}^{(2)}=\omega_{ij}^{(2)}+\eta\,\delta_i^{(2)}\,y_j^{(1)},\ \ i=1,\dots,n,\ \ j=1,\dots,N,$$

where, for simplicity of notation, we have omitted to write the index k. Now we also want to improve the $\boldsymbol{\omega}^{(1)}$ weights, but unfortunately we cannot use the same rules as that for the output nodes, since no target value is available to estimate the error term of a hidden node. The idea is to propagate the error backward in the following way

$$e_j^{(1)}=\sum_{i=1}^{n}\omega_{ij}^{(2)}\,\delta_i^{(2)},\ \ j=1,\dots,N,$$

that is, in vector form we have

$$\boldsymbol{e}^{(1)}=\boldsymbol{\omega}^{(2)^T}\boldsymbol{\delta}^{(2)}.$$

By using this new error expression, we can define

$$\delta_i^{(1)}=e_i^{(1)}\,\sigma'\left(z_i^{(1)}\right),\ \ i=1,\dots,N.$$

Similarly to the previous step, we can formulate a delta rule for the weights $\boldsymbol{\omega}^{(1)}$, that is

$$\omega_{ij}^{(1)}=\omega_{ij}^{(1)}+\eta\,\delta_i^{(1)}\,x_j,\ \ i=1,\dots,N,\ \ j=0,1,\dots,m.$$

This idea of propagating the error and the resulting delta rule for updating the weights $\boldsymbol{\omega}^{(1)}$ is actually the result of finding the minimum of the Loss Function $\frac{\|\boldsymbol{t}-\boldsymbol{y}\|_2^2}{2}$ with respect to the elements of $\boldsymbol{\omega}^{(1)}$. Indeed, by calculation we derive

$$\frac{1}{2}\frac{\partial}{\partial\,\omega_{ij}^{(1)}}\|\boldsymbol{t}-\boldsymbol{y}\|_2^2=-\sum_{l=1}^{n}\delta_l^{(2)}\omega_{li}^{(2)}\,\sigma'\left(z_i^{(1)}\right)x_j.$$

2.3.5 Backpropagation (BP)

We can conclude summarising the BP method [1] with the SDM scheme. In this algorithm, we denote with N_{epochs} the number of learning sweeps over the training data. The matrices of weights $\boldsymbol{\omega}^{(1)}$ and $\boldsymbol{\omega}^{(2)}$ are initialised such that they have full rank. For example, for $\boldsymbol{\omega}^{(1)}$ (and similarly for $\boldsymbol{\omega}^{(2)}$) we set $\omega_{jj}^{(1)}=1,\ \ j=0,1,\dots,m$

and $w_{ij}^{(1)}=0$, if $i \neq j$, and this setting guarantees the injectivity of $\boldsymbol{\omega}^{(1)}$ and the surjectivity of its transpose.

2.3.6 Algorithm (Backpropagation and SDM method)

Input: initialise $\boldsymbol{\omega}^{(1)}$ and $\boldsymbol{\omega}^{(2)}$, choose N_{epochs}, $\eta > 0$.

Learning data: input nodes $\boldsymbol{x}^k = (x_1^k, \ldots, x_m^k)$, with target values $\boldsymbol{t}^k$, $k = 1, \ldots, K$.

for $l = 1$ to N_{epochs}

for $k = 1$ to K

$$\boldsymbol{z}^{(1)} = \boldsymbol{\omega}^{(1)} \boldsymbol{x}^k$$

$$\boldsymbol{x}^{(1)} = \sigma(\boldsymbol{z}^{(1)})$$

$$\boldsymbol{z}^{(2)} = \boldsymbol{\omega}^{(2)} \boldsymbol{x}^{(1)}$$

$$\boldsymbol{y} = \sigma(\boldsymbol{z}^{(2)})$$

$$\boldsymbol{e} = \boldsymbol{t}^k - \boldsymbol{y}$$

$$\boldsymbol{\delta}^{(2)} = \boldsymbol{e} \odot \sigma'(\boldsymbol{z}^{(2)})$$

$$\boldsymbol{e}^{(1)} = \boldsymbol{\omega}^{(2)^T} \boldsymbol{\delta}^{(2)}$$

$$\boldsymbol{\delta}^{(1)} = \boldsymbol{e}^{(1)} \odot \sigma'(\boldsymbol{z}^{(1)})$$

$$w_{ij}^{(1)} = w_{ij}^{(1)} + \eta \, \delta_i^{(1)} x_j^k, \quad i = 1, \ldots, N, \; j = 0, 1, \ldots, m$$

$$w_{ij}^{(2)} = w_{ij}^{(2)} + \eta \, \delta_i^{(2)} x_j^{(1)}, \quad i = 1, \ldots, n, \quad j = 1, \ldots, N$$

end

mix the data input

end

Remark

The symbol $\boldsymbol{a} \odot \boldsymbol{b}$ represents componentwise multiplication.

2.3.7 Definition of Loss Function

In general, a *Loss Function* $L(u)$ [3] is a functional that quantifies how much the output of a Neural Network deviates from the actual or expected value. In other words, it measures the error between the predictions made by the model and the desired outcomes. The goal of training the model is to minimize this functional so that the predictions are as accurate as possible.

Example (L2-Loss Function)

A commonly used Loss Function is the L_2-Loss Function

$$L_2(y, f(\boldsymbol{x}, \boldsymbol{\omega})) = |y - f(\boldsymbol{x}, \boldsymbol{\omega})|^2,$$

where $y \in \mathbb{R}$ is the exact value of the solution corresponding to the input $\boldsymbol{x}$, and $f(\boldsymbol{x}, \boldsymbol{\omega}) \in \mathbb{R}$ è the approximate value calculated through the Neural Network.

Very often, however, one may run into problems that require minimizing more complex operators. To do that, we will use the

2.3.8 Gradient Descent method

The *Gradient Descent method* [10] is a special version of the Steepest Descent Method in which the step is fixed, viz.

$$\mathbf{(2.9)} \quad \boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \eta \nabla f(\boldsymbol{x}^k).$$

In our case, let $\boldsymbol{\omega}, \boldsymbol{\theta}$ be the parameters on which the Neural Network $N(\boldsymbol{x}, t; \boldsymbol{\omega}, \boldsymbol{\theta})$ depends, then we will have

$$\mathbf{(2.10)} \quad \begin{aligned} \boldsymbol{\omega}_{new} &= \boldsymbol{\omega}_{old} - \eta \nabla_{\omega} N(\boldsymbol{x}, t; \boldsymbol{\omega}_{old}, \boldsymbol{\theta}), \\ \boldsymbol{\theta}_{new} &= \boldsymbol{\theta}_{old} - \eta \nabla_{\theta} N(\boldsymbol{x}, t; \boldsymbol{\omega}, \boldsymbol{\theta}_{old}), \end{aligned}$$

where η is the *learning rate*.

The *Stochastic Gradient Descent method* is a variant of the Gradient method. Unlike classical Gradient Descent, which uses the entire dataset to compute gradients, SGD uses only a single sample (or a small subset of samples, called Mini-Batch Gradient Descent) to update model parameters. This leads to several advantages and disadvantages over whole-batch based methods.

Advantages:

- parameter updates are much more frequent;
- reduces computation time per update, making the algorithm faster on very large training sets;
- converges faster (if it converges at all);
- the algorithm can exit local minimum points;
- requires less memory, since the entire training set does not have to be loaded.

Disadvantages:

- convergence is not guaranteed; the algorithm may oscillate around the minimum;
- it is often necessary to reduce the learning rate to get to convergence.

To calculate the gradient of the Loss Function, we use the

2.3.9 Automatic Differentiation

The idea behind the backpropagation is that an algorithm can be viewed as a composition of simpler operations, and then its derivative can be derived through the chain rule [3] by deriving these simpler operations.

Let f be an algorithm that can be written as a composition of simpler functions

$$\textbf{(2.11)} \quad f = f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1 .$$

Then we can define the *intermediate variables*

$$\begin{aligned} w_0 &= x, \\ w_1 &= f_1(w_0), \\ w_2 &= f_2(w_1), \\ &\vdots \\ w_n &= f_n(w_{n-1}), \end{aligned}$$

from which

$$\begin{aligned} w_0' &= \frac{\partial x}{\partial x} = 1, \\ w_1' &= \frac{\partial w_1}{\partial w_0} w_0', \\ w_2' &= \frac{\partial w_2}{\partial w_1} w_1', \\ &\vdots \\ w_n' &= \frac{\partial w_n}{\partial w_{n-1}} w_{n-1}'. \end{aligned}$$

Consequently, the derivative of f respect to x can be calculated as

$$\textbf{(2.12)} \quad \frac{\partial f}{\partial x} = \frac{\partial w_n}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial w_{n-2}} \ldots \frac{\partial w_2}{\partial w_1} \frac{\partial w_1}{\partial w_0} = \prod_{i=1}^{n} \frac{\partial w_i}{\partial w_{i-1}}.$$

2.4 The Universal Function Approximation Theorem for Neural Networks

2.4.1 Introduction

Let us consider the set

$$\textbf{(2.13)}\quad \boldsymbol{M}(\sigma)=span\{\sigma(\boldsymbol{\omega}\cdot\boldsymbol{x}-\theta)|\theta\in\mathbb{R},\ \boldsymbol{\omega}\in\mathbb{R}^n\}.$$

In [9] they asked the following question:

for which activation function σ is true that, for any $f\in C(\mathbb{R}^n)$, any compact subset K of $\mathbb{R}^n$, and any $\epsilon>0$, there exists a function $g\in\boldsymbol{M}(\sigma)$ such that

$$\textbf{(2.14)}\quad \max_{x\in K}\|N_\sigma(\boldsymbol{x})-f(\boldsymbol{x})\|<\epsilon\ ?$$

In other words, when do we have density of the linear space $\boldsymbol{M}(\sigma)$ in the space $C(\mathbb{R}^n)$, in the topology of uniform convergence on compact sets? In this case we will say that σ *has the density property*.

Density is a necessary, but not a sufficient, condition. In fact, it theoretically represents the ability to approximate well, but it does not imply that there is a valid and efficient approximation scheme. Lack of density implies that it is impossible to approximate a large class of functions. Nevertheless, it is must be understood that density does not imply that one can approximate every function well from

$$\textbf{(2.15)}\quad \boldsymbol{M}_r(\sigma)=\left\{\sum_{i=1}^{r} c_i\sigma(\boldsymbol{\omega}^{(i)}\cdot\boldsymbol{x}-\theta^{(i)})|c_i,\theta^{(i)}\in\mathbb{R},\ \boldsymbol{\omega}^{(i)}\in\mathbb{R}^n\right\},$$

for some fixed r. On the contrary, there is generally a lower bound on the degree to which one can approximate using $\boldsymbol{M}_r(\sigma)$, independent of the choice of σ. In the 1980s-90s many papers have been published which considered the density problem for general classes of activation functions:

- Gallant and White (1988) constructed a specific continuous, non-decreasing sigmoidal function (called *cosine squasher*) from which it was possible to obtain any trigonometric (Fourier) series.

- Irie and Miyake (1988) constructed an integral representation for any $f \in L^1(\mathbb{R}^n)$ by using a kernel of the form $\sigma(\boldsymbol{\omega} \cdot \boldsymbol{x} - \theta)$, where σ was an arbitrary function in $L^1(\mathbb{R})$.
- Carroll and Dickinson (1989) used a discretized inverse Radon transform to approximate L^2 functions with compact support in the L^2 norm, using any continuous sigmoidal function as an activation function.
- The main result of Cybenko (1989) is the density property, in the uniform norm on compact sets, for any continuous sigmoidal function. (Cybenko does not demand monotonicity in his definition of sigmoidality.)
- Funahashi (1989) (independently of Cybenko (1989)) proved the density property, in the uniform norm on compact sets, for any continuous monotone sigmoidal function. He noted that, for σ continuous, monotone and bounded, it follows that $\sigma(\cdot + \alpha) - \sigma(\cdot + \beta) \in L^1(\mathbb{R})$, $\forall\, \alpha, \beta \in \mathbb{R}$.
- Stinchcombe and White (1989) proved that σ has the density property for every $\sigma \in L^1(\mathbb{R})$ with $\int_{-\infty}^{+\infty} \sigma(t)\, dt \neq 0$.
- Cotter (1990) considers different types of models and non-sigmoidal activation functions, as for example $\sigma(t) = e^t$, to obtain density.
- Jones (1990) shows, using ridge functions (which we will soon define), that to answer the question of density it suffices to consider only the univariate problem. He then proves that a bounded (not necessarily monotone or continuous) sigmoidal activation function suffices.
- Hornik (1991) proved density for any continuous bounded and nonconstant activation function, and also in other norms.
- Itô (1991-1992) studied the problem of density using monotone sigmoidal functions, with only weights of norm 1. He also considers conditions under which one obtains uniform convergence on all of $\mathbb{R}^n$.
- Mhaskar and Micchelli (1992) extended the density result to what they call *k-th degree sigmoidal functions*. They prove that if σ is continuous, bounded by some polynomial of degree k on all of $\mathbb{R}$, and

$$\lim_{t \to -\infty} \frac{\sigma(t)}{t^k} = 0, \quad \lim_{t \to +\infty} \frac{\sigma(t)}{t^k} = 1,$$

then density holds if and only if σ is not a polynomial.

As we have noted, a variety of techniques were used to attack a problem which many considered important and difficult. The solution to this problem, however, turns out to be surprisingly simple. Leshno, Lin, Pinkus and Schocken (1993) proved that the necessary and sufficient condition, for any continuous activation function, to have the density property is that it not be a polynomial.

2.4.2 Theorem (Universal function approximation theorem for Neural Networks)

Let $f \in C(K, \mathbb{R}^n)$ be a continuous function from a compact subset $K \subseteq \mathbb{R}^m$ to $\mathbb{R}^n$, then there exists a Neural Network N_σ with a single hidden layer and activation function $\sigma \in C(\mathbb{R}, \mathbb{R})$ which satisfies

$$\max_{x \in K} \| N_\sigma(\boldsymbol{x}) - f(\boldsymbol{x}) \| < \epsilon,$$

for some $\epsilon > 0$, if and only if σ is not a polynomial.

The proof of this theorem is rather long and we will first need to establish several lemmas and propositions.

Remark

If σ is a polynomial then density cannot possibly hold. In fact, if σ is a polynomial of degree m, then for every choice of $\boldsymbol{\omega} \in \mathbb{R}^n$ and $\theta \in \mathbb{R}$, we have that $\sigma(\boldsymbol{\omega} \cdot \boldsymbol{x} - \theta)$ is a multivariate polynomial of total degree at most m, then

$$\boldsymbol{M}(\sigma) = span\{\sigma(\boldsymbol{\omega} \cdot \boldsymbol{x} - \theta) | \theta \in \mathbb{R},\ \boldsymbol{\omega} \in \mathbb{R}^n\}$$

becomes the space of all polynomials of total degree *m* and does not span $C(\mathbb{R}^n)$.

2.4.3 Definition of ridge function

A function $g: \mathbb{R}^n \to \mathbb{R}$ is said to be a *ridge function*, if there exists a function $f: \mathbb{R} \to \mathbb{R}$ and $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_n) \in \mathbb{R}^n$ (called *direction*), such that

$$\textbf{(2.16)} \quad g(\boldsymbol{x}) = f(\boldsymbol{\omega} \cdot \boldsymbol{x}),\ \ \forall \boldsymbol{x} \in \mathbb{R}^n.$$

In other words, they are multivariate constant functions on the parallel hyperplanes $\boldsymbol{\omega} \cdot \boldsymbol{x} = c$, $c \in \mathbb{R}$.

Notation

We will denote by

$$\textbf{(2.17)}\quad R(\Omega)=span\{g(\boldsymbol{\omega}\cdot\boldsymbol{x})|g\in C(\mathbb{R}),\boldsymbol{\omega}\in\Omega\}$$

the span of all ridge functions defined over some set $\Omega\subseteq\mathbb{R}^n$.

Ridge functions are relevant in the theory of the single hidden layer perceptron model since each factor $\sigma(\boldsymbol{\omega}\cdot\boldsymbol{x}-\theta)$ is a ridge function for every choice of σ, $\boldsymbol{\omega}$, *and* θ.

2.4.4 Theorem (Vostrecov, Kreines)

The set of ridge functions

$$R(\Omega)=span\{g(\boldsymbol{\omega}\cdot\boldsymbol{x})|g\in C(\mathbb{R}),\boldsymbol{\omega}\in\Omega\},$$

defined over some compact set $\Omega\subseteq\mathbb{R}^n$, is dense in $C(\mathbb{R}^n)$, in the topology of uniform convergence over Ω, if and only if there are no non-trivial homogeneous polynomials that vanishes on Ω.

Proof

For the proof, see [13].

Remark

Because of the homogeneity of the directions (allowing a direction $\boldsymbol{\omega}$ is equivalent to allowing all directions $\mu\boldsymbol{\omega}$, for every real μ, since we vary over all $g\in C(\mathbb{R}^n)$), it in fact suffices to consider directions normalized to lie on the unit ball

$$S^{n-1}=\{\boldsymbol{y}\mid\|\boldsymbol{y}\|_2=1\}\subset\mathbb{R}^n.$$

Then the previous theorem is equivalent to says that $R(\Omega)$, defined over some compact set $\Omega\subseteq S^{n-1}$, is dense in $C(\mathbb{R}^n)$, if no non-trivial homogeneous polynomial has zero set containing Ω. In what follows we will always assume that $\Omega\subseteq S^{n-1}$.

Notation

Let Λ be a subset of $\mathbb{R}$. We will denote with $\Lambda\times\Omega$ the subset of $\mathbb{R}^n$ given by

$$\textbf{(2.18)}\quad \Lambda\times\Omega=\{\lambda\boldsymbol{\omega}\mid\lambda\in\Lambda,\boldsymbol{\omega}\in\Omega\}.$$

2.4.5 Proposition 1 (about density)

Assume that Λ, Θ are subsets of $\mathbb{R}$ for which

$$\textbf{(2.19)}\quad \boldsymbol{N}(\sigma;\Lambda,\Theta)=span\{\sigma(\lambda t-\theta)|\lambda\in\Lambda,\ \ \theta\in\Theta\},$$

is dense in $C(\mathbb{R})$, in the topology of uniform convergence on compact sets. Let us assume, in addition, that $\Omega\subseteq S^{n-1}$ is such that $R(\Omega)$ is dense in $C(\mathbb{R}^n)$, in the topology of uniform convergence on compact sets. Then

$$\textbf{(2.20)}\quad \boldsymbol{M}(\sigma;\Lambda\times\Omega,\Theta)=span\{\sigma(\boldsymbol{\omega}\cdot\boldsymbol{x}-\theta)|\boldsymbol{\omega}\in\Lambda\times\Omega,\ \ \theta\in\Theta\}$$

is dense in $C(\mathbb{R}^n)$, in the topology of uniform convergence on compact sets.

Proof

Let $f\in C(K)$ for some compact set $K\in\mathbb{R}^n$. Since $R(\Omega)$ is dense in $C(K)$, given $\epsilon>0$, there exist $g_i\in C(\mathbb{R})$ and $\boldsymbol{\omega}^i\in\Omega$, $i=1,\dots,r$, such that

$$\left|f(\boldsymbol{x})-\sum_{i=1}^{r}g_i(\boldsymbol{\omega}^i\cdot\boldsymbol{x})\right|<\frac{\epsilon}{2},$$

for all $\boldsymbol{x}\in K$. Since K is compact, then

$$\{\boldsymbol{\omega}^i\cdot\boldsymbol{x}\ \mid\ \boldsymbol{x}\in K\}\subseteq[\alpha_i,\beta_i],$$

for some finite interval $[\alpha_i,\beta_i]$, $i=1,\dots,r$.

Now, since $\boldsymbol{N}(\sigma;\Lambda,\Theta)$ is dense in $C([\alpha_i,\beta_i])$, $i=1,\dots,r$, there exist constants

$$c_{ij}\in\mathbb{R},\ \ \lambda_{ij}\in\Lambda,\ \ \theta_{ij}\in\Theta,\ \ j=1,\dots,m_i, i=1,\dots,r,$$

for which

$$\left|g_i(t)-\sum_{j=1}^{m_i}c_{ij}\sigma(\lambda_{ij}t-\theta_{ij})\right|<\frac{\epsilon}{2r},$$

for all $t\in[\alpha_i,\beta_i]$, *and* $i=1,\dots,r$. Then

$$\left|f(\boldsymbol{x})-\sum_{i=1}^{r}\sum_{j=1}^{m_i}c_{ij}\sigma(\lambda_{ij}\boldsymbol{\omega}^i\cdot\boldsymbol{x}-\theta_{ij})\right|<\epsilon,$$

for all $\boldsymbol{x}\in K$.

■

2.4.6 Proposition 2 (about density)

Let $\sigma\in C^{\infty}(\mathbb{R})$ and assume σ is not polynomial. Then $\boldsymbol{N}(\sigma;\mathbb{R},\mathbb{R})$ is dense in $C(\mathbb{R})$.

Proof

It is known that, if $\sigma \in C^{\infty}$ on any open interval and is not polynomial, then there exists a point $-\theta_0$ in the interval for which $\sigma^{(k)}(-\theta_0) \neq 0$ for all $k = 0, 1, 2, \ldots$, see Corominas and Sunyer Balaguer (1954).

Since $\sigma \in C^{\infty}(\mathbb{R})$, and

$$\frac{[\sigma((\lambda + h)t - \theta_0) - \sigma(\lambda t - \theta_0)]}{h} \in \boldsymbol{N}(\sigma; \mathbb{R}, \mathbb{R}) \text{ for all } h \neq 0,$$

it follows that

$$\frac{d}{d\lambda}\sigma(\lambda t - \theta_0)\bigg|_{\lambda = 0} = t\,\sigma'(-\theta_0) \in \overline{\boldsymbol{N}(\sigma; \mathbb{R}, \mathbb{R})}.$$

Similarly

$$\frac{d^k}{d\lambda^k}\sigma(\lambda t - \theta_0)\bigg|_{\lambda = 0} = t^k \sigma^{(k)}(-\theta_0) \in \overline{\boldsymbol{N}(\sigma; \mathbb{R}, \mathbb{R})}, \quad \forall k.$$

Since $\sigma^{(k)}(-\theta_0) \neq 0$, $k = 0, 1, 2, \ldots$, the set $\overline{\boldsymbol{N}(\sigma; \mathbb{R}, \mathbb{R})}$ contains all monomials and thus all polynomials. By the Stone-Weierstrass Theorem this implies that $\boldsymbol{N}(\sigma; \mathbb{R}, \mathbb{R})$ is dense in $C(K)$ for every compact set $K \subset \mathbb{R}$.

■

Now, let us restate the previous Proposition in a more general form.

Corollary 1

Let Λ be any set containing a sequence of values tending to zero, and let Θ be any open interval. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be such that $\sigma \in C^{\infty}(\Theta)$, and σ is not a polynomial on Θ. Then $\boldsymbol{N}(\sigma; \Lambda, \Theta)$ is dense in $C(\mathbb{R})$.

Corollary 2

Let

$$\textbf{(2.21)} \quad \boldsymbol{N}_r(\sigma) = \left\{ \sum_{i=1}^{r} c_i \sigma(\lambda_i t - \theta_i) \,|\, c_i,\ \lambda_i,\ \theta_i \in \mathbb{R} \right\}.$$

If Θ is any open interval and $\sigma \in C^{\infty}(\Theta)$ is not a polynomial on Θ, then $\overline{\boldsymbol{N}_r(\sigma)}$ contains π_{r-1} (the linear space of algebraic polynomials of degree at most $r-1$).

2.4.7 Proposition 3 (about density)

Let $\sigma \in C(\mathbb{R})$ be not a polynomial, then $\boldsymbol{N}(\sigma; \mathbb{R}, \mathbb{R})$ is dense in $C(\mathbb{R})$.

Proof

For each $\phi \in C_0^\infty(\mathbb{R})$, let

$$\sigma_\phi(t) = (\sigma * \phi)(t) = \int_{-\infty}^{+\infty} \sigma(t-y)\phi(y)\,dy,$$

be the convolution of σ and ϕ. Since $\sigma, \phi \in C(\mathbb{R})$ and ϕ has compact support, the above integral converges for all t, and can be seen that

$$\sigma_\phi \in \overline{N(\sigma; \{1\}, \mathbb{R})} \cap C^\infty(\mathbb{R}).$$

Since

$$\sigma_\phi(\lambda t - \theta) = \int_{-\infty}^{+\infty} \sigma(\lambda t - \theta - y)\phi(y)\,dy,$$

for each $\lambda \in \mathbb{R}$, it also follows that

$$\overline{N(\sigma_\phi; \mathbb{R}, \mathbb{R})} \subset \overline{N(\sigma; \mathbb{R}, \mathbb{R})}.$$

Since $\sigma_\phi \in C^\infty(\mathbb{R})$, we have, from the proof of **proposition 2.4.6**, that

$$t^k \sigma_\phi^{(k)}(-\theta) \in \overline{N(\sigma_\phi; \mathbb{R}, \mathbb{R})}, \quad \forall \theta \in \mathbb{R}, \text{ and } \forall k.$$

Now, suppose by contradiction that $N(\sigma; \mathbb{R}, \mathbb{R})$ is not dense in $C(\mathbb{R})$, then there exists k such that $t^k \notin \overline{N(\sigma; \mathbb{R}, \mathbb{R})}$. Thus $t^k \notin \overline{N(\sigma_\phi; \mathbb{R}, \mathbb{R})}$ for each $\phi \in C_0^\infty(\mathbb{R})$. This implies that $\sigma_\phi^{(k)}(-\theta) = 0$, $\forall \theta \in \mathbb{R}$, and $\forall \phi \in C_0^\infty(\mathbb{R})$. Then σ_ϕ is a polynomial of degree at most $k-1$ for each $\phi \in C_0^\infty(\mathbb{R})$. It is well known that there exist a sequence $\{\phi_n\}_n \subset C_0^\infty(\mathbb{R})$ for which σ_{ϕ_n} converges to σ uniformly on any compact set in $\mathbb{R}$ (we can take, for example, the *mollifiers*, see Adams (1975) p.29). Polynomials of a fixed degree form a closed finite-dimensional linear subspace. Since σ_ϕ is a polynomial of degree at most $k-1$ for every ϕ_n, it follows that σ is a polynomial of degree at most $k-1$. This contradicts our assumption.

■

Now, we are ready to prove the universal function approximation theorem for Neural Networks.

Proof (Universal function approximation theorem for Neural Network)

By using the **proposition 2.4.7** and noting that if f is a polynomial, then the density property cannot hold, essentially proves the theorem. Setting the degree of

f to any fixed positive integer implies that the linear span of the set of all Neural Networks with one hidden layer cannot span all of $\mathbb{R}^n$.

■

2.4.8 Application

Let us consider the continuous function

$$f(x)=\cos(x),$$

defined over the interval $[-\pi,\pi]$. Our goal is to build a neural network that can approximate this function sufficiently well. For the moment, we consider this case only as an example to show the ability of Neural Networks. In the next paragraphs, more clarification will be given on how the learning process works. Let us consider the L_2-Loss Function

$$L_2(u(x;\boldsymbol{\omega},\boldsymbol{\theta}),f(x))=\int_{-\pi}^{\pi}(u-f(x))^2\,dx,$$

where $u(x;\boldsymbol{\omega},\boldsymbol{\theta})$ represents the Neural Network. Let us set $\{x_k\}_{k=1}^{N}$ uniformly sampled nodes on the interval $[-\pi,\pi]$, then the L_2 Loss Function can be approximated as

$$L_2(u(x;\boldsymbol{\omega},\boldsymbol{\theta}),f(x))\approx\frac{1}{N}\sum_{k=1}^{N}\left(u(x_k)-f(x_k)\right)^2.$$

For the approximation of the function under study, we constructed a Neural Network consisting of an input node, two hidden layers each consisting of two nodes, and an output node. To train the Neural Network we set 11 uniformly sampled training nodes in the interval $[-\pi,\pi]$, and using the Backpropagation algorithm we obtained the following error

$$L_2(u(x;\boldsymbol{\omega},\boldsymbol{\theta}),f(x))=9.5638\times10^{-4}.$$

Figure 2.12 shows the comparison between the graph of the function and that derived from the Neural Network.

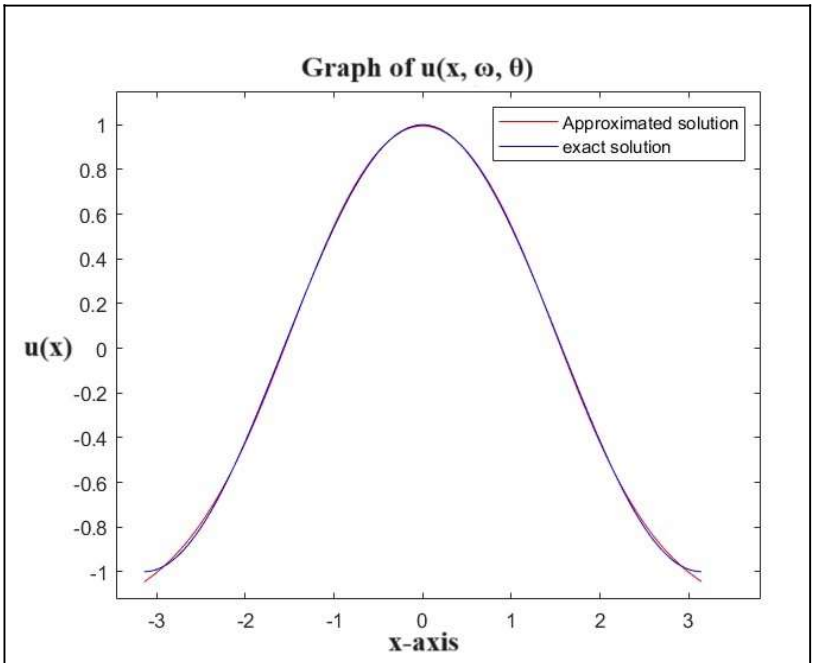

Figure 2.12: comparison between the graph of the exact solution and the approximated one.

For more details regarding Matlab codes, see Appendix A.

2.5 Neural Network for Solving Differential Equations

Our goal now is to construct and train a Neural Network so as to approximate the solution of a PDE.

2.5.1 A first approach

To fix ideas about how this process works, let us first consider the following one dimensional Cauchy's problem for a first-order ODE [1]:

$$\textbf{(2.22)} \quad \begin{cases} y'(x) = f(x, y(x)), \\ y(a) = y_a. \end{cases}$$

We can define a *trial function* of the type

$$\textbf{(2.23)} \quad \bar{y}(x) = y_a + (x-a)N(x; \boldsymbol{\omega}, \boldsymbol{\theta}),$$

where $N(x; \boldsymbol{\omega}, \boldsymbol{\theta})$, represents the Neural Network we want to build, while $\boldsymbol{\omega}$ and $\boldsymbol{\theta}$ are the sets of weights and biases, respectively, which compose the Neural

Network. We observe that the trial function was created in such a way as to automatically satisfy the initial condition.

Suppose we construct a Neural Network with one input node, K hidden nodes and one output node. We denote by

- $\boldsymbol{\omega}^{(1)}$ the weights of the first layer;
- $\boldsymbol{\theta}$ the vector of biases;
- $\boldsymbol{\omega}^{(2)}$ the weights of the second level.

The output is linear, that is, no activation function is applied at the end. Obviously by construction we have that

$$N(x;\boldsymbol{\omega},\boldsymbol{\theta})=\sum_{j=1}^{K}\omega_j^{(2)}\sigma(\omega_j^{(1)}x+\theta_j),$$

from which, deriving with respect to x, we have

$$N'(x;\boldsymbol{\omega},\boldsymbol{\theta})=\sum_{j=1}^{K}\omega_j^{(2)}\omega_j^{(1)}\sigma'(\omega_j^{(1)}x+\theta_j).$$

By using these two expressions, it is possible to derive the derivative of the trial function, i.e.

$$\begin{aligned}\overline{y}'(x)&=N(x;\boldsymbol{\omega},\boldsymbol{\theta})+(x-a)N'(x;\boldsymbol{\omega},\boldsymbol{\theta})\\&=\sum_{j=1}^{K}\omega_j^{(2)}\sigma(\omega_j^{(1)}x+\theta_j)+(x-a)\sum_{j=1}^{K}\omega_j^{(2)}\omega_j^{(1)}\sigma'(\omega_j^{(1)}x+\theta_j).\end{aligned}$$

How it works:

Suppose we have a training set $\boldsymbol{x}=(x_1,\dots,x_M)$ composed of M points in the interval $[a,b]$. At each step, suppose we give as input the point x_j. Obviously the NN has not learned yet, so it will generate an error $e_j=\overline{y}'(x_j)-f(x_j,\overline{y}(x_j))$. We will use this error to improve the weights and biases via the Backpropagation argorithm, with the goal of minimizing the error. We denote by $\overline{\boldsymbol{x}}=(\overline{x_1},\dots,\overline{x_N})$ the set of test points in the interval $[a,b]$. We can now define our Loss Function as

$$\textbf{(2.24)}\quad L^2(\overline{\boldsymbol{x}};\boldsymbol{\omega},\boldsymbol{\theta})=\frac{1}{2}\sum_{i=1}^{N}[\overline{y}'(\overline{x_i})-f(\overline{x_i},\overline{y}(\overline{x_i}))]^2.$$

If now the total error is less than a fixed ϵ, the algorithm stops, otherwise the procedure iterates until the solution is "good enough." For more information on the results see [1], section 14.3.

Remark

Let us observe that minimizing the Loss Function is equivalent to minimizing, for each point x_j in the training set, the quantity

$$(\overline{y}'(x_j) - f(x_j, \overline{y}(x_j)))^2.$$

Doing this, from a computational point of view, brings us a number of advantages, as we shall see later.

2.5.2 Neural Networks for PDEs

Let $\boldsymbol{x} \in U \subset \mathbb{R}^n$, F be a given function and $u : U \to \mathbb{R}$ the unknown of the second order differential equation [14]

$$\textbf{(2.25)} \quad F(\boldsymbol{x}, u(\boldsymbol{x}), \nabla u(\boldsymbol{x}), \Delta u(\boldsymbol{x})) = 0,$$

equipped with boundary conditions.

Discretization

First we need to discretize the domain of the function and its boundary by a grid of points $\hat{U} = \{\boldsymbol{x}_i\}_i$ and $\partial \hat{U} = \{\overline{\boldsymbol{x}}_j\}_j$, respectively. Accordingly, the problem can be rewritten as a system of equations

$$\textbf{(2.26)} \quad F(\boldsymbol{x}_i, u(\boldsymbol{x}_i), \nabla u(\boldsymbol{x}_i), \Delta u(\boldsymbol{x}_i)) = 0, \quad \forall \boldsymbol{x}_i \in \hat{U},$$

with associated boundary conditions. We denote by $\overline{u}(\boldsymbol{x}; \boldsymbol{\omega}, \boldsymbol{\theta})$ the test solution, where $\boldsymbol{\omega}$ and $\boldsymbol{\theta}$ are the sets of weights and biases, respectively, then the problem is to minimize the error

$$L^2(\hat{U}, \boldsymbol{\omega}, \boldsymbol{\theta}) = \sum_{x_i \in \hat{U}} F(\boldsymbol{x}_i, \overline{u}(\boldsymbol{x}_i; \boldsymbol{\omega}, \boldsymbol{\theta}), \nabla \overline{u}(\boldsymbol{x}_i; \boldsymbol{\omega}, \boldsymbol{\theta}), \Delta \overline{u}(\boldsymbol{x}_i; \boldsymbol{\omega}, \boldsymbol{\theta}))^2,$$

subject to the constraints imposed by the boundary conditions.

How to construct the test solution

To construct the trial function $\overline{u}(\boldsymbol{x}; \boldsymbol{\omega}, \boldsymbol{\theta})$, we must keep in mind the boundary conditions to which the solution is subject, consequently we obtain

$$\textbf{(2.27)} \quad \overline{u}(\boldsymbol{x}; \boldsymbol{\omega}, \boldsymbol{\theta}) = A(\boldsymbol{x}) + f(\boldsymbol{x}, N(\boldsymbol{x}; \boldsymbol{\omega}, \boldsymbol{\theta})),$$

where $A(\boldsymbol{x})$ is independent of $\boldsymbol{\omega}$ and $\boldsymbol{\theta}$, and satisfies the boundary conditions, $f(\boldsymbol{x}, N(\boldsymbol{x}; \boldsymbol{\omega}, \boldsymbol{\theta}))$ is constructed to satisfy the PDE, not contributing to the boundary condition that is already satisfied, while $N(\boldsymbol{x}; \boldsymbol{\omega}, \boldsymbol{\theta})$ represents the

output of the Neural Network whose parameters must be adjusted so as to minimize the error.

Neural Networks training

Suppose we have a training set $\boldsymbol{x}=(\boldsymbol{x}_1,\dots,\boldsymbol{x}_M)$ composed of M points in the domain U. At each step, suppose we give as input the point $\boldsymbol{x}_j$. Obviously the NN has not yet learned, so it will generate an error

$$e_j=F(\boldsymbol{x}_j,u(\boldsymbol{x}_j),\nabla u(\boldsymbol{x}_j),\Delta u(\boldsymbol{x}_j)),$$

that will be used to improve the weights and biases via the Back Propagation algorithm, with the goal of minimizing the error $L^2(\hat{U},\boldsymbol{\omega},\boldsymbol{\theta})$ on the grid nodes. If the total error is less than a fixed ε, the algorithm stops, otherwise we iterate the process until the solution is "good enough".

2.5.3 A different approach

Let $\boldsymbol{x}\in\Omega\subset\mathbb{R}^n$ and $u:U\to\mathbb{R}$ be the solution of the problem

$$\textbf{(2.28)}\quad \begin{cases} F(\boldsymbol{x},u)=0, \ in\ \Omega, \\ B(\boldsymbol{x},u)=0, \ on\ \partial\Omega, \end{cases}$$

where, for simplicity of expression, we have not specified the dependence on the derivatives. Following what was done in [12], our goal is to find the solution $\hat{u}(\boldsymbol{x})$ of the problem **(2.28)**, minimizing a given Loss Function $L(u)$. For this process to be valid and useful, the following properties must be satisfied:

Property 1: The function that minimizes $L(u)$ must coincide with the solution of the PDE, i.e.

$$\underset{u\in C^k}{arg\ min}\, L(u)=\hat{u},$$

where k is the order of the equation under consideration.

Although Neural Networks are capable of approximating any function, this is not true in the case of finite Neural Networks. In particular, we cannot assume that the solution of any PDE can be expressed by a finite Neural Network. Consequently, the true minimum of L may not be reached, and so we must impose another requirement on L.

Property 2: For each $\epsilon > 0$, there exists $\delta_\epsilon > 0$, such that

$$L(u) - L(\hat{u}) < \delta_\epsilon \quad \Rightarrow \quad \|u - \hat{u}\| < \epsilon .$$

Property 3: $L(u)$ has a unique global minimum.

Property 4: For any $\epsilon > 0$, there exists $\delta_\epsilon > 0$, such that

$$\|u - \hat{u}\| > \delta_\epsilon \quad \Rightarrow \quad L(u) - L(\hat{u}) > \epsilon .$$

We can now define the loss operators as

$$L_F^p(\boldsymbol{x}, u) = \|F(\boldsymbol{x}, u)\|_p^p = \int_\Omega |F(\boldsymbol{x}, u)|^p \, d\boldsymbol{x},$$
$$L_B^p(\boldsymbol{x}, u) = \|B(\boldsymbol{x}, u)\|_p^p = \int_{\partial\Omega} |B(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}.$$

Our goal is to minimize both norms through a single Loss Function. The idea is to use the function

$$\textbf{(2.29)} \quad L(\boldsymbol{x}, u; p) = \lambda L_F(\boldsymbol{x}, u; p) + (1 - \lambda) L_B(\boldsymbol{x}, u; p),$$

where the parameter $\lambda \in (0, 1)$ is called the *loss weight,* and means that the two operators can have different weights. A possible choice for this parameter is

$$\textbf{(2.30)} \quad \lambda = \frac{\|\partial\Omega\|}{\|\partial\Omega\| + \|\Omega\|}.$$

This parameter can be optimized through some methodologies, see [11].

It is possible to prove that the Loss Function **(2.29)** satisfies properties 1-4, that is, the following are valid:

Theorem 1

The Loss Function **(2.29)** reaches its minimum when it is evaluated in the solution of the PDE.

Theorem 2

For linear PDEs, the Loss Function **(2.29)**, with $p = 2$, has a unique global minimum point.

Theorem 3

Let's consider the problem

$$\begin{cases} F(\boldsymbol{x},u)=f(\boldsymbol{x}), \ in\ \Omega, \\ B(\boldsymbol{x},u)=b(\boldsymbol{x}), \ on\ \partial\Omega, \end{cases}$$

and let $\hat{u}$ be a solution. Then, for each $\epsilon>0$, there exists $\delta_\epsilon>0$, such that

$$L(u)-L(\hat{u})<\delta_\epsilon \quad \Rightarrow \quad \|u-\hat{u}\|<\epsilon.$$

Theorem 4

Let's consider the problem

$$\begin{cases} F(\boldsymbol{x},u)=f(\boldsymbol{x}), \ in\ \Omega, \\ B(\boldsymbol{x},u)=b(\boldsymbol{x}), \ on\ \partial\Omega, \end{cases}$$

and let $\hat{u}$ be a solution. Then, for each $\epsilon>0$, there exists $\delta_\epsilon>0$, such that

$$\|u-\hat{u}\|>\delta_\epsilon \quad \Rightarrow \quad L(u)>\epsilon.$$

2.5.4 Mixed boundary conditions

Sometimes, we are in the case where the boundary $\partial\Omega$ can be separated into several parts $\partial\Omega_1,\dots,\partial\Omega_n$, each of which is characterized by different boundary conditions, i.e.

$$\textbf{(2.31)} \quad B(\boldsymbol{x},u)=\begin{cases} B_1(\boldsymbol{x},u), & \boldsymbol{x}\in\partial\Omega_1, \\ \dots & \\ B_n(\boldsymbol{x},u), & \boldsymbol{x}\in\partial\Omega_n. \end{cases}$$

In this case, the loss operator L_B^p is defined as

$$L_B^p(\boldsymbol{x},u)=\sum_i \gamma_i \int_{\partial\Omega_i} |B_i(\boldsymbol{x},u)|^p d\,\boldsymbol{x},$$

where, as before, it was taken into account that different functionals may have different weights.

This theoretical framework must now be translated into practical algorithms that train Neural Networks to approximate the solution to our problem.

2.5.5 Weight space

First we need to move from the space of functions to a space of weights, as it was done in [12]. To do this we introduce the parameters $\boldsymbol{\omega},\boldsymbol{\theta}$ which represent the weights and biases of the Neural Network, respectively. Through the Neural Network we define the parametrized function $u(\boldsymbol{x};\boldsymbol{\omega},\boldsymbol{\theta})$. Using this expression, the Loss Function becomes

(2.32) $L^p(u(\boldsymbol{x};\boldsymbol{\omega},\boldsymbol{\theta}))=\lambda\int_{\Omega}|F(u(\boldsymbol{x};\boldsymbol{\omega},\boldsymbol{\theta}))|^p d\boldsymbol{x}+(1-\lambda)\int_{\partial\Omega}|B(u(\boldsymbol{x};\boldsymbol{\omega},\boldsymbol{\theta}))|^p d\boldsymbol{x}.$

Unfortunately, through this process, it is not assured that the properties examined above still hold, and this problem has not been solved yet. Unfortunately, there is still not much theoretical knowledge regarding the convergence of Neural Networks, and consequently we will not deal with it. Nevertheless, from an empirical point of view, this transition seems to be valid, especially for large Neural Networks. The hope is that, with the achievement of new theoretical results, it will become possible to show that, for $N\rightarrow+\infty$, where N is the number of degree of freedom of the network, the enunciated properties are still valid.

2.5.6 Monte Carlo method

The Monte Carlo method is used to approximate integrals by summations of functional evaluations, i.e.

(2.33) $\int_{\Omega} f(\boldsymbol{x})d\boldsymbol{x}\approx\|\Omega\|\frac{1}{n}\sum_{i=1}^{n} f(\boldsymbol{x}_i),$

where the $\boldsymbol{x}_i$ are uniformly sampled points on Ω.

Using the Monte Carlo method, the Loss Function is approximated as

(2.34) $L^p(u)\approx\lambda\|\Omega\|\frac{1}{n_F}\sum_{i=1}^{n_F}|F(u_i^F)|^p+(1-\lambda)\|\partial\Omega\|\frac{1}{n_B}\sum_{i=1}^{n_B}|B(u_i^B)|^p,$

where, to simplify the notation, we placed

- n_F is the number of uniformly sampled points on Ω;
- n_B is the number of uniformly sampled points on $\partial\Omega$;
- $u=u(\boldsymbol{x};\boldsymbol{\omega},\boldsymbol{\theta})$;
- $u_i^F=u(\boldsymbol{x}_i^F;\boldsymbol{\omega},\boldsymbol{\theta})$;
- $u_i^B=u(\boldsymbol{x}_i^B;\boldsymbol{\omega},\boldsymbol{\theta})$.

Example

Let Ω be a domain of $\mathbb{R}^2$, let us consider the Laplace's equation

(2.34) $\begin{cases}\Delta u(\boldsymbol{x})=0, & in\ \ \Omega,\\ u(\boldsymbol{x})=G(\boldsymbol{x}), & on\ \ \Omega.\end{cases}$

To lead us back to a form of type **(2.28)**, we rewrite **(2.34)** as

$$\textbf{(2.35)}\quad \begin{cases} \Delta u(\boldsymbol{x})=0, \ \ in \ \ \Omega, \\ u(\boldsymbol{x})-G(\boldsymbol{x})=0, \ \ on \ \ \partial\Omega. \end{cases}$$

The Loss Function will be

$$L^p(\boldsymbol{x},u)=\lambda\int_{\Omega}|\Delta u(\boldsymbol{x})|^p d\,\boldsymbol{x}+(1-\lambda)\int_{\partial\Omega}|u(\boldsymbol{x})-G(\boldsymbol{x})|^p d\,\boldsymbol{x}.$$

Let $\Omega=[0,A]\times[0,B]$ be a square, then we can set $\{x_i\}_{i=0}^{n}$ uniformly sampled points on the interval $[0,A]$ and we can set $\{y_j\}_{j=0}^{m}$ uniformly sampled points on the interval $[0,B]$. We thus constructed a grid of points on Ω of the form $\{(x_i,y_j)\}_{i=0,\dots,n;\,j=0,\dots,m}$. Once we move to the space of weights, and using the Monte Carlo method, we derive

$$L^p(u(\boldsymbol{x};\boldsymbol{\omega},\boldsymbol{\theta}))=\frac{\lambda AB}{(n+1)(m+1)}\sum_{i=0}^{n}\sum_{j=0}^{m}|\Delta(u((x_i,y_j);\boldsymbol{\omega},\boldsymbol{\theta}))|^p+$$
$$+\frac{2(1-\lambda)(A+B)}{2(n+m)}\sum_{i=1}^{2(n+m)}|u(z_i;\boldsymbol{\omega},\boldsymbol{\theta})-G(z_i)|^p,$$

where, to simplify the notation, we renamed the nodes on $\partial\Omega$ with $z_i, \ \ i=1,\dots,2(n+m)$.

2.6 Neural Network methods for Solving Reaction-Diffusion models

2.6.1 Introduction

We have already mentioned that a reaction-diffusion model deals with the evolution, over time, of a population of individuals residing within an open spatial region $\Omega\subset\mathbb{R}^d$. Let us then consider the problem

$$\textbf{(2.36)}\quad \begin{cases} \frac{\partial}{\partial t}u(\boldsymbol{x},t)-D\,\Delta u(\boldsymbol{x},t)=f(u(\boldsymbol{x},t)), \ in \ \Omega\times(0,T), \\ u(\boldsymbol{x},0)=u_0(\boldsymbol{x}), \ \ on \ \ \overline{\Omega}\times\{0\}, \\ u(\boldsymbol{x},t)=b(\boldsymbol{x},t), \ \ on \ \ \partial\Omega\times(0,T]. \end{cases}$$

Our goal, as usual, is to find a solution $\hat{u}(\boldsymbol{x},t)$ of our problem, minimizing a given Loss Function $L(u)$. We can rewrite the problem as:

(2.37)
$$\begin{cases} \frac{\partial}{\partial t}u(\boldsymbol{x},t)-D\Delta u(\boldsymbol{x},t)-f(u(\boldsymbol{x},t))=0, \ in\ \Omega\times(0,T), \\ u(\boldsymbol{x},0)-u_0(\boldsymbol{x})=0, \ on\ \ \overline{\Omega}\times\{0\}, \\ u(\boldsymbol{x},t)-b(\boldsymbol{x},t)=0, \ on\ \ \partial\Omega\times(0,T], \end{cases}$$

and, similarly to what was done previously, we define the Loss functionals

(2.38)
$$\begin{aligned} L_F^p(\boldsymbol{x},t,u)&=\int_0^T dt\int_\Omega \left|\frac{\partial}{\partial t}u(\boldsymbol{x},t)-D\Delta u(\boldsymbol{x},t)-f(u(\boldsymbol{x},t))\right|^p d\boldsymbol{x}, \\ L_I^p(\boldsymbol{x},u)&=\int_{\overline{\Omega}} \left|u(\boldsymbol{x},0)-u_0(\boldsymbol{x})\right|^p d\boldsymbol{x}, \\ L_B^p(\boldsymbol{x},t,u)&=\int_0^T dt\int_{\partial\Omega} |u(\boldsymbol{x},t)-b(\boldsymbol{x},t)|^p d\boldsymbol{x}. \end{aligned}$$

Obviously, our aim is to construct a single Loss Function that depends on the previous operators, then we place

(2.39) $L^p(\boldsymbol{x},t,u)=\lambda_1 L_F^p(\boldsymbol{x},t,u)+\lambda_2 L_I^p(\boldsymbol{x},u)+\lambda_3 L_B^p(\boldsymbol{x},t,u),$

where the parameters $\lambda_1,\lambda_2,\lambda_3$ are such that $\lambda_1+\lambda_2+\lambda_3=1$ and they reflect the fact that the three functionals L_F, L_I, L_B can be of different importance. Now, we need to build and train a Neural Network to help us approximate the solution to our problem. In this case, both the components x_i, of the point $\boldsymbol{x}\in\mathbb{R}^d$, and time t are given as input of the Neural Network. We observe that, using a derivable activation function (e.g., logistic or tanh type), it is possible to calculate

$$\frac{\partial}{\partial t}u(\boldsymbol{x},t;\boldsymbol{\omega},\boldsymbol{\theta}), \ \ \Delta u(\boldsymbol{x},t;\boldsymbol{\omega},\boldsymbol{\theta}).$$

Let $\{t_i\}_{i=0}^m$ uniformly sampled points on the interval $[0,T]$, $\{x_k\}_{k\in K}$ be uniformly sampled points on Ω and $\{\boldsymbol{y}_j\}_{j\in J}$ be the uniformly sampled points on $\partial\Omega$. Using the Monte Carlo method, we derive

$$\begin{aligned} L^p(u(\boldsymbol{x},t;\boldsymbol{\omega},\boldsymbol{\theta}))\approx&\frac{\lambda_1 T\|\Omega\|}{(m+1)|K|}\sum_{i=0}^m\sum_{k\in K}\left|\frac{\partial}{\partial t}u_{ki}-D\Delta u_{ki}-f(u_{ki})\right|^p \\ &+\frac{\lambda_2\|\Omega\|}{|K|}\sum_{k\in K}|u_{k0}-u_0(\boldsymbol{x}_k)|^p+\frac{\lambda_2\|\partial\Omega\|}{|J|}\sum_{j\in J}|\overline{u}_{j0}-u_0(\boldsymbol{y}_j)|^p \\ &+\frac{\lambda_3 T\|\partial\Omega\|}{m|J|}\sum_{i=0}^m\sum_{j\in J}|\overline{u}_{ji}-b_{ji}|^p, \end{aligned}$$

where

- $u=u(\boldsymbol{x},t;\boldsymbol{\omega},\boldsymbol{\theta})$;
- $u_{ki}=u(\boldsymbol{x}_k,t_i;\boldsymbol{\omega},\boldsymbol{\theta})$;
- $u_{k0}=u(\boldsymbol{x}_k,0;\boldsymbol{\omega},\boldsymbol{\theta})$;

- $\overline{u_{ji}} = u(\boldsymbol{y}_j, t_i; \boldsymbol{\omega}, \boldsymbol{\theta})$;
- $\overline{u_{j0}} = u(\boldsymbol{y}_j, 0; \boldsymbol{\omega}, \boldsymbol{\theta})$;
- $\overline{b_{ji}} = u(\boldsymbol{y}_j, t_i; \boldsymbol{\omega}, \boldsymbol{\theta})$.

2.6.2 Calculation of derivatives

Suppose we have a Neural Network defined by:

- $m+1$ input nodes $x_1, \dots, x_m, x_{m+1} = t$;
- N hidden layers, consisting respectively of $d_1, \dots, d_N$ nodes;
- one node output u;
- $\boldsymbol{\omega}^{(1)}$ are the weights of the input nodes;
- $\boldsymbol{\omega}^{(j)}$ are the weights of the hidden layer $j-1$, $j = 2, \dots, N+1$;
- $\boldsymbol{\theta}^{(j)}$ are the biases of the layer $j-1$, $j = 1, \dots, N+1$;

For simplicity of notation we have posed $t = x_{m+1}$ and we will denote by $\boldsymbol{X}^{(0)} = (x_1, \dots, x_{m+1})^T$ the input vector.

We can schematize the operation of the Neural Network as follows

$$
\begin{aligned}
&\boldsymbol{X}^{(0)} = (x_1, \dots, x_{m+1})^T, \\
&\boldsymbol{z}^{(1)} = \boldsymbol{\omega}^{(1)} \boldsymbol{X}^{(0)} + \boldsymbol{\theta}^{(1)}, \\
&\boldsymbol{X}^{(1)} = \sigma(\boldsymbol{z}^{(1)}), \\
&\quad\vdots \\
&\boldsymbol{z}^{(k)} = \boldsymbol{\omega}^{(k)} \boldsymbol{X}^{(k-1)} + \boldsymbol{\theta}^{(k)}, \quad k = 2, \dots, N, \\
&\boldsymbol{X}^{(k)} = \sigma(\boldsymbol{z}^{(k)}), \quad k = 2, \dots, N, \\
&\quad\vdots \\
&\boldsymbol{z}^{(N+1)} = \boldsymbol{\omega}^{(N+1)} \boldsymbol{X}^{(N)} + \boldsymbol{\theta}^{(N+1)}, \\
&u = \boldsymbol{z}^{(N+1)}.
\end{aligned}
\tag{2.41}
$$

Then, our Neural Network can be seen as

$$N_\sigma = \boldsymbol{z}^{(N+1)} \circ \sigma \circ \boldsymbol{z}^{(N)} \circ \sigma \circ \dots \circ \sigma \circ \boldsymbol{z}^{(2)} \circ \sigma \circ \boldsymbol{z}^{(1)},$$

that is $u(\boldsymbol{x}, t; \boldsymbol{\omega}, \boldsymbol{\theta}) = \boldsymbol{z}^{(N+1)}(\sigma(\boldsymbol{z}^{(N)}(\sigma(\dots(\sigma(\boldsymbol{z}^{(1)}))\dots))))$.

To compute the Loss Function, we need to calculate

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

Remark

Let us observe that

$$\frac{\partial z_i^{(k)}}{\partial x_j^{(k-1)}} = \delta_{mj}\omega_{im}^{(k)} = \omega_{ij}^{(k)}, \quad k=1,\dots,N+1, i=1,\dots,d_k, j=1,\dots,d_{k-1}.$$

Because of what was said in the section on automatic differentiation, we get

$$\frac{\partial u}{\partial x_j} = \frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(z^{(N)})\frac{\partial z^{(N)}}{\partial \boldsymbol{X}^{(N-1)}}\dots\frac{\partial z^{(2)}}{\partial \boldsymbol{X}^{(1)}}\sigma'(z^{(1)})\frac{\partial z^{(1)}}{\partial x_j}, \quad j=1,\dots,m+1.$$

where $\frac{\partial z^{(1)}}{\partial x_j} = \boldsymbol{\omega}^{(1)}[:, j]$, representing the j-th column of the weight matrix $\boldsymbol{\omega}^{(1)}$.

In addition, from the above observation, it follows that

$$\textbf{(2.41)} \quad \frac{\partial z^{(k)}}{\partial \boldsymbol{X}^{(k-1)}} = \boldsymbol{\omega}^{(k)}, \quad k=1,\dots,N+1,$$

from which

$$\textbf{(2.42)} \quad \frac{\partial u}{\partial x_j} = \boldsymbol{\omega}^{(N+1)}\sigma'(z^{(N)})\odot\boldsymbol{\omega}^{(N)}\dots\omega^{(2)}\sigma'(z^{(1)})\odot\boldsymbol{\omega}^{(1)}[:, j], \quad j=1,\dots,m+1,$$

where $\boldsymbol{a}\odot\boldsymbol{b}$ represents the componentwise multiplication.

Now, to compute Δu, we need to calculate the second derivatives:

$$\begin{aligned}\textbf{(2.43)} \quad \frac{\partial^2 u}{\partial x_j^2} &= \frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\frac{\partial\,\sigma'(z^{(N)})}{\partial x_j}\frac{\partial z^{(N)}}{\partial \boldsymbol{X}^{(N-1)}}\sigma'(z^{(N-1)})\dots\frac{\partial z^{(2)}}{\partial \boldsymbol{X}^{(1)}}\sigma'(z^{(1)})\frac{\partial z^{(1)}}{\partial x_j} \\ &+ \frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(z^{(N)})\frac{\partial z^{(N)}}{\partial \boldsymbol{X}^{(N-1)}}\frac{\partial\,\sigma'(z^{(N-1)})}{\partial x_j}\dots\frac{\partial z^{(2)}}{\partial \boldsymbol{X}^{(1)}}\sigma'(z^{(1)})\frac{\partial z^{(1)}}{\partial x_j} \\ &+ \dots + \\ &+ \frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(z^{(N)})\frac{\partial z^{(N)}}{\partial \boldsymbol{X}^{(N-1)}}\sigma'(z^{(N-1)})\dots\frac{\partial z^{(2)}}{\partial \boldsymbol{X}^{(1)}}\frac{\partial\,\sigma'(z^{(1)})}{\partial x_j}\frac{\partial z^{(1)}}{\partial x_j}.\end{aligned}$$

Proceeding with the calculations we obtain that

$$\textbf{(2.44)} \quad \frac{\partial\,\sigma'(z^{(k)})}{\partial x_j} = \sigma''(z^{(k)})\frac{\partial z^{(k)}}{\partial x_j} = \sigma''(z^{(k)})\frac{\partial z^{(k)}}{\partial \boldsymbol{X}^{(k-1)}}\frac{\partial \boldsymbol{X}^{(k-1)}}{\partial x_j},$$

where, if we use the logistic activation function, we set

$$\textbf{(2.45)} \quad \sigma''(z^{(k)}) = \sigma(z^{(k)})\odot(1-\sigma(z^{(k)}))\odot(1-2\,\sigma(z^{(k)})).$$

Observing that

$$\frac{\partial \boldsymbol{X}^{(k)}}{\partial x_j} = \frac{\partial\,\sigma(z^{(k)})}{\partial x_j} = \sigma'(z^{(k)})\frac{\partial z^{(k)}}{\partial x_j} = \sigma'(z^{(k)})\frac{\partial z^{(k)}}{\partial \boldsymbol{X}^{(k-1)}}\dots\sigma'(z^{(1)})\frac{\partial z^{(1)}}{\partial x_j},$$

we derive

$$\frac{\partial \sigma'(\boldsymbol{z}^{(k)})}{\partial x_j} = \sigma''(\boldsymbol{z}^{(k)}) \frac{\partial \boldsymbol{z}^{(k)}}{\partial \boldsymbol{X}^{(k-1)}} \sigma'(\boldsymbol{z}^{(k-1)}) \frac{\partial \boldsymbol{z}^{(k-1)}}{\partial \boldsymbol{X}^{(k-2)}} \dots \sigma'(\boldsymbol{z}^{(1)}) \frac{\partial \boldsymbol{z}^{(1)}}{\partial x_j}.$$

Remark

It is useful to define derivatives recursively as:

$$\frac{\partial \boldsymbol{X}^{(1)}}{\partial x_j} = \sigma'(\boldsymbol{z}^{(1)}) \odot \boldsymbol{\omega}^{(1)}[:, j],$$

$$\frac{\partial \boldsymbol{X}^{(2)}}{\partial x_j} = \sigma'(\boldsymbol{z}^{(2)}) \odot \left(\boldsymbol{\omega}^{(2)} \frac{\partial \boldsymbol{X}^{(1)}}{\partial x_j} \right),$$

$$\vdots$$

$$\frac{\partial \boldsymbol{X}^{(k)}}{\partial x_j} = \sigma'(\boldsymbol{z}^{(k)}) \odot \left(\boldsymbol{\omega}^{(k)} \frac{\partial \boldsymbol{X}^{(k-1)}}{\partial x_j} \right),$$

$$\vdots$$

$$\frac{\partial \boldsymbol{X}^{(N)}}{\partial x_j} = \sigma'(\boldsymbol{z}^{(N)}) \odot \left(\boldsymbol{\omega}^{(N)} \frac{\partial \boldsymbol{X}^{(N-1)}}{\partial x_j} \right),$$

$$\frac{\partial u}{\partial x_j} = \boldsymbol{\omega}^{(N+1)} \frac{\partial \boldsymbol{X}^{(N)}}{\partial x_j},$$

where $\sigma'(\boldsymbol{z}^{(k)}) = \sigma(\boldsymbol{z}^{(k)}) \odot (1 - \sigma(\boldsymbol{z}^{(k)}))$.

This recursive method is very practical, as it allows us to "easily" derive the second derivatives as well, in fact

$$\frac{\partial^2 \boldsymbol{X}^{(1)}}{\partial x_j^2} = \sigma''(\boldsymbol{z}^{(1)}) \odot (\boldsymbol{\omega}^{(1)}[:, j])^{\odot 2},$$

$$\vdots$$

$$\frac{\partial^2 \boldsymbol{X}^{(k)}}{\partial x_j^2} = \sigma''(\boldsymbol{z}^{(k)}) \odot \left(\boldsymbol{\omega}^{(k)} \frac{\partial \boldsymbol{X}^{(k-1)}}{\partial x_j} \right)^{\odot 2} + \sigma'(\boldsymbol{z}^{(k)}) \odot \left(\boldsymbol{\omega}^{(k)} \frac{\partial^2 \boldsymbol{X}^{(k-1)}}{\partial x_j^2} \right),$$

$$\vdots$$

$$\frac{\partial^2 \boldsymbol{X}^{(N)}}{\partial x_j^2} = \sigma''(\boldsymbol{z}^{(N)}) \odot \left(\boldsymbol{\omega}^{(N)} \frac{\partial \boldsymbol{X}^{(N-1)}}{\partial x_j} \right)^{\odot 2} + \sigma'(\boldsymbol{z}^{(N)}) \odot \left(\boldsymbol{\omega}^{(N)} \frac{\partial^2 \boldsymbol{X}^{(N-1)}}{\partial x_j^2} \right),$$

$$\frac{\partial^2 u}{\partial x_j^2} = \boldsymbol{\omega}^{(N+1)} \frac{\partial^2 \boldsymbol{X}^{(N)}}{\partial x_j^2}.$$

Consequently, we obtain that

$$\Delta u=\sum_{j=1}^{m}\boldsymbol{\omega}^{(N+1)}\frac{\partial^2\boldsymbol{X}^{(N)}}{\partial x_j^2}$$

where $\frac{\partial^2\boldsymbol{X}^{(N)}}{\partial x_i^2}$ was calculated iteratively using the procedure defined above.

2.6.3 Calculation of $\frac{\partial L^2}{\partial \omega_{ij}^{(k)}}, \frac{\partial L^2}{\partial \theta_i^{(k)}}$

By placing

$$L_1=\left(u_t-D\Delta u-f(u)\right)^2,\ L_2=\left(u(\boldsymbol{x},0)-u_0(\boldsymbol{x})\right)^2,\ L_3=(u-b)^2,$$

we can define the Loss Function as

(2.46) $$L^2(u,u_t,\Delta u)=\lambda_1\int_0^T dt\int_\Omega L_1\ d\boldsymbol{x}+\lambda_2\int_\Omega L_2\ d\boldsymbol{x}+\lambda_3\int_0^T dt\int_{\partial\Omega} L_3\ d\boldsymbol{s},$$

where

- $u=u(\boldsymbol{x},t;\boldsymbol{\omega},\boldsymbol{\theta})$;
- $u(\boldsymbol{x},0)=u(\boldsymbol{x},t=0;\boldsymbol{\omega},\boldsymbol{\theta})$;
- $b=b(\boldsymbol{x},t)$;
- $u_t,\Delta u$ were calculated using the procedures shown above.

As we mentioned earlier, minimizing the Loss Function is equivalent to minimizing, for each point x in the training set, the quantities

$$L_1=\left(u_t-D\Delta u-f(u)\right)^2,\ L_2=\left(u(\boldsymbol{x},0)-u_0(\boldsymbol{x})\right)^2,\ L_3=(u-b)^2.$$

So, studying $\frac{\partial L^2}{\partial \omega_{ij}^{(k)}}$ is equivalent to studying:

- $\frac{\partial L_1}{\partial \omega_{ij}^{(k)}}=\frac{\partial L_1}{\partial \boldsymbol{z}^{(k)}}\frac{\partial \boldsymbol{z}^{(k)}}{\partial \omega_{ij}^{(k)}}=2\left(u_t-D\Delta u-f(u)\right)\left[\frac{\partial u_t}{\partial \boldsymbol{z}^{(k)}}-D\frac{\partial \Delta u}{\partial \boldsymbol{z}^{(k)}}-f'(u)\frac{\partial u}{\partial \boldsymbol{z}^{(k)}}\right]\frac{\partial \boldsymbol{z}^{(k)}}{\partial \omega_{ij}^{(k)}}$;
- $\frac{\partial L_2}{\partial \omega_{ij}^{(k)}}=\frac{\partial L_2}{\partial \boldsymbol{z}^{(k)}}\frac{\partial \boldsymbol{z}^{(k)}}{\partial \omega_{ij}^{(k)}}=2\left(u(\boldsymbol{x},0)-u_0(\boldsymbol{x})\right)\left[\left.\frac{\partial u}{\partial \boldsymbol{z}^{(k)}}\right|_{t=0}\right]\frac{\partial \boldsymbol{z}^{(k)}}{\partial \omega_{ij}^{(k)}}$;
- $\frac{\partial L_3}{\partial \omega_{ij}^{(k)}}=\frac{\partial L_3}{\partial \boldsymbol{z}^{(k)}}\frac{\partial \boldsymbol{z}^{(k)}}{\partial \omega_{ij}^{(k)}}=2(u-b)\frac{\partial u}{\partial \boldsymbol{z}^{(k)}}\frac{\partial \boldsymbol{z}^{(k)}}{\partial \omega_{ij}^{(k)}}$.

Similar considerations can be made for $\frac{\partial L^2}{\partial \theta_i^{(k)}}$.

Let's start by calculating

$$\frac{\partial u}{\partial \boldsymbol{z}^{(k)}}=\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(\boldsymbol{z}^{(N)})\frac{\partial \boldsymbol{z}^{(N)}}{\partial \boldsymbol{X}^{(N-1)}}\dots\frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\sigma'(\boldsymbol{z}^{(k)}).$$

Let us observe that

$$z_r^{(k)}=\omega_{rs}^{(k)}x_s^{(k-1)}+\theta_r^{(k)},$$

from which

$$\frac{\partial z_r^{(k)}}{\partial \omega_{ij}^{(k)}}=\delta_{ri}\delta_{sj}x_s^{(k-1)}=\delta_{ri}x_j^{(k-1)},$$

that is

$$\frac{\partial z_i^{(k)}}{\partial \omega_{ij}^{(k)}}=x_j^{(k-1)},\quad and\quad \frac{\partial z_r^{(k)}}{\partial \omega_{ij}^{(k)}}=0,\quad \forall\, r\neq i,$$

which implies, in a compact form, that

$$\textbf{(2.48)}\quad \frac{\partial \boldsymbol{z}^{(k)}}{\partial \omega_{ij}^{(k)}}=[0,\dots,0,x_j^{(k-1)},0,\dots,0]^T,$$

with $x_j^{(k-1)}$ in the i-th position. Similarly, it is shown that

$$\textbf{(2.49)}\quad \frac{\partial \boldsymbol{z}^{(k)}}{\partial \theta_i^{(k)}}=[0,\dots,0,1,0,\dots,0]^T,$$

with 1 in the i-th position.

Now, for all $r=1,\dots,m+1,$ let's calculate

$$\begin{aligned}
\frac{\partial u_{x_r}}{\partial \boldsymbol{z}^{(k)}}&=\frac{\partial}{\partial \boldsymbol{z}^{(k)}}\frac{\partial u}{\partial x_r}=\frac{\partial}{\partial x_r}\frac{\partial u}{\partial \boldsymbol{z}^{(k)}}\\
&=\frac{\partial}{\partial x_r}\left(\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(\boldsymbol{z}^{(N)})\frac{\partial \boldsymbol{z}^{(N)}}{\partial \boldsymbol{X}^{(N-1)}}\dots\frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\sigma'(\boldsymbol{z}^{(k)})\right)\\
&=\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\frac{\partial \sigma'(\boldsymbol{z}^{(N)})}{\partial x_r}\frac{\partial \boldsymbol{z}^{(N)}}{\partial \boldsymbol{X}^{(N-1)}}\sigma'(\boldsymbol{z}^{(N-1)})\dots\frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\sigma'(\boldsymbol{z}^{(k)})\\
&\qquad\qquad +\dots+\\
&+\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(\boldsymbol{z}^{(N)})\dots\frac{\partial \boldsymbol{z}^{(s+1)}}{\partial \boldsymbol{X}^{(s)}}\frac{\partial \sigma'(\boldsymbol{z}^{(s)})}{\partial x_r}\frac{\partial \boldsymbol{z}^{(s)}}{\partial \boldsymbol{X}^{(s-1)}}\sigma'(\boldsymbol{z}^{(s-1)})\dots\frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\sigma'(\boldsymbol{z}^{(k)})\\
&\qquad\qquad +\dots+\\
&+\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(\boldsymbol{z}^{(N)})\frac{\partial \boldsymbol{z}^{(N)}}{\partial \boldsymbol{X}^{(N-1)}}\sigma'(\boldsymbol{z}^{(N-1)})\dots\frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\frac{\partial \sigma'(\boldsymbol{z}^{(k)})}{\partial x_r},
\end{aligned}$$

where

$$\frac{\partial \sigma'(z^{(s)})}{\partial x_r} = \sigma''(z^{(s)})\frac{\partial z^{(s)}}{\partial x_r} = \sigma''(z^{(s)})\frac{\partial z^{(s)}}{\partial \boldsymbol{X}^{(s-1)}}\frac{\partial \boldsymbol{X}^{(s-1)}}{\partial x_r}$$
$$= \sigma''(z^{(s)})\frac{\partial z^{(s)}}{\partial \boldsymbol{X}^{(s-1)}}\sigma'(z^{(s-1)})\frac{\partial z^{(s-1)}}{\partial x_r}$$
$$= \sigma''(z^{(s)})\frac{\partial z^{(s)}}{\partial \boldsymbol{X}^{(s-1)}}\sigma'(z^{(s-1)})\frac{\partial z^{(s-1)}}{\partial \boldsymbol{X}^{(s-2)}}\sigma'(z^{(s-2)})\ldots\sigma'(z^{(1)})\frac{\partial z^{(1)}}{\partial x_r}.$$

Remark

We observe that

$$\frac{\partial \sigma'(z^{(s)})}{\partial x_r} = \sigma''(z^{(s)})\frac{\partial z^{(s)}}{\partial x_r},$$

then we obtain the recursive formula

$$\frac{\partial \sigma'(z^{(s+1)})}{\partial x_r} = \sigma''(z^{(s+1)})\frac{\partial z^{(s+1)}}{\partial x_r} = \sigma''(z^{(s+1)})\frac{\partial z^{(s+1)}}{\partial \boldsymbol{X}^{(s)}}\sigma'(z^{(s)})\frac{\partial z^{(s)}}{\partial x_r}.$$

Now, let us consider

$$\textbf{(2.50)}\quad \frac{\partial \Delta u}{\partial z^{(k)}} = \frac{\partial}{\partial z^{(k)}}\sum_{r=1}^{m}\frac{\partial^2 u}{\partial x_r^2} = \sum_{r=1}^{m}\frac{\partial^2}{\partial x_r^2}\left(\frac{\partial u}{\partial z^{(k)}}\right),$$

Then we can study

$$\textbf{(2.51)}\quad \begin{aligned}
&\frac{\partial^2}{\partial x_r^2}\left(\frac{\partial u}{\partial z^{(k)}}\right) = \frac{\partial}{\partial x_r}\left(\frac{\partial u_{x_r}}{\partial z^{(k)}}\right)\\
&= \frac{\partial}{\partial x_r}\left(\frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\frac{\partial \sigma'(z^{(N)})}{\partial x_r}\frac{\partial z^{(N)}}{\partial \boldsymbol{X}^{(N-1)}}\sigma'(z^{(N-1)})\ldots\frac{\partial z^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\sigma'(z^{(k)})\right.\\
&\quad \overset{+\ldots+}{+}\frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(z^{(N)})\ldots\frac{\partial z^{(s+1)}}{\partial \boldsymbol{X}^{(s)}}\frac{\partial \sigma'(z^{(s)})}{\partial x_r}\frac{\partial z^{(s)}}{\partial \boldsymbol{X}^{(s-1)}}\sigma'(z^{(s-1)})\ldots\frac{\partial z^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\sigma'(z^{(k)})\\
&\quad \left.\overset{+\ldots+}{+}\frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(z^{(N)})\frac{\partial z^{(N)}}{\partial \boldsymbol{X}^{(N-1)}}\sigma'(z^{(N-1)})\ldots\frac{\partial z^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\frac{\partial \sigma'(z^{(k)})}{\partial x_r}\right).
\end{aligned}$$

By using the linearity of the derivative, we can calculate, for example

$$\begin{aligned}
&\frac{\partial}{\partial x_r}\left(\frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(z^{(N)})\ldots\frac{\partial z^{(s+1)}}{\partial \boldsymbol{X}^{(s)}}\frac{\partial \sigma'(z^{(s)})}{\partial x_r}\frac{\partial z^{(s)}}{\partial \boldsymbol{X}^{(s-1)}}\sigma'(z^{(s-1)})\ldots\frac{\partial z^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\sigma'(z^{(k)})\right)\\
&= \frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\frac{\partial \sigma'(z^{(N)})}{\partial x_r}\ldots\frac{\partial z^{(s+1)}}{\partial \boldsymbol{X}^{(s)}}\frac{\partial \sigma'(z^{(s)})}{\partial x_r}\frac{\partial z^{(s)}}{\partial \boldsymbol{X}^{(s-1)}}\sigma'(z^{(s-1)})\ldots\frac{\partial z^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\sigma'(z^{(k)})\\
&\quad \overset{+\ldots+}{+}\frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(z^{(N)})\ldots\frac{\partial z^{(s+1)}}{\partial \boldsymbol{X}^{(s)}}\frac{\partial^2 \sigma'(z^{(s)})}{\partial x_r^2}\frac{\partial z^{(s)}}{\partial \boldsymbol{X}^{(s-1)}}\sigma'(z^{(s-1)})\ldots\frac{\partial z^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\sigma'(z^{(k)})\\
&\quad \overset{+\ldots+}{+}\frac{\partial z^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(z^{(N)})\ldots\frac{\partial z^{(s+1)}}{\partial \boldsymbol{X}^{(s)}}\frac{\partial \sigma'(z^{(s)})}{\partial x_r}\frac{\partial z^{(s)}}{\partial \boldsymbol{X}^{(s-1)}}\sigma'(z^{(s-1)})\ldots\frac{\partial z^{(k+1)}}{\partial \boldsymbol{X}^{(k)}}\frac{\partial \sigma'(z^{(k)})}{\partial x_r},
\end{aligned}$$

where,

$$\textbf{(2.52)} \quad \frac{\partial^2 \sigma'(z^{(s)})}{\partial x_r^{\,2}} = \frac{\partial}{\partial x_r}\left(\frac{\partial \sigma'(z^{(s)})}{\partial x_r}\right) = \frac{\partial}{\partial x_r}\left(\sigma''(z^{(s)})\frac{\partial z^{(s)}}{\partial x_r}\right)$$
$$= \sigma'''(z^{(s)})\frac{\partial z^{(s)}}{\partial x_r} + \sigma''(z^{(s)})\frac{\partial^2 z^{(s)}}{\partial x_r^{\,2}},$$

with

$$\textbf{(2.53)} \quad \sigma'''(z^{(s)}) = \sigma(z^{(s)}) \odot (1-\sigma(z^{(s)})) \odot (1-6\,\sigma(z^{(s)}) \odot (1-\sigma(z^{(s)}))),$$

in the case of using the logistic activation function.

Substituting first in **(2.51)** and then in **(2.50)** gives the expression of $\frac{\partial \Delta u}{\partial z^{(k)}}$.

Remark

Minimizing, for each point x in the training set, the quantities

$$L_1 = \left(u_t - D\Delta u - f(u)\right)^2,\ L_2 = \left(u(\boldsymbol{x}, 0) - u_0(\boldsymbol{x})\right)^2,\ L_3 = (u-b)^2,$$

is better than minimizing the Loss Function. In fact, let us suppose that we want to minimize

$$L^2(u, u_t, \Delta u) = \lambda_1 \int_0^T dt \int_\Omega L_1 \; d\boldsymbol{x} + \lambda_2 \int_\Omega L_2 \; d\boldsymbol{x} + \lambda_3 \int_0^T dt \int_{\partial\Omega} L_3 \; d\boldsymbol{s},$$

then its derivative is

$$\frac{\partial L^2}{\partial \omega_{ij}^{(k)}} = \lambda_1 \int_0^T dt \int_\Omega \frac{\partial L_1}{\partial \omega_{ij}^{(k)}} \; d\boldsymbol{x} + \lambda_2 \int_\Omega \frac{\partial L_2}{\partial \omega_{ij}^{(k)}} \; d\boldsymbol{x} + \lambda_3 \int_0^T dt \int_{\partial\Omega} \frac{\partial L_3}{\partial \omega_{ij}^{(k)}} \; d\boldsymbol{s}.$$

If we now apply the gradient method, our Neural Network would update its weights based on the total error and not from the error produced by the individual point. This implies that the Neural Network learns that a particular configuration of points generates a particular error, without understanding from which specific point the error comes.

2.6.4 Recursive representation

From a purely computational point of view, we want to simplify the work for the calculator by preventing it from recalculating terms that have already been calculated. We set

$$\textbf{(2.54)}\quad \alpha^{(k)}=\frac{\partial u}{\partial z^{(k)}},\quad \beta_r^{(k)}=\frac{\partial u_{x_r}}{\partial z^{(k)}},\ r=1,\dots,m+1,\quad \gamma^{(k)}=\frac{\partial \Delta u}{\partial z^{(k)}},$$

and we want to find recursive formulas to connect

- $\alpha^{(k)}$ *with* $\alpha^{(k+1)}$,
- $\beta_r^{(k)}$ *with* $\beta_r^{(k+1)}$, $r=1,\dots,m+1$,
- $\gamma^{(k)}$ *with* $\gamma^{(k+1)}$.

Observing that

$$\beta_r^{(k)}=\frac{\partial}{\partial x_r}\left(\alpha^{(k)}\right),\ r=1,\dots,m+1,\ \textit{and}\ \gamma^{(k)}=\sum_{r=1}^{m}\frac{\partial}{\partial x_r}\left(\beta_r^{(k)}\right),$$

and by using hidden layers of the same dimensions, proceeding with the calculations, we obtain that

- $$\alpha^{(k)}=\frac{\partial u}{\partial z^{(k)}}=\frac{\partial z^{(N+1)}}{\partial X^{(N)}}\sigma'\left(z^{(N)}\right)\frac{\partial z^{(N)}}{\partial X^{(N-1)}}\dots\sigma'\left(z^{(k+1)}\right)\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right)$$
  $$=\frac{\partial u}{\partial z^{(k+1)}}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right)=\alpha^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right);$$

- $$\beta_r^{(k)}=\frac{\partial}{\partial x_r}\left(\alpha^{(k)}\right)=\frac{\partial}{\partial x_r}\left(\alpha^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right)\right)$$
  $$=\frac{\partial}{\partial x_r}\left(\alpha^{(k+1)}\right)\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right)+\alpha^{(k+1)}\frac{\partial}{\partial x_r}\left(\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right)\right)$$
  $$=\beta_r^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right)+\alpha^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial \sigma'\left(z^{(k)}\right)}{\partial x_r},\quad r=1,\dots,m+1;$$

- $$\gamma^{(k)}=\sum_{r=1}^{m}\frac{\partial}{\partial x_r}\left(\beta_r^{(k)}\right)=\sum_{r=1}^{m}\frac{\partial}{\partial x_r}\left(\beta_r^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right)+\alpha^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial \sigma'\left(z^{(k)}\right)}{\partial x_r}\right)$$
  $$=\sum_{r=1}^{m}\left[\frac{\partial}{\partial x_r}\left(\beta_r^{(k+1)}\right)\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right)+\beta_r^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial \sigma'\left(z^{(k)}\right)}{\partial x_r}\right.$$
  $$\left.+\frac{\partial}{\partial x_r}\left(\alpha^{(k+1)}\right)\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial \sigma'\left(z^{(k)}\right)}{\partial x_r}+\alpha^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial^2 \sigma'\left(z^{(k)}\right)}{\partial x_r^2}\right]$$
  $$=\sum_{r=1}^{m}\left[\frac{\partial}{\partial x_r}\left(\beta_r^{(k+1)}\right)\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right)+\beta_r^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial \sigma'\left(z^{(k)}\right)}{\partial x_r}\right.$$
  $$\left.+\beta_r^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial \sigma'\left(z^{(k)}\right)}{\partial x_r}+\alpha^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial^2 \sigma'\left(z^{(k)}\right)}{\partial x_r^2}\right]$$
  $$=\gamma^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'\left(z^{(k)}\right)+2\sum_{r=1}^{m}\beta_r^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial \sigma'\left(z^{(k)}\right)}{\partial x_r}+\alpha^{(k+1)}\sum_{r=1}^{m}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial^2 \sigma'\left(z^{(k)}\right)}{\partial x_r^2}.$$

Remark

It may also be useful to find a numerical method to calculate

- $\frac{\partial \sigma'(\boldsymbol{z}^{(s)})}{\partial x_r}, \quad r=1,\dots,m+1;$
- $\frac{\partial^2 \sigma'(\boldsymbol{z}^{(s)})}{\partial x_r^2}, \quad r=1,\dots,m+1.$

We observe that

$$\frac{\partial \sigma'(\boldsymbol{z}^{(s)})}{\partial x_r}=\sigma''(\boldsymbol{z}^{(s)})\frac{\partial \boldsymbol{z}^{(s)}}{\partial x_r}=\sigma''(\boldsymbol{z}^{(s)})\odot\left(\boldsymbol{\omega}^{(k)}\frac{\partial \sigma(\boldsymbol{z}^{(s-1)})}{\partial x_r}\right)=\sigma''(\boldsymbol{z}^{(s)})\odot\left(\boldsymbol{\omega}^{(k)}\frac{\partial \boldsymbol{X}^{(s-1)}}{\partial x_r}\right),$$

where

- $\frac{\partial \boldsymbol{X}^{(s-1)}}{\partial x_r}=\boldsymbol{\omega}^{(s-1)}\sigma'(\boldsymbol{z}^{(s-2)})\odot\boldsymbol{\omega}^{(s-2)}\dots\omega^{(2)}\sigma'(\boldsymbol{z}^{(1)})\odot\boldsymbol{\omega}^{(1)}[:,r], \quad r=1,\dots,m+1,$
- $\sigma'(\boldsymbol{z}^{(k)})=\sigma(\boldsymbol{z}^{(k)})\odot(1-\sigma(\boldsymbol{z}^{(k)})),$
- $\sigma''(\boldsymbol{z}^{(k)})=\sigma(\boldsymbol{z}^{(k)})\odot(1-\sigma(\boldsymbol{z}^{(k)}))\odot(1-2\,\sigma(\boldsymbol{z}^{(k)})),$

in the case of the sigmoid activation function.

Moreover, we have that

$$\begin{aligned}\frac{\partial^2 \sigma'(\boldsymbol{z}^{(s)})}{\partial x_r^2}&=\frac{\partial}{\partial x_r}\left(\frac{\partial \sigma'(\boldsymbol{z}^{(s)})}{\partial x_r}\right)=\frac{\partial}{\partial x_r}\left(\sigma''(\boldsymbol{z}^{(s)})\frac{\partial \boldsymbol{z}^{(s)}}{\partial x_r}\right)=\sigma'''(\boldsymbol{z}^{(s)})\frac{\partial \boldsymbol{z}^{(s)}}{\partial x_r}+\sigma''(\boldsymbol{z}^{(s)})\frac{\partial^2 \boldsymbol{z}^{(s)}}{\partial x_r^2}\\&=\sigma'''(\boldsymbol{z}^{(s)})\odot\left(\boldsymbol{\omega}^{(s)}\frac{\partial \sigma(\boldsymbol{z}^{(s-1)})}{\partial x_r}\right)+\sigma''(\boldsymbol{z}^{(s)})\left(\boldsymbol{\omega}^{(s)}\frac{\partial^2 \sigma(\boldsymbol{z}^{(s-1)})}{\partial x_r^{\,2}}\right)\\&=\sigma'''(\boldsymbol{z}^{(s)})\odot\left(\boldsymbol{\omega}^{(s)}\frac{\partial \boldsymbol{X}^{(s-1)}}{\partial x_r}\right)+\sigma''(\boldsymbol{z}^{(s)})\left(\boldsymbol{\omega}^{(s)}\frac{\partial^2 \boldsymbol{X}^{(s-1)}}{\partial x_r^{\,2}}\right),\end{aligned}$$

where

- $\frac{\partial \boldsymbol{X}^{(s-1)}}{\partial x_r}$ is the same as above;
- $\frac{\partial^2 \boldsymbol{X}^{(s-1)}}{\partial x_r^{\,2}}$ is calculate by using the recursively procedure
  - $\frac{\partial^2 \boldsymbol{X}^{(k)}}{\partial x_j^2}=\sigma''(\boldsymbol{z}^{(k)})\odot\left(\boldsymbol{\omega}^{(k)}\frac{\partial \boldsymbol{X}^{(k-1)}}{\partial x_j}\right)^{\odot 2}+\sigma'(\boldsymbol{z}^{(k)})\odot\left(\boldsymbol{\omega}^{(k)}\frac{\partial^2 \boldsymbol{X}^{(k-1)}}{\partial x_j^2}\right),$
  - $\frac{\partial^2 \boldsymbol{X}^{(1)}}{\partial x_j^2}=\sigma''(\boldsymbol{z}^{(1)})\odot(\boldsymbol{\omega}^{(1)}[:,j])^{\odot 2},$
- $\sigma'(\boldsymbol{z}^{(k)})$, $\sigma''(\boldsymbol{z}^{(k)})$ are the same as above,
- $\sigma'''(\boldsymbol{z}^{(s)})=\sigma(\boldsymbol{z}^{(s)})\odot(1-\sigma(\boldsymbol{z}^{(s)}))\odot(1-6\,\sigma(\boldsymbol{z}^{(s)})\odot(1-\sigma(\boldsymbol{z}^{(s)}))).$

2.6.5 How it works

The first step is to approximate L^2 by the Monte Carlo method:

$$\begin{aligned} L^2 \approx & \lambda_1 \frac{T\|\Omega\|}{|I||J|} \sum_{i\in I} \sum_{j\in J} L_1(x_i, t_j) \\ & + \lambda_2 \left[\frac{\|\Omega\|}{|I|} \sum_{i\in I} L_2(x_i, 0) + \frac{\|\partial\Omega\|}{|K|} \sum_{k\in K} L_2(\bar{x}_k, 0) \right] \\ & + \lambda_3 \frac{T\|\partial\Omega\|}{|K||J|} \sum_{k\in K} \sum_{j\in J} L_3(\bar{x}_k, t_j), \end{aligned}$$

where

- x_i are uniformly sampled points on Ω;
- $\bar{x}_k$ are uniformly sampled points on $\partial\Omega$;
- t_j are uniformly sampled points on $[0, T]$.

Then, we need to calculate

- $\frac{\partial L_1}{\partial \omega_{ij}^{(k)}} = \frac{\partial L_1}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial \omega_{ij}^{(k)}} = 2\left(u_t - D\Delta u - f(u)\right)\left[\beta_{m+1}^{(k)} - D\gamma^{(k)} - f'(u)\alpha^{(k)}\right] \frac{\partial z^{(k)}}{\partial \omega_{ij}^{(k)}}$;
- $\frac{\partial L_2}{\partial \omega_{ij}^{(k)}} = \frac{\partial L_2}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial \omega_{ij}^{(k)}} = 2\left(u(\boldsymbol{x}, 0) - u_0(\boldsymbol{x})\right)\left[\alpha^{(k)}\big|_{t=0}\right] \frac{\partial z^{(k)}}{\partial \omega_{ij}^{(k)}}$;
- $\frac{\partial L_3}{\partial \omega_{ij}^{(k)}} = \frac{\partial L_2}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial \omega_{ij}^{(k)}} = 2(u - b)\left[\alpha^{(k)}\big|_{\partial\Omega}\right] \frac{\partial z^{(k)}}{\partial \omega_{ij}^{(k)}}$.

2.6.6 Updating weights process

To recapitulate, the process of updating weights is as follows:

- For each point (x, t) in the training set:
- Calculate

$$\begin{aligned} \alpha^{(N+1)} &= \frac{\partial u}{\partial z^{(N+1)}} = 1, \\ \beta_r^{(N+1)} &= \frac{\partial u_{x_r}}{\partial z^{(N+1)}} = 0,\ r = 1, \ldots, m+1, \\ \gamma^{(N+1)} &= \frac{\partial \Delta u}{\partial z^{(N+1)}} = 0, \end{aligned}$$

  evaluated at the point (x, t).

- Calculate

$$\frac{\partial L_1}{\partial \omega_{ij}^{(N+1)}},\ \frac{\partial L_2}{\partial \omega_{ij}^{(N+1)}},\ \frac{\partial L_3}{\partial \omega_{ij}^{(N+1)}},$$

evaluated at the point (x,t).

- Update the weights $\boldsymbol{\omega}^{(N+1)}$:

$$\omega_{ij}^{(N+1)}=\omega_{ij}^{(N+1)}-\eta\frac{\partial L_s}{\partial \omega_{ij}^{(N+1)}}.$$

Let us observe that the choice of which $\frac{\partial L_s}{\partial \omega_{ij}^{(N+1)}}$ to calculate depends on which point (x,t) has been fixed.

- Calculate

$$\begin{aligned}
\alpha^{(N)}&=\alpha^{(N+1)}\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(\boldsymbol{z}^{(N)})=\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(\boldsymbol{z}^{(N)}),\\
\beta_r^{(N)}&=\beta_r^{(N+1)}\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(\boldsymbol{z}^{(N)})+\alpha^{(N+1)}\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\frac{\partial \sigma'(\boldsymbol{z}^{(N)})}{\partial x_r}\\
&=\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\frac{\partial \sigma'(\boldsymbol{z}^{(N)})}{\partial x_r},\quad r=1,\dots,m+1,\\
\gamma^{(N)}&=\gamma^{(N+1)}\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\sigma'(\boldsymbol{z}^{(N)})+2\sum_{r=1}^{m}\beta_r^{(N+1)}\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\frac{\partial \sigma'(\boldsymbol{z}^{(N)})}{\partial x_r}\\
&\quad+\alpha^{(N+1)}\sum_{r=1}^{m}\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\frac{\partial^2 \sigma'(\boldsymbol{z}^{(N)})}{\partial x_r^2}\\
&=\sum_{r=1}^{m}\frac{\partial \boldsymbol{z}^{(N+1)}}{\partial \boldsymbol{X}^{(N)}}\frac{\partial^2 \sigma'(\boldsymbol{z}^{(N)})}{\partial x_r^2},
\end{aligned}$$

evaluated at the point (x,t).

- Calculate

$$\frac{\partial L_1}{\partial \omega_{ij}^{(N)}},\quad \frac{\partial L_2}{\partial \omega_{ij}^{(N)}},\quad \frac{\partial L_3}{\partial \omega_{ij}^{(N)}},$$

evaluated at the point (x,t).

- Update the weights $\boldsymbol{\omega}^{(N)}$:

$$\omega_{ij}^{(N)}=\omega_{ij}^{(N)}-\eta\frac{\partial L_s}{\partial \omega_{ij}^{(N)}}.$$

Let us observe that the choice of which $\frac{\partial L_s}{\partial \omega_{ij}^{(N)}}$ to calculate depends on which point (x,t) has been fixed.

…..

- Calculate

$$\alpha^{(k)}=\alpha^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'(z^{(k)}),$$

$$\beta_r^{(k)}=\beta_r^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'(z^{(k)})+\alpha^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial \sigma'(z^{(k)})}{\partial x_r},\quad r=1,\dots,m+1,$$

$$\gamma^{(k)}=\gamma^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'(z^{(k)})+2\sum_{r=1}^{m}\beta_r^{(k+1)}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial \sigma'(z^{(k)})}{\partial x_r}$$
$$+\alpha^{(k+1)}\sum_{r=1}^{m}\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial^2 \sigma'(z^{(k)})}{\partial x_r^2},$$

evaluated at the point (x,t).

- Calculate

$$\frac{\partial L_1}{\partial \omega_{ij}^{(k)}},\ \frac{\partial L_2}{\partial \omega_{ij}^{(k)}},\ \frac{\partial L_3}{\partial \omega_{ij}^{(k)}},$$

evaluated at the point (x,t).

- Update the weights $\omega^{(k)}$:

$$\omega_{ij}^{(k)}=\omega_{ij}^{(k)}-\eta\frac{\partial L_s}{\partial \omega_{ij}^{(k)}}.$$

Let us observe that the choice of which $\frac{\partial L_s}{\partial \omega_{ij}^{(k)}}$ to calculate depends on which point (x,t) has been fixed.

…..

- Calculate

$$\alpha^{(1)}=\alpha^{(2)}\frac{\partial z^{(2)}}{\partial X^{(1)}}\sigma'(z^{(1)}),$$

$$\beta_r^{(1)}=\beta_r^{(2)}\frac{\partial z^{(2)}}{\partial X^{(1)}}\sigma'(z^{(1)})+\alpha^{(2)}\frac{\partial z^{(2)}}{\partial X^{(1)}}\frac{\partial \sigma'(z^{(1)})}{\partial x_r},\quad r=1,\dots,m+1,$$

$$\gamma^{(1)}=\gamma^{(2)}\frac{\partial z^{(2)}}{\partial X^{(1)}}\sigma'(z^{(1)})+2\sum_{r=1}^{m}\beta_r^{(2)}\frac{\partial z^{(2)}}{\partial X^{(1)}}\frac{\partial \sigma'(z^{(1)})}{\partial x_r}$$
$$+\alpha^{(2)}\sum_{r=1}^{m}\frac{\partial z^{(2)}}{\partial X^{(1)}}\frac{\partial^2 \sigma'(z^{(1)})}{\partial x_r^2},$$

evaluated at the point (x,t).

- Calculate

$$\frac{\partial L_1}{\partial \omega_{ij}^{(1)}},\ \frac{\partial L_2}{\partial \omega_{ij}^{(1)}},\ \frac{\partial L_3}{\partial \omega_{ij}^{(1)}},$$

evaluated at the point (x,t).

- Update the weights $\omega^{(1)}$:

$$\omega_{ij}^{(l)} = \omega_{ij}^{(l)} - \eta \frac{\partial L_s}{\partial \omega_{ij}^{(l)}}.$$

Let us observe that the choice of which $\frac{\partial L_s}{\partial \omega_{ij}^{(l)}}$ to calculate depends on which point (x, t) has been fixed.

2.6.7 Algorithm (Backpropagation)

Input: N_{layer}, $\boldsymbol{\omega}^{(1)}, \ldots, \boldsymbol{\omega}^{(N_{layer}+1)}, \boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(N_{layer}+1)}$, $\eta > 0$, $L_{function}$.

Learning data: $\boldsymbol{X} = \{\boldsymbol{X}_k = (x_1, \ldots, x_m, x_{m+1} = t)_k\}_{k=1,\ldots,K}$.

Parameters: *error=0.1, errormax=0.001, threshold=1, thresholdmax=1000.*

while error > errormax && threshold < thresholdmax

for $k = 1$ to K

$\boldsymbol{X}_k^{(0)} = \boldsymbol{X}_k$;

for $j = 1$ to N_{layer}

$\boldsymbol{X}_k^{(j)} = \sigma(\boldsymbol{\omega}^{(j)} \boldsymbol{X}_k^{(j-1)} + \boldsymbol{\theta}^{(j)})$;

end

$\boldsymbol{u}_k = \boldsymbol{X}_k^{(N_{layer}+1)} = \boldsymbol{\omega}^{(N_{layer}+1)} \boldsymbol{X}_k^{(N_{layer})} + \boldsymbol{\theta}^{(N_{layer}+1)}$

$\alpha^{(N_{layer}+1)} = 1$;

$\beta_r^{(N_{layer}+1)} = 0$, $r = 1, \ldots, m+1$;

$\gamma^{(N_{layer}+1)} = 0$;

calculate $\frac{\partial L_s}{\partial \boldsymbol{z}^{(N+1)}}$;

$\boldsymbol{W}^{(N_{layer}+1)} = [\boldsymbol{\theta}^{(N_{layer}+1)}, \boldsymbol{\omega}^{(N_{layer}+1)}]$;

$\boldsymbol{W}^{(N_{layer}+1)} = \boldsymbol{W}^{(N_{layer}+1)} - \eta \frac{\partial L_s}{\partial \boldsymbol{z}^{(N+1)}} [1; \boldsymbol{X}^{(N_{layer})}]^T$;

for $k = N_{layer}$ to 1

$$\alpha^{(k)} = \alpha^{(k+1)} \frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}} \sigma'(\boldsymbol{z}^{(k)});$$

$$\beta_r^{(k)} = \beta_r^{(k+1)} \frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}} \sigma'(\boldsymbol{z}^{(k)}) + \alpha^{(k+1)} \frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}} \frac{\partial \sigma'(\boldsymbol{z}^{(k)})}{\partial x_r}, \quad r = 1, \ldots, m+1;$$

$$\gamma^{(k)} = \gamma^{(k+1)} \frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}} \sigma'(\boldsymbol{z}^{(k)}) + 2 \sum_{r=1}^{m} \beta_r^{(k+1)} \frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}} \frac{\partial \sigma'(\boldsymbol{z}^{(k)})}{\partial x_r} + \alpha^{(k+1)} \sum_{r=1}^{m} \frac{\partial \boldsymbol{z}^{(k+1)}}{\partial \boldsymbol{X}^{(k)}} \frac{\partial^2 \sigma'(\boldsymbol{z}^{(k)})}{\partial x_r^2};$$

calculate $\frac{\partial L_s}{\partial \boldsymbol{z}^{(k)}}$;

$\boldsymbol{W}^{(k)}=[\boldsymbol{\theta}^{(k)}, \boldsymbol{\omega}^{(k)}]$;

$\boldsymbol{W}^{(k)}=\boldsymbol{W}^{(k)}-\eta \frac{\partial L_s}{\partial \boldsymbol{z}^{(k)}}[1; \boldsymbol{X}^{(k-1)}]^T$;

end

end

$error=L^2(\boldsymbol{X}, \boldsymbol{\omega}, \boldsymbol{\theta})$;

mix the input vector $\boldsymbol{X}=\{\boldsymbol{X}_k=(x_1, \ldots, x_m, x_{m+1}=t)_k\}_{k=1, \ldots, K}$;

$threshold=threshold+1$;

end

Remark

For the choice of training set, we have two options:

1. In each epoch, the model is trained on the same training data, but the data are shuffled for each epoch. This shuffling helps ensure that the model does not simply learn the order of the data and can generalize better.
2. If it is possible to do so, at each epoch the training set is randomly generated within the definition range of our function. This process helps even more the generalization ability of the Neural Network, as it always learns from different points.

Remark

In the calculation

$$\textbf{(2.56)}\quad \boldsymbol{W}^{(j)}=\boldsymbol{W}^{(j)}-\eta \frac{\partial L_s}{\partial \boldsymbol{z}^{(k)}}[1; \boldsymbol{X}^{(j-1)}]^T$$

we added 1 to the vector $\boldsymbol{X}^{(j-1)}$ because the bias has to be taken into account.

2.7 Application to Fisher - Kolmogorov – Petrovsky - Piskunov Equation

2.7.1 Introduction

Let $x \in \Omega = [a, b]$ and $t \in [0, T]$, and consider the dimensionless Fisher-Kolmogorov-Petrovsky-Piskunov equation

$$y_t - y_{xx} = y(1-y),$$

For simplicity, we start by considering spatially homogeneous initial conditions of the type

$$y(x, 0) = y_0.$$

2.7.2 Spatially homogeneous case

Let us consider the spatially homogeneous case

$$\textbf{(2.57)} \quad \begin{cases} y_t = y(1-y), & in \ (0, T], \\ y(0) = y_0, & in \ t = 0, \end{cases}$$

with $y_0 = 1.5$, $T = 5$. As we have already seen in section 1.6, the solution is of the form

$$\textbf{(2.58)} \quad y(t) = y_0 \frac{e^t}{1 + y_0(e^t - 1)},$$

therefore, the population tends to reach the value 1, for $t \to +\infty$, see Figure 1.1.
In the spatially homogeneous case, the boundary conditions lose their meaning.
Now, let us consider the trial solution

$$\textbf{(2.59)} \quad \overline{y}(t) = y_0 + t\, u(t; \boldsymbol{\omega}, \boldsymbol{\theta}),$$

where the initial condition is automatically satisfied. In this case, the Loss Function becomes

$$\textbf{(2.60)} \quad L^2(\overline{y}, \overline{y}_t) = \int_0^T \left[\overline{y}_t - \overline{y}(1-\overline{y})\right]^2 dt,$$

where

$$\textbf{(2.61)} \quad \overline{y}_t(t) = u(t; \boldsymbol{\omega}, \boldsymbol{\theta}) + t\, u_t(t; \boldsymbol{\omega}, \boldsymbol{\theta}).$$

Then, we can calculate the derivative

(2.62) $$\frac{\partial}{\partial z^{(k)}}\left\{\left[\bar{y}_t-\bar{y}(1-\bar{y})\right]^2\right\}dt=2\left[\bar{y}_t-\bar{y}(1-\bar{y})\right]\left[\frac{\partial \bar{y}_t}{\partial z^{(k)}}-(1-2\bar{y})\frac{\partial \bar{y}}{\partial z^{(k)}}\right],$$

where

- $\frac{\partial \bar{y}_t}{\partial z^{(k)}}=\frac{\partial u}{\partial z^{(k)}}+t\frac{\partial u_t}{\partial z^{(k)}}$;
- $\frac{\partial \bar{y}}{\partial z^{(k)}}=t\frac{\partial u}{\partial z^{(k)}}$.

We have already seen that

$$\frac{\partial u}{\partial z^{(k)}}=\frac{\partial z^{(N+1)}}{\partial X^{(N)}}\sigma'(z^{(N)})\frac{\partial z^{(N)}}{\partial X^{(N-1)}}\ldots\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'(z^{(k)}),$$

and that

$$\begin{aligned}\frac{\partial u_t}{\partial z^{(k)}}&=\frac{\partial}{\partial z^{(k)}}\frac{\partial u}{\partial t}=\frac{\partial}{\partial t}\frac{\partial u}{\partial z^{(k)}}\\&=\frac{\partial}{\partial t}\left(\frac{\partial z^{(N+1)}}{\partial X^{(N)}}\sigma'(z^{(N)})\frac{\partial z^{(N)}}{\partial X^{(N-1)}}\ldots\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'(z^{(k)})\right)\\&=\frac{\partial z^{(N+1)}}{\partial X^{(N)}}\frac{\partial \sigma'(z^{(N)})}{\partial t}\frac{\partial z^{(N)}}{\partial X^{(N-1)}}\sigma'(z^{(N-1)})\ldots\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'(z^{(k)})\\&\quad+\ldots+\\&\quad+\ldots+\\&+\frac{\partial z^{(N+1)}}{\partial X^{(N)}}\sigma'(z^{(N)})\ldots\frac{\partial z^{(s+1)}}{\partial X^{(s)}}\frac{\partial \sigma'(z^{(s)})}{\partial t}\frac{\partial z^{(s)}}{\partial X^{(s-1)}}\sigma'(z^{(s-1)})\ldots\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\sigma'(z^{(k)})\\&\quad+\ldots+\\&+\frac{\partial z^{(N+1)}}{\partial X^{(N)}}\sigma'(z^{(N)})\frac{\partial z^{(N)}}{\partial X^{(N-1)}}\sigma'(z^{(N-1)})\ldots\frac{\partial z^{(k+1)}}{\partial X^{(k)}}\frac{\partial \sigma'(z^{(k)})}{\partial t},\end{aligned}$$

where

$$\begin{aligned}\frac{\partial \sigma'(z^{(s)})}{\partial t}&=\sigma''(z^{(s)})\frac{\partial z^{(s)}}{\partial t}=\sigma''(z^{(s)})\frac{\partial z^{(s)}}{\partial X^{(s-1)}}\frac{\partial X^{(s-1)}}{\partial t}\\&=\sigma''(z^{(s)})\frac{\partial z^{(s)}}{\partial X^{(s-1)}}\sigma'(z^{(s-1)})\frac{\partial z^{(s-1)}}{\partial t}\\&=\sigma''(z^{(s)})\frac{\partial z^{(s)}}{\partial X^{(s-1)}}\sigma'(z^{(s-1)})\frac{\partial z^{(s-1)}}{\partial X^{(s-2)}}\sigma'(z^{(s-2)})\ldots\sigma'(z^{(1)})\frac{\partial z^{(1)}}{\partial t}.\end{aligned}$$

Remark

It is important to observe that we have two sets of nodes:

- t_{train} is the set of uniformly sampled nodes in $[0,T]$, which are used to train the Neural Network;

- $\boldsymbol{t}_{test}$ is the set of uniformly sampled nodes in $[0, T]$, which are used to verify, by calculating the Loss Function, that the training is working on any point in the interval $[0, T]$, and not only on the training set.

2.7.3 Matlab implementation

For the implementation in Matlab we need to discretize the domain. Let $\boldsymbol{t}_{test} = \{t_i\}_{i=1}^{M}$ be a set of uniformly sampled nodes on $[0, T]$, then the Loss Function becomes

$$\textbf{(2.63)} \quad L^2(\overline{y}, \overline{y}_t) \approx \frac{T}{M} \sum_{i=1}^{M} [\overline{y}_t(t_i) - \overline{y}(t_i)(1 - \overline{y}(t_i))]^2.$$

Let us define $I_{L^2}(t) := [\overline{y}_t(t) - \overline{y}(t)(1 - \overline{y}(t))]^2$.

Then, for each $t_j \in \boldsymbol{t}_{train}$, we can calculate the derivative

$$\frac{\partial I_{L^2}}{\partial \boldsymbol{z}^{(k)}}(t_j) = 2\left[\overline{y}_t(t_j) - \overline{y}(t_j)(1 - \overline{y}(t_j))\right]\left[\frac{\partial \overline{y}_t}{\partial \boldsymbol{z}^{(k)}}(t_j) - (1 - 2\overline{y}(t_j))\frac{\partial \overline{y}}{\partial \boldsymbol{z}^{(k)}}(t_j)\right],$$

where

- $\frac{\partial \overline{y}_t}{\partial \boldsymbol{z}^{(k)}}(t_j) = \frac{\partial u}{\partial \boldsymbol{z}^{(k)}}(t_j) + t_j \frac{\partial u_t}{\partial \boldsymbol{z}^{(k)}}(t_j) = \alpha^{(k)}(t_j) + t_j \beta_t^{(k)}(t_j);$
- $\frac{\partial \overline{y}}{\partial \boldsymbol{z}^{(k)}}(t_j) = t_j \frac{\partial u}{\partial \boldsymbol{z}^{(k)}}(t_j) = t_j \alpha^{(k)}(t_j).$

2.7.4 Matlab codes description

To implement a Neural Network in Matlab to approximate the solution of our problem, we proceeded in the following order:

- Construction of the ***function SHCNeuralNetwork***:

  This function receives as input the parameters

  $$(N_{layer},\ omega,\ theta,\ t),$$

  where
  - N_{layer} is the number of hidden layers,
  - *omega* is an array containing the weights matrices,
  - *theta* is an array containing the biases vectors,
  - t is the input node,

and calculates the output of a Neural Network. The sigmoid function was used as the activation function.

- Construction of the ***function SHCTimeDerivative***:
  This function receives as input the parameters
  $$(N_{layer}, omega, theta, t),$$
  where
  - N_{layer} is the number of hidden layers,
  - $omega$ is an array containing the weights matrices,
  - $theta$ is an array containing the biases vectors,
  - t is the input node,

  and calculates, through the automatic differentiation process described in Section 2.5.2, the functional evaluation of the derivative, with respect to *t,* of the Neural Network. Of course, having used the sigmoid function as the activation function, it was taken into account that
  $$\sigma'(\boldsymbol{x}) = \sigma(\boldsymbol{x}) \odot (1 - \sigma(\boldsymbol{x})).$$

- Construction of the ***function SHCLossFunction***:
  This function receives as input the parameters
  $$(N_{layer}, omega, theta, t, T, y_0),$$
  where
  - N_{layer} is the number of hidden layers,
  - $omega$ is an array containing the weights matrices,
  - $theta$ is an array containing the biases vectors,
  - t is the input node,
  - T is the right boundary of the interval $[0, T]$,
  - y_0 is the initial condition,

  and returns as output the value of the Loss Function.

- Construction of the ***function SHCWeightsUpdate***:
  This function receives as input the parameters
  $$(N_{layer}, omega, theta, t_{test}, T, y_0, lrate)$$

where

- N_{layer} is the number of hidden layers,
- *omega* is an array containing the weights matrices,
- *theta* is an array containing the biases vectors,
- t_{test} is the test set of input nodes,
- T is the right boundary of the interval $[0, T]$,
- y_0 is the initial condition,
- *lrate* is the learning rate,

and updates the weights of the Neural Network through the process described above.

Remark

Let us observe that we have two sets of points:

- t_{test}, which is given as input from outside;
- t_{train}, which is randomly generated at each step.

The code is structured as follows:

- There is a while loop that repeats until the error generated is small enough (or if a certain threshold of iterations is exceeded);
- at each step of the while loop:
  - a new training set t_{train} is generated. This is to ensure that the Neural Network always learns from different points;
  - a for loop starts on all the points in the training set. For each point,

$$2\left[\overline{y}_t(t_j)-\overline{y}(t_j)(1-\overline{y}(t_j))\right]\left[\frac{\partial \overline{y}_t}{\partial \boldsymbol{z}^{(k)}}(t_j)-(1-2\overline{y}(t_j))\frac{\partial \overline{y}}{\partial \boldsymbol{z}^{(k)}}(t_j)\right],$$

    is calculated and the weights are updated;
  - at the end of the for loop a check is made on the error generated, i.e., the Loss Function L^2 is calculated on the test set t_{test}.
- If the error generated is small enough, the while loop stops and outputs the value of the Loss Function and the new values of omega and theta.

Once the structure for the operation of the learning process has been built

- ***function SHCNeuralNetwork,***
- ***function SHCTimeDerivative,***
- ***function SHCLossFunction,***
- ***function SHCWeightsUpdate,***

we tried to apply it in the script ***SHCtest***.

At first, having no particular criteria for choosing the parameters and weights, we have chosen them totally randomly. Unfortunately, by comparing the graph of the exact solution and the graph of the solution approximated by the Neural Network with each other, we noticed a big difference between them, see Figure 2.13. This discrepancy, is due to the fact that the output of the Neural Network is strongly influenced by the choice of the parameters and weights. To solve this drawback, we built

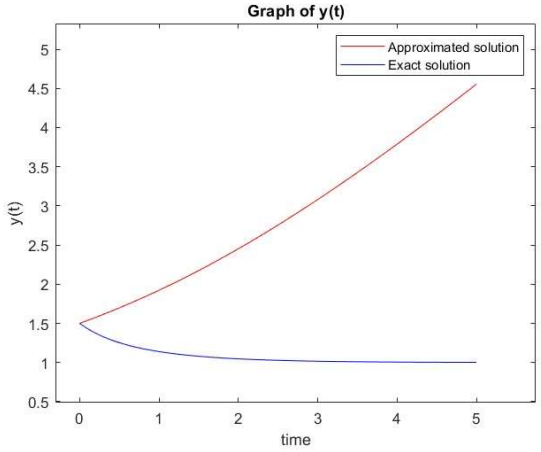

Figure 2.13: comparison of the exact solution and the wrongly approximated solution

- ***SHCParametersOptimization***:

  This script, runs for loops by varying the number of hidden layers, the dimension of the hidden layers, and the learning rate, and outputs back the "best" parameters so that, after applying the ***function SHCWeightsUpdate***, the error produced by the Loss Function is as low as possible. Using this script, it was observed that a good choice for parameters is as follows:

PARAMETERS	
Number of hidden layers (N_layer)	1
Dimension of any hidden layer (dimension)	3
Learning rate (lrate)	0.001

- ***script SHCWeightsOptimization***:

  This script uses the parameters found through the script ***SHCParameterOptimization*** and generates new matrices of weights and biases so that, once the function ***SHCWeightsUpdate*** is applied, the error generated by the Loss Function is as low as possible.

As it was done in [12], to generate the weights and bias matrices, we used the

Xavier initialization method (or Glorot initialization)

It is a technique used to initialize Neural Network weights in order to improve model convergence during training. This technique is particularly useful and it is used to reduce the problems of vanishing/exploding gradients. The main idea is to keep the variance of activation values constant across the various layers of the network, preventing the outputs from becoming too small or too large. This is achieved by choosing the initial weights through a distribution with a specified variance.

For a layer with m input nodes and n output nodes, the weights $\boldsymbol{W}=\{\boldsymbol{\omega},\boldsymbol{\theta}\}$ are initialized using:

- uniform distribution, i.e.

$$\textbf{(2.64)}\quad \boldsymbol{W}\sim U\left(-\sqrt{\frac{6}{m+n}},+\sqrt{\frac{6}{m+n}}\right);$$

- normal (Gaussian) distribution, i.e.

$$\textbf{(2.65)}\quad \boldsymbol{W}\sim N\left(0,+\sqrt{\frac{2}{m+n}}\right).$$

Once ***SHCParametersOptimization*** and ***SHCWeightsOptimization*** were implemented, we have implemented the ***SHCTest2*** script using those obtained from ***SHCParametersOptimization*** as parameters and those obtained from ***SHCWeightsOptimization*** as weights and bias. We thus obtained a good

approximation of the solution, as we can see in Figure 2.14. In the interval $[0, T]$ we have fixed $n_{test}=50$ nodes for error calculation.

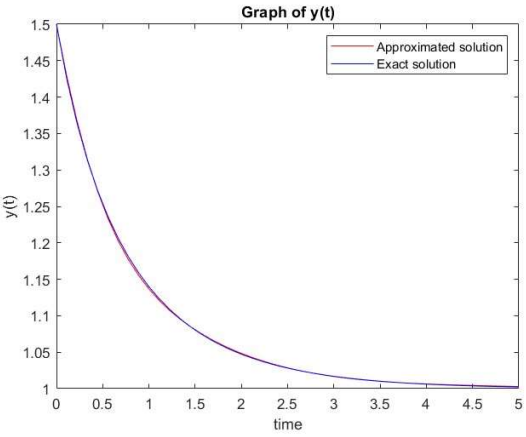

Figure 2.14:comparison of the exact solution and the approximated solution

By using the Loss Function, it was calculated that

$$Error = L^2(\overline{y}, \overline{y}_t) \approx \frac{T}{n_{test}} \sum_{i=1}^{n_{test}} [\overline{y}_t(t_i) - \overline{y}(t_i)(1-\overline{y}(t_i))]^2 = 7.5932 \times 10^{-4},$$

then the solution found "minimizes" the Loss Function.
In addition, however, we wanted to calculate the average distance between the approximate and the real solution, obtaining

$$QuadraticError = \frac{1}{n_{test}} \sum_{i=1}^{n_{test}} |s(t_i) - u(t_i)|^2 = 0.00022.$$

For more information about the Matlab codes, see Appendix B.

2.7.5 General case with spatially homogeneous initial conditions

Let us now consider the dimensionless Fisher-Kolmogorov-Petrovsky-Piskunov problem with boundary conditions

$$\textbf{(2.66)} \quad \begin{cases} y_t - y_{xx} = y(1-y), \ \ in \ \ (a,b)\times(0,T], \\ y(x,0) = y_0, \ \ in \ \ [a,b]\times\{t=0\}, \\ y(a,t) = y_a(t), \ y(b,t) = y_b(t), \ \ t\in[0,T], \end{cases}$$

where, for example, we set $a=-3$, $b=3$, $T=5$.

We observe that $y_a(0) = y_b(0) = y_0$, so we choose, as an initial test, for simplicity, the boundary conditions

$$y_a(t) = y_b(t) = y_0 \frac{e^t}{1 + y_0(e^t - 1)},$$

which represents the exact solution in the spatially homogeneous case.

Now, let us consider the trial solution

$$\textbf{(2.67)}\quad \overline{y}(x,t)=y_a(t)\frac{b-x}{b-a}+y_b(t)\frac{x-a}{b-a}+t\frac{(b-x)(x-a)}{(b-a)^2}u(x,t;\boldsymbol{\omega},\boldsymbol{\theta}),$$

and we observe that the initial condition and the boundary conditions are automatically satisfied, in fact:

- $\overline{y}(x,0)=y_a(0)\frac{b-x}{b-a}+y_b(0)\frac{x-a}{b-a}=y_0\left[\frac{b-x}{b-a}+\frac{x-a}{b-a}\right]=y_0;$
- $\overline{y}(a,t)=y_a(t);$
- $\overline{y}(b,t)=y_b(t).$

In this case, the Loss Function becomes

$$\textbf{(2.68)}\quad L^2(\overline{y},\overline{y}_t,\overline{y}_{xx})=\int_0^T dt\int_\Omega\left[\overline{y}_t-\overline{y}_{xx}-\overline{y}(1-\overline{y})\right]^2 d\boldsymbol{x},$$

where

- $\overline{y}_t=y_a'(t)\frac{b-x}{b-a}+y_b'(t)\frac{x-a}{b-a}+\frac{(b-x)(x-a)}{(b-a)^2}u+t\frac{(b-x)(x-a)}{(b-a)^2}u_t;$
- $\overline{y}_x=\frac{-y_a(t)}{b-a}+\frac{y_b(t)}{b-a}+t\frac{-(x-a)+(b-x)}{(b-a)^2}u+t\frac{(b-x)(x-a)}{(b-a)^2}u_x$
  $=\frac{y_b(t)-y_a(t)}{b-a}+t\frac{-2x+b+a}{(b-a)^2}u+t\frac{(b-x)(x-a)}{(b-a)^2}u_x;$
- $\overline{y}_{xx}=-t\frac{2}{(b-a)^2}u+t\frac{-2x+b+a}{(b-a)^2}u_x+t\frac{-2x+b+a}{(b-a)^2}u_x+t\frac{(b-x)(x-a)}{(b-a)^2}u_{xx}$
  $=-t\frac{2}{(b-a)^2}u+2t\frac{-2x+b+a}{(b-a)^2}u_x+t\frac{(b-x)(x-a)}{(b-a)^2}u_{xx}.$

Let us define $I_{L^2}(x,t):=[\overline{y}_t(x,t)-\overline{y}(x,t)(1-\overline{y}(x,t))]^2$. Then, we can calculate the derivative

$$\frac{\partial I_{L^2}}{\partial z^{(k)}}=\frac{\partial}{\partial z^{(k)}}\left\{\left[\overline{y}_t-\overline{y}_{xx}-\overline{y}(1-\overline{y})\right]^2\right\}=2\left[\overline{y}_t-\overline{y}_{xx}-\overline{y}(1-\overline{y})\right]\left[\frac{\partial\overline{y}_t}{\partial z^{(k)}}-\frac{\partial\overline{y}_{xx}}{\partial z^{(k)}}-(1-2\overline{y})\frac{\partial\overline{y}}{\partial z^{(k)}}\right],$$

where

- $\frac{\partial\overline{y}}{\partial z^{(k)}}=t\frac{(b-x)(x-a)}{(b-a)^2}\frac{\partial u}{\partial z^{(k)}};$
- $\frac{\partial\overline{y}_t}{\partial z^{(k)}}=\frac{(b-x)(x-a)}{(b-a)^2}\frac{\partial u}{\partial z^{(k)}}+t\frac{(b-x)(x-a)}{(b-a)^2}\frac{\partial u_t}{\partial z^{(k)}};$

- $\frac{\partial \overline{y}_{xx}}{\partial z^{(k)}} = -t\frac{2}{(b-a)^2}\frac{\partial u}{\partial z^{(k)}} + 2t\frac{-2x+b+a}{(b-a)^2}\frac{\partial u_x}{\partial z^{(k)}} + t\frac{(b-x)(x-a)}{(b-a)^2}\frac{\partial u_{xx}}{\partial z^{(k)}}.$

Reminder

Remember that $\frac{\partial u}{\partial z^{(k)}}$, $\frac{\partial u_t}{\partial z^{(k)}}$, $\frac{\partial u_{xx}}{\partial z^{(k)}}$, were calculated using the formulas in section 2.6.3.

2.7.6 Matlab implementation

For the implementation in Matlab we need to discretize the domain. Let $\boldsymbol{t} = \{t_i\}_{i=1}^{M}$ be the set of uniformly sampled points on $[0, T]$, and let $\boldsymbol{x} = \{x_j\}_{j=1}^{N}$ be the set of uniformly sampled points on $[a, b]$. Then the Loss Function becomes

(2.69) $$L^2(\overline{y}, \overline{y}_t, \overline{y}_{xx}) \approx \frac{T(b-a)}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}[\overline{y}_t(x_j,t_i) - \overline{y}_{xx}(x_j,t_i) - \overline{y}(x_j,t_i)(1-\overline{y}(x_j,t_i))]^2.$$

Then, for each (x_j, t_i) in the training set, we can calculate the derivative

$$\frac{\partial I_{L^2}}{\partial z^{(k)}}(x_j,t_i) = 2[\overline{y}_t(x_j,t_i) - \overline{y}_{xx}(x_j,t_i) - \overline{y}(x_j,t_i)(1-\overline{y}(x_j,t_i))] \times \left[\frac{\partial \overline{y}_t}{\partial z^{(k)}}(x_j,t_i) - \frac{\partial \overline{y}_{xx}}{\partial z^{(k)}}(x_j,t_i) - (1-2\overline{y}(x_j,t_i))\frac{\partial \overline{y}}{\partial z^{(k)}}(x_j,t_i)\right],$$

where

- $\frac{\partial \overline{y}}{\partial z^{(k)}}(x_j,t_i) = t_i\frac{(b-x_j)(x_j-a)}{(b-a)^2}\alpha^{(k)}(x_j,t_i);$
- $\frac{\partial \overline{y}_t}{\partial z^{(k)}}(x_j,t_i) = \frac{(b-x_j)(x_j-a)}{(b-a)^2}\alpha^{(k)}(x_j,t_i) + t_i\frac{(b-x_j)(x_j-a)}{(b-a)^2}\beta_t^{(k)}(x_j,t_i);$
- $\frac{\partial \overline{y}_{xx}}{\partial z^{(k)}}(x_j,t_i) = -t_i\frac{2}{(b-a)^2}\alpha^{(k)}(x_j,t_i) + 2t_i\frac{-2x_j+b+a}{(b-a)^2}\beta_x^{(k)}(x_j,t_i) + t_i\frac{(b-x_j)(x_j-a)}{(b-a)^2}\gamma^{(k)}(x_j,t_i).$

2.7.7 Matlab codes description

To implement a Neural Network in Matlab to approximate the solution of our problem, we proceeded in the following order:

- Construction of the ***function NeuralNetwork***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_0)$$

  where

  - N_{layer} is the number of hidden layers,
  - *omega* is an array containing the weights matrices,
  - *theta* is an array containing the biases vectors,
  - X_0 is the input node,

  and calculate the output of a Neural Network. The sigmoid function was used as the activation function.

- Construction of the ***function TimeDerivative***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_0)$$

  where

  - N_{layer} is the number of hidden layers,
  - *omega* is an array containing the weights matrices,
  - *theta* is an array containing the biases vectors,
  - X_0 is the input node,

  and calculates, through the automatic differentiation process described in Section 2.5.2, the functional evaluation of the derivative, with respect to *t,* of the Neural Network. Of course, having used the sigmoid function as the activation function, it was taken into account that

  $$\sigma'(\boldsymbol{x}) = \sigma(\boldsymbol{x}) \odot (1 - \sigma(\boldsymbol{x})).$$

- Construction of the ***function SpaceDerivative***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_0)$$

  where

  - N_{layer} is the number of hidden layers,
  - *omega* is an array containing the weights matrices,
  - *theta* is an array containing the biases vectors,

- X_0 is the input node,

and calculates, through the automatic differentiation process described in Section 2.5.2, the functional evaluation of the derivative, with respect to x, of the Neural Network.

- Construction of the ***function Laplacian***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_0)$$

  where

  - N_{layer} is the number of hidden layers,
  - $omega$ is an array containing the weights matrices,
  - $theta$ is an array containing the biases vectors,
  - X_0 is the input node,

  and calculates, through the automatic differentiation process described in Section 2.5.2, the functional evaluation of the laplacian of the Neural Networks. Of course, having used the sigmoid function as the activation function, it was taken into account that

  $$\sigma''(z^{(k)}) = \sigma(z^{(k)}) \odot (1 - \sigma(z^{(k)})) \odot (1 - 2\sigma(z^{(k)})).$$

- Construction of the ***function LossFunction***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_{test}, T, a, b, ya, yb, dya, dyb),$$

  where

  - N_{layer} is the number of hidden layers,
  - $omega$ is an array containing the weights matrices,
  - $theta$ is an array containing the biases vectors,
  - X_{test} is the test set,
  - T is the right boundary of the interval $[0, T]$,
  - a is the left boundary of the interval $[a, b]$,
  - b is the right boundary of the interval $[a, b]$,
  - $ya \ and \ yb$ are the boundary conditions,
  - $dya \ and \ dyb$ are the derivatives, respect to t, of $ya \ and \ yb$,

and returns as output the value of the Loss Function, evaluated on the test set.

- Construction of the ***function WeightsUpdate***:
  This function receives as input the parameters
  $$(N_{layer}, omega, theta, X_{test}, ya, yb, dya, dyb, a, b, T, lrate),$$
  where
  - N_{layer} is the number of hidden layers,
  - $omega$ is an array containing the weights matrices,
  - $theta$ is an array containing the biases vectors,
  - X_{test} are uniformly distributed grid point on $[a, b]\times[0, T]$, which are used to test the error of the Neural Network,
  - $ya \ and \ yb$ are the boundary conditions,
  - $dya \ and \ dyb$ are the derivatives, respect to t, of $ya \ and \ yb$,
  - a is the left boundary of the interval $[a, b]$,
  - b is the right boundary of the interval $[a, b]$,
  - T is the right boundary of the interval $[0, T]$,
  - $lrate$ represents the learning rate

  and updates the weights of the Neural Network using the Gradient Method.

Remark

Let us observe that we have two sets of points:

- X_{test}, which is given as input from outside;
- X_{train}, which is randomly generated at each step.

The code is structured as follows:

- There is a while loop that repeats until the error generated is small enough (or if a certain threshold of iterations is exceeded);
- at each step of the while loop:
  - a new training set X_{train} is generated. This is to ensure that the Neural Network always learns from different points;
  - a for loop starts on all the points in the training set. For each point,

$$2[\overline{y}_t(x_j,t_i)-\overline{y}_{xx}(x_j,t_i)-\overline{y}(x_j,t_i)(1-\overline{y}(x_j,t_i))]$$
$$\times\left[\frac{\partial\overline{y}_t}{\partial z^{(k)}}(x_j,t_i)-\frac{\partial\overline{y}_{xx}}{\partial z^{(k)}}(x_j,t_i)-(1-2\overline{y}(x_j,t_i))\frac{\partial\overline{y}}{\partial z^{(k)}}(x_j,t_i)\right],$$

is calculated and the weights are updated;

- at the end of the for loop a check is made on the error generated, i.e., the Loss Function L^2 is calculated on the test set X_{test}.

- If the error generated is small enough, the while loop stops and outputs the value of the Loss Function and the new values of omega and theta.

Once the structure for the operation of the learning process has been built

- ***function NeuralNetwork,***
- ***function TimeDerivative,***
- ***function SpaceDerivative,***
- ***function Laplacian,***
- ***function LossFunction,***
- ***function WeightsUpdate,***

we tried to apply it in the script ***test***. After a few attempts, it was observed that a good choice for parameters is as follows:

PARAMETERS	
Number of hidden layers (N_layer)	1
Dimension of any hidden layer (dimension)	2
Learning rate (lrate)	0.001

To generate the weights and bias matrices, we have used, as previously, the *Xavier initialization method (or Glorot initialization)*. We thus obtained a good approximation of the solution, as we can see in Figure 2.15. For the calculation of the error, we have fixed $t_{test}=50$ time nodes in the interval $[0,T]$ and $x_{test}=20$ space nodes in the interval $[a,b]$.

By using the Loss Function, it was calculated that

$$L^2\approx\frac{T(b-a)}{t_{test}\times x_{test}}\sum_{i=1}^{t_{test}}\sum_{j=1}^{x_{test}}[\overline{y}_t(x_j,t_i)-\overline{y}_{xx}(x_j,t_i)-\overline{y}(x_j,t_i)(1-\overline{y}(x_j,t_i))]^2=3.5792\times10^{-4},$$

then the solution found “minimizes” the Loss Function.

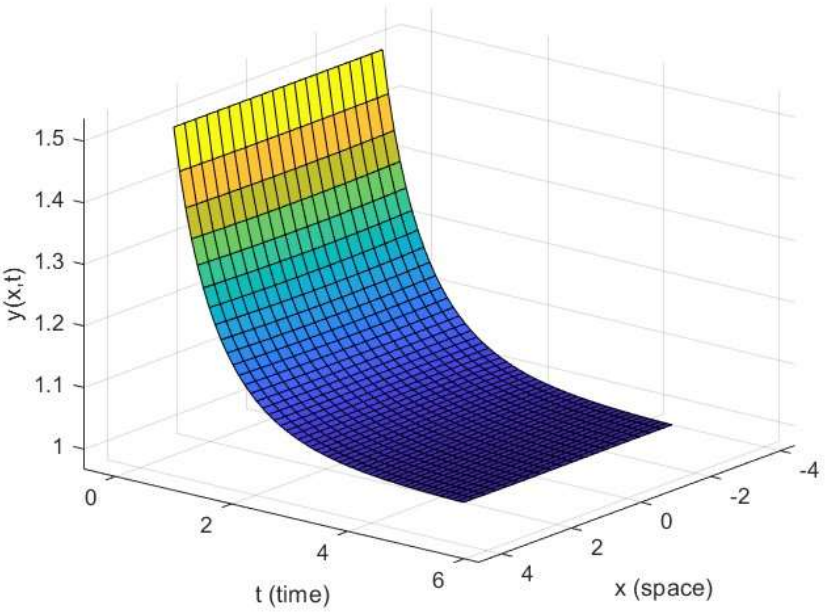

Figure 2.15: Approximated solution of the dimensionless FKPP problem with spatially homogeneous initial conditions

In addition, we can observe that the boundary conditions turn out to be actually verified, in fact, as we can see in Figure 2.16, the graphs are "completely" overlaid.

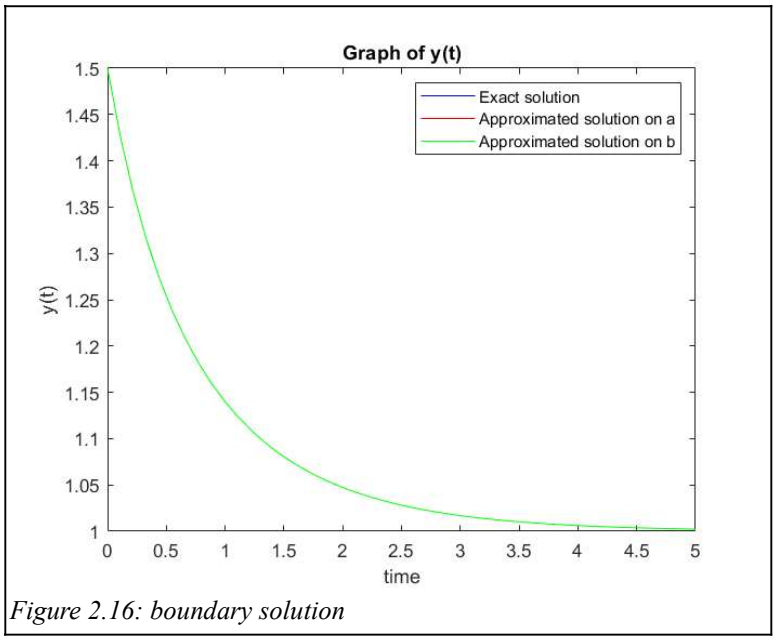

Figure 2.16: boundary solution

For more information about Matlab codes, see Appendix C.

2.7.8 Case with non-homogeneous initial conditions

Let us now consider the dimensionless Fisher-Kolmogorov-Petrovsky-Piskunov problem with boundary conditions

$$\begin{cases} y_t - y_{xx} = y(1-y), \ \ in \ \ (a,b)\times(0,T], \\ y(x,0) = y_0(x), \ \ in \ \ [a,b]\times\{t=0\}, \\ y(a,t) = y_a(t), y(b,t) = y_b(t), \ \ t\in[0,T], \end{cases}$$

where $y_0(x)\in C([a,b])$ and $y_a(t),\ y_b(t)\in C([0,T])$. Let us observe that, in order to have continuity on the boundary, one must have

$$\begin{cases} y_a(0) = y_0(a), \\ y_b(0) = y_0(b). \end{cases}$$

Now, let us consider the trial solution

$$\overline{y}(x,t) = \frac{b-x}{b-a} y_a(t) + \frac{x-a}{b-a} y_b(t) + \frac{(b-x)(x-a)}{(b-a)^2} u(x,t;\boldsymbol{\omega},\boldsymbol{\theta}),$$

and we observe that the boundary condition are automatically satisfied, in fact:

- $\overline{y}(a,t) = y_a(t)$;
- $\overline{y}(b,t) = y_b(t)$.

The initial condition and the reaction-diffusion equation remain to be satisfied. In this case the Loss Function becomes

$$L^2(\overline{y}, \overline{y}_t, \overline{y}_{xx}) = \int_0^T dt \int_a^b \left[\overline{y}_t - \overline{y}_{xx} - \overline{y}(1-\overline{y})\right]^2 dx + \int_a^b \left[\overline{y}(x,0) - y_0(x)\right]^2 dx,$$

where

- $\overline{y}_t = \frac{b-x}{b-a} y_a'(t) + \frac{x-a}{b-a} y_b'(t) + \frac{(b-x)(x-a)}{(b-a)^2} u_t$;
- $\overline{y}_x = \frac{-y_a(t)}{b-a} + \frac{y_b(t)}{b-a} + \frac{-2x+b+a}{(b-a)^2} u + \frac{(b-x)(x-a)}{(b-a)^2} u_x$;
- $\overline{y}_{xx} = \frac{-2}{(b-a)^2} u + 2\frac{-2x+b+a}{(b-a)^2} u_x + \frac{(b-x)(x-a)}{(b-a)^2} u_{xx}$.

Let us observe it is required that $y_a(t),\ y_b(t)\in C^1([0,T])$.

Now, we can calculate the two derivatives

$$\frac{\partial}{\partial z^{(k)}}\left\{\left[\overline{y}_t - \overline{y}_{xx} - \overline{y}(1-\overline{y})\right]^2\right\} = 2\left[\overline{y}_t - \overline{y}_{xx} - \overline{y}(1-\overline{y})\right]\left[\frac{\partial \overline{y}_t}{\partial z^{(k)}} - \frac{\partial \overline{y}_{xx}}{\partial z^{(k)}} - (1-2\overline{y})\frac{\partial \overline{y}}{\partial z^{(k)}}\right],$$

$$\frac{\partial}{\partial \boldsymbol{z}^{(k)}}\{[\bar{y}(x,0)-y_0(x)]^2\}=2[\bar{y}(x,0)-y_0(x)]\frac{\partial \bar{y}}{\partial \boldsymbol{z}^{(k)}}(x,0),$$

where

- $\frac{\partial \bar{y}}{\partial \boldsymbol{z}^{(k)}}=\frac{(b-x)(x-a)}{(b-a)^2}\frac{\partial u}{\partial \boldsymbol{z}^{(k)}}$;
- $\frac{\partial \bar{y}_t}{\partial \boldsymbol{z}^{(k)}}=\frac{(b-x)(x-a)}{(b-a)^2}\frac{\partial u}{\partial \boldsymbol{z}^{(k)}}+\frac{(b-x)(x-a)}{(b-a)^2}\frac{\partial u_t}{\partial \boldsymbol{z}^{(k)}}$;
- $\frac{\partial \bar{y}_{xx}}{\partial \boldsymbol{z}^{(k)}}=-\frac{2}{(b-a)^2}\frac{\partial u}{\partial \boldsymbol{z}^{(k)}}+2\frac{-2x+b+a}{(b-a)^2}\frac{\partial u_x}{\partial \boldsymbol{z}^{(k)}}+\frac{(b-x)(x-a)}{(b-a)^2}\frac{\partial u_{xx}}{\partial \boldsymbol{z}^{(k)}}$.

Reminder

Remember that $\frac{\partial u}{\partial \boldsymbol{z}^{(k)}}$, $\frac{\partial u_t}{\partial \boldsymbol{z}^{(k)}}$, $\frac{\partial u_{xx}}{\partial \boldsymbol{z}^{(k)}}$, were calculated using the formulas in section 2.6.3.

2.7.9 Matlab implementation

For the implementation in Matlab we need to discretize the domain. Let $\boldsymbol{t}=\{t_i\}_{i=1}^M$ be the set of uniformly sampled points on $[0,T]$, and let $\boldsymbol{x}=\{x_j\}_{j=1}^N$ be the set of uniformly sampled points on $[a,b]$. Then the Loss Function becomes

$$\begin{aligned}L^2(\bar{y},\bar{y}_t,\bar{y}_{xx})\approx&\frac{T(b-a)}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}[\bar{y}_t(x_j,t_i)-\bar{y}_{xx}(x_j,t_i)-\bar{y}(x_j,t_i)(1-\bar{y}(x_j,t_i))]^2\\&+\frac{(b-a)}{N}\sum_{j=1}^{N}[\bar{y}(x_j,0)-y_0(x_j)]^2,\end{aligned}\tag{2.69}$$

Then, for each (x_j,t_i) in the training set, we can calculate the two derivatives

$$\begin{aligned}&2[\bar{y}_t(x_j,t_i)-\bar{y}_{xx}(x_j,t_i)-\bar{y}(x_j,t_i)(1-\bar{y}(x_j,t_i))]\\&\quad\times\left[\frac{\partial \bar{y}_t}{\partial \boldsymbol{z}^{(k)}}(x_j,t_i)-\frac{\partial \bar{y}_{xx}}{\partial \boldsymbol{z}^{(k)}}(x_j,t_i)-(1-2\bar{y}(x_j,t_i))\frac{\partial \bar{y}}{\partial \boldsymbol{z}^{(k)}}(x_j,t_i)\right],\end{aligned}$$

$$2[\bar{y}(x_j,0)-y_0(x_j)]\frac{\partial \bar{y}}{\partial \boldsymbol{z}^{(k)}}(x_j,0),$$

where

- $\frac{\partial \bar{y}}{\partial \boldsymbol{z}^{(k)}}(x_j,t_i)=\frac{(b-x_j)(x_j-a)}{(b-a)^2}\alpha^{(k)}(x_j,t_i)$;
- $\frac{\partial \bar{y}_t}{\partial \boldsymbol{z}^{(k)}}(x_j,t_i)=\frac{(b-x_j)(x_j-a)}{(b-a)^2}\alpha^{(k)}(x_j,t_i)+\frac{(b-x_j)(x_j-a)}{(b-a)^2}\beta_t^{(k)}(x_j,t_i)$;

- $$\frac{\partial \overline{y}_{xx}}{\partial z^{(k)}}(x_j,t_i)=-\frac{2}{(b-a)^2}\alpha^{(k)}(x_j,t_i)+2\frac{-2x_j+b+a}{(b-a)^2}\beta_x^{(k)}(x_j,t_i)+\frac{(b-x_j)(x_j-a)}{(b-a)^2}\gamma^{(k)}(x_j,t_i).$$

For the implementation, we will set

- $[a,b]=[-3,3];$
- $[0,T]=[0,5];$
- $y_0(x)=\cos\left(\frac{\pi}{3}x\right)+2.5;$
- $y_a(t)=y_b(t)=y_0(a)\frac{e^t}{1+y_0(a)(e^t-1)}.$

2.7.10 Matlab codes description

To implement a Neural Network in Matlab to approximate the solution of our problem, we proceeded in the following order:

- Construction of the ***function GCNeuralNetwork***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_0)$$

  where

  - N_{layer} is the number of hidden layers,
  - *omega* is an array containing the weights matrices,
  - *theta* is an array containing the biases vectors,
  - X_0 is the input node,

  and calculate the output of the Neural Network. The sigmoid function was used as the activation function.

- Construction of the ***function GCTimeDerivative***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_0)$$

  where

  - N_{layer} is the number of hidden layers,
  - *omega* is an array containing the weights matrices,

- ○ *theta* is an array containing the biases vectors,
- ○ X_0 is the input node,

and calculates, through the automatic differentiation process described in Section 2.5.2, the functional evaluation of the derivative, with respect to *t,* of the Neural Network. Of course, having used the sigmoid function as the activation function, it was taken into account that

$$\sigma'(\boldsymbol{x}) = \sigma(\boldsymbol{x}) \odot (1 - \sigma(\boldsymbol{x})).$$

- Construction of the ***function GCSpaceDerivative***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_0)$$

  where

  - ○ N_{layer} is the number of hidden layers,
  - ○ *omega* is an array containing the weights matrices,
  - ○ *theta* is an array containing the biases vectors,
  - ○ X_0 is the input node,

  and calculates, through the automatic differentiation process described in Section 2.5.2, the functional evaluation of the derivative, with respect to x, of the Neural Network.

- Construction of the ***function GCLaplacian***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_0)$$

  where

  - ○ N_{layer} is the number of hidden layers,
  - ○ *omega* is an array containing the weights matrices,
  - ○ *theta* is an array containing the biases vectors,
  - ○ X_0 is the input node,

  and calculates, through the automatic differentiation process described in Section 2.5.2, the functional evaluation of the laplacian of the Neural Networks. Of course, having used the sigmoid function as the activation function, it was taken into account that

$$\sigma''(z^{(k)}) = \sigma(z^{(k)}) \odot (1 - \sigma(z^{(k)})) \odot (1 - 2\sigma(z^{(k)})).$$

- Construction of the ***function GCLossFunction***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_{test}, X_{test\,0}, T, a, b, y_0, ya, dya),$$

  where

  - N_{layer} is the number of hidden layers,
  - $omega$ is an array containing the weights matrices,
  - $theta$ is an array containing the biases vectors,
  - X_{test} is the test set on $[a, b] \times (0, T]$,
  - $X_{test\,0}$ is the test set on $[a, b] \times \{t = 0\}$,
  - T is the right boundary of the interval $[0, T]$,
  - a is the left boundary of the interval $[a, b]$,
  - b is the right boundary of the interval $[a, b]$,
  - y_0 is the initial condition,
  - ya is the boundary condition (remember that $ya = yb$),
  - dya is the derivative, respect to t, of ya (remember that $dya = dyb$),

  and returns as output the value of the Loss Function, evaluated at the test sets X_{test} and $X_{test\,0}$.

- Construction of the ***function GCWeightsUpdate***:

  This function receives as input the parameters

  $$(N_{layer}, omega, theta, X_{test}, X_{test\,0}, y_0, ya, dya, a, b, T, lrate),$$

  where

  - N_{layer} is the number of hidden layers,
  - $omega$ is an array containing the weights matrices,
  - $theta$ is an array containing the biases vectors,
  - X_{test} is the test set on $[a, b] \times (0, T]$,
  - $X_{test\,0}$ is the test set on $[a, b] \times \{t = 0\}$,
  - T is the right boundary of the interval $[0, T]$,
  - a is the left boundary of the interval $[a, b]$,
  - b is the right boundary of the interval $[a, b]$,

- y_0 is the initial condition,
- ya is the boundary condition (remembere that $ya = yb$),
- dya is the derivative, respect to t, of ya (remembere that $dya = dyb$),
- $lrate$ represents the learning rate

and updates the weights of the Neural Network using the Gradient Method.

Remark

Let us observe that we have four sets of points:

- $X_{test}, X_{test\,0}$ which are given as input from outside;
- $X_{train}, X_{train\,0}$ which are randomly generated at each step, and represent, respectively, the training set on $[a, b]\times(0, T]$, and the training set on $[a, b]\times\{t = 0\}$.

The code is structured as follows:

- There is a while loop that repeats until the error generated is small enough (or if a certain threshold of iterations is exceeded);
- at each step of the while loop:
  - two new training sets $X_{train}, X_{train\,0}$ are generated. This is to ensure that the Neural Network always learns from different points;
  - a for loop starts on all the points in the training sets. For each point are calculated

$$\frac{\partial L_1}{\partial z^{(k)}}(x_j, t_i) = 2\left[\bar{y}_t(x_j, t_i) - \bar{y}_{xx}(x_j, t_i) - \bar{y}(x_j, t_i)(1 - \bar{y}(x_j, t_i))\right] \times\left[\frac{\partial \bar{y}_t}{\partial z^{(k)}}(x_j, t_i) - \frac{\partial \bar{y}_{xx}}{\partial z^{(k)}}(x_j, t_i) - (1 - 2\bar{y}(x_j, t_i))\frac{\partial \bar{y}}{\partial z^{(k)}}(x_j, t_i)\right],$$

$$\frac{\partial L^2}{\partial z^{(k)}}(x_j, 0) = 2\left[\bar{y}(x_j, 0) - y_0(x_j)\right]\frac{\partial \bar{y}}{\partial z^{(k)}}(x_j, 0),$$

  depending on whether $t \neq 0$ or $t = 0$, and the weights are updated;
  - at the end of the for loop a check is made on the error generated, i.e., the Loss Function L^2 is calculated on the test sets $X_{test}, X_{test\,0}$.
- If the error generated is small enough, the while loop stops and outputs the value of the Loss Function and the new values of omega and theta.

Once the structure for the operation of the learning process has been built

- ***function GCNeuralNetwork,***
- ***function GCTimeDerivative,***
- ***function GCSpaceDerivative,***
- ***function GCLaplacian,***
- ***function GCLossFunction,***
- ***function GCWeightsUpdate,***

we tried to apply it in the script ***test***.

After a few attempts, it was observed that a good choice for parameters is as follows:

PARAMETERS	
Number of hidden layers (N_layer)	2
Dimension of any hidden layer (dimension)	19
Learning rate (lrate)	0.01

To generate the weights and bias matrices, we have used, as previously, the *Xavier initialization method (or Glorot initialization)*. We thus obtained a good approximation of the solution, as we can see in Figure 2.17. For the calculation of the error, we have fixed $t_{test}=50$ time nodes in the interval $[0,T]$ and $x_{test}=x_{test\,0}=20$ space nodes in the interval $[a,b]$. By using the Loss Function, it was calculated that

$$L^2(\overline{y},\overline{y}_t,\overline{y}_{xx})\approx\frac{T(b-a)}{M\,N}\sum_{i=1}^{M}\sum_{j=1}^{N}[\overline{y}_t(x_j,t_i)-\overline{y}_{xx}(x_j,t_i)-\overline{y}(x_j,t_i)(1-\overline{y}(x_j,t_i))]^2$$
$$+\frac{(b-a)}{N}\sum_{j=1}^{N}[\overline{y}(x_j,0)-y_0(x_j)]^2=6.4376\times10^{-3},$$

then the solution found “minimizes” the Loss Function.

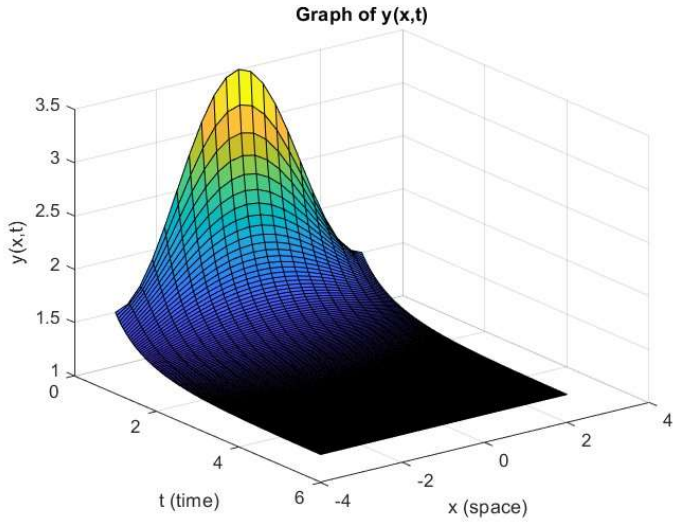

Figure 2.17:Approximated solution of the dimensionless FKPP problem with sinusoidal initial conditions

In addition, we can observe that the initial condition is also verified as we can see in Figure 2.18.

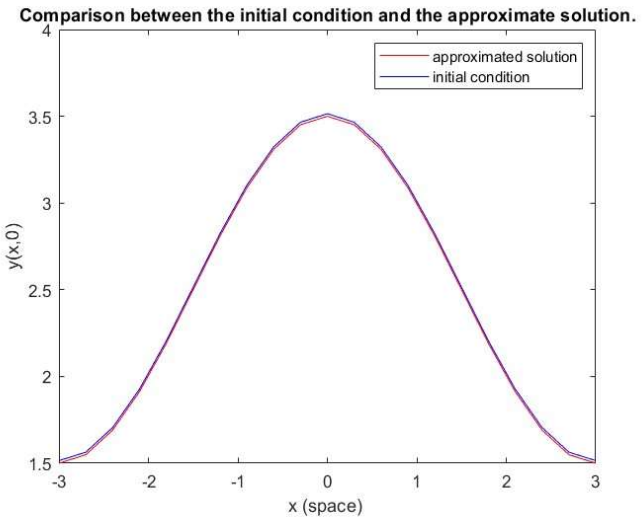

Figure 2.18: comparison between the initial condition and the approximated solution

For more information about Matlab codes, see Appendix D.

Remark

In this case, we did not obtain satisfactory results from the point of view of efficiency. In fact, after numerous unsuccessful attempts, calculating the correct weights and biases took several hours, despite using a high-end PC with an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.20 GHz and a 16 GB RAM.

Conclusions

We can say that the use of Neural Networks is undoubtedly a very powerful tool for the study of solutions of partial derivative equations and, in particular, reaction-diffusion equations. However, there are significant advantages and disadvantages to consider.

Advantages:

1. model-free learning: Neural Networks, if well trained, allow us to approximate the solution of a PDE without explicit knowledge of the solution;
2. generalization ability: Neural Networks, if well-trained, allow us to find a good approximation of the solution at any point in the domain of definition, despite having trained the Neural Network only on a limited number of nodes;
3. computational speed: once trained, Neural Networks provide us with solutions very quickly;
4. high dimension: Neural Networks allow us to handle PDEs in large spaces, whereas other numerical methods, in this case, may suffer;
5. novelty: since it is still a relatively little-used tool in this area, its potential and functionality is not known exactly. Nothing prohibits us from thinking that methods can be found to further improve its effectiveness and efficiency.

Disadvantages:

1. need for training data: Neural Networks, in order to be trained, require large amounts of data. In addition, obtaining qualitatively good data can be computationally expensive;
2. training time: training Neural Networks can take a long time, especially in the case of very deep and complex Networks;
3. optimization: Neural Networks are highly dependent on the initial choice of parameters (weights and bias). This leads to the implementation of

additional methods to find the best initial parameters so that the Neural Network can be trained properly;

4. overfitting: Neural Networks easily go into overfitting, i.e., they overfit the training data, as a result they have poor generalization ability, especially if small training sets are used or have noise;
5. instability: the resulting solutions can sometimes be unstable and inaccurate;
6. integration: integrating Neural Networks with other existing methods can be particularly complex and requires a lot of experience;
7. novelty: being a relatively new tool, there are not yet many studies that allow us to understand how and why certain wrong solutions are produced.

It still remains open the problem of how to improve the algorithm to update the weights in the case where the Loss Function is the sum of several subfunctions that converge at the point of minimum with different velocities. In fact, in the case with non-homogeneous initial conditions, we have not obtained satisfactory results from the point of view of efficiency (the algorithm really takes too long to compute the correct weights, always if it finds them, because it may “explode”).
We set as a future goal to continue the research in this field, trying to obtain further results, not only from a purely theoretical point of view, but also, and especially, from a practical point of view. In fact, we will focus on extending what has been done for reaction-diffusion equations, also for reaction-diffusion systems, such as, for example, the prey-predator model (the idea would be to build and train Neural Networks with two, or more, output nodes).

APPENDIX A - Function NeuralNetwork (1 of 1)

```
function X = NeuralNetwork (N_layer, omega, theta, x)

% sigmoid function
sigmoid = @(u) 1./(1+exp(-u));

% column vector array of node values
X=cell(1,N_layer+1);
X{1}=sigmoid(omega{1}*x+theta{1});

for i=2:N_layer
    % value of nodes in layer i
    X{i}=sigmoid(omega{i}*X{i-1}+theta{i});
end
% activation function is not applied to the last layer
X{N_layer+1}=omega{N_layer+1}*X{N_layer}+theta{N_layer+1};

end
```

APPENDIX A - Function SpaceDerivative (1 of 1)

```
function dx = SpaceDerivative(N_layer, omega, theta, x)

% calculation of node values
X=NeuralNetwork(N_layer, omega, theta, x);

% calculation of the time derivative
dx=cell(1,N_layer+1);
dx{1}=X{1}.*(1-X{1}).*omega{1};

for i=2:N_layer
    dx{i}=omega{i}*dx{i-1};
    dx{i}=X{i}.*(1-X{i}).*dx{i};
end

dx{N_layer+1}=omega{N_layer+1}*dx{N_layer};

end
```

APPENDIX A - Function LossFunction (1 of 1)

```
function L = LossFunction(N_layer,X_test,omega,theta,f,a,b)

    %length of the test set
    n=length(X_test);

    % calculation of u
    u=zeros(1,n);
    for i=1:n
        x=X_test(i);
        X=NeuralNetwork(N_layer, omega, theta, x);
        u(i)=X{N_layer+1};
    end

    % calculation of the Loss Function
    L=0;
    for i=1:n
        x=X_test(i);
        L=L+(u(i)-f(x))^2;
    end
    L=L*(b-a)/n;

end
```

APPENDIX A - Function WeightsUpdate (1 of 2)

```
function [error,thetanew,omeganew] = WeightsUpdate(N_layer,omega,theta,...
                                                    X_test,a,b,f,lrate)

% parameters initialization
error=10;
errormax=0.0005;
threshold=1;
thresholdmax=20000;
W=cell(1,N_layer+1);
thetanew=theta;
omeganew=omega;

% weights update
while error>errormax && threshold<thresholdmax
    n_train=20;
    X_train=a+(b-a)*rand(1,n_train);

    % calculation of u
    u=zeros(1,n_train);
    for i=1:n_train
        x=X_train(i);
        X=NeuralNetwork(N_layer, omega, theta, x);
        u(i)=X{N_layer+1};
    end

    % initialization of alfa
    alfa=cell(n_train,N_layer+1);

    % weights update
    for i=1:n_train
        % weights update of the layer N+1
        x=X_train(i);
        X=NeuralNetwork(N_layer,omega,theta,x);
        alfa{i,N_layer+1}=1;
        dL=2*(u(i)-f(x))*[1;X{N_layer}]';
        W{N_layer+1}=[theta{N_layer+1},omega{N_layer+1}];
        W{N_layer+1}=W{N_layer+1}-lrate*dL;
        theta{N_layer+1}=W{N_layer+1}(:,1);
        omega{N_layer+1}=W{N_layer+1}(:,2:end);

        for k=N_layer:-1:2
            % weights update of the layer k
            x=X_train(i);
            X=NeuralNetwork(N_layer, omega, theta, x);
            alfa{i,k}=alfa{i,k+1}.*(omega{k+1}*(X{k}.*(1-X{k})));
            dL=2*(u(i)-f(x))*alfa{i,k}*[1;X{k-1}]';
            W{k}=[theta{k},omega{k}];
            W{k}=W{k}-lrate*dL;
            theta{k}=W{k}(:,1);
            omega{k}=W{k}(:,2:end);
        end

        % weights update of the layer 1
```

```
        x=X_train(i);
        X=NeuralNetwork(N_layer,omega,theta,x);
        alfa{i,1}=alfa{i,2}.*(omega{2}*(X{1}.*(1-X{1})));
        dL=2*(u(i)-f(x))*alfa{i,1}*[1;x]';
        W{1}=[theta{1},omega{1}];
        W{1}=W{1}-lrate*dL;
        theta{1}=W{1}(:,1);
        omega{1}=W{1}(:,2:end);

    end

    % error calculation
    errornew=LossFunction(N_layer,X_test,omega,theta,f,a,b);

    % error and weights update
    if errornew<error
        error=errornew;
        omeganew=omega;
        thetanew=theta;
    end

    % Checking if the error is too big --->stop
    if errornew>100
        threshold=thresholdmax-1;
    end

    % threshold update
    threshold=threshold+1;

end

end
```

APPENDIX A - Script Test1 (1 of 2)

```
% test1

N_layer=1; % number of hidden layer
dim=5; % dimension of any hidden layer
lrate=0.01; % learning rate
f=@(x) cos(x); % initial condition

a=-pi; % left bound
b=pi;  %right bound

% construction of omega and theta
omega = cell(1, N_layer + 1);
theta = cell(1, N_layer + 1);

% Initialization of weights with Xavier
W = sqrt(2/(dim+2))*randn(dim,2);
theta{1} = W(:, 1);
omega{1} = W(:, 2:end);

for k = 2:N_layer
    W = sqrt(2/(2*dim))*randn(dim,dim+1);
    theta{k}=W(:,1);
    omega{k}=W(:,2:end);
end

W = sqrt(2/(1+dim))*randn(1,dim+1);
theta{N_layer+1}=W(:,1);
omega{N_layer+1}=W(:,2:end);

% Construction of the test set
ntest=100;
X_test=linspace(a,b, ntest);

% calculation of the error
error=LossFunction(N_layer,X_test,omega,theta,f,a,b);
disp("error= "+error);

% weights update
[errornew,thetanew,omeganew]=WeightsUpdate(N_layer,omega,theta, ...
                                                    X_test,a,b,f,lrate);
disp("errornew= "+errornew);

% saving the new error and the new weights
if errornew<error
    error=errornew;
    for i = 1:N_layer+1
        fileNameOmega = ['omega', num2str(i), '.xlsx'];
        writematrix(omeganew{i}, fileNameOmega);
        fileNameTheta = ['theta', num2str(i), '.xlsx'];
        writematrix(thetanew{i}, fileNameTheta);
    end
    omega=omeganew;
    theta=thetanew;
```

```
end

%calculation of u
u=zeros(1,ntest);
for i=1:ntest
    x=X_test(i);
    X=NeuralNetwork(N_layer, omega, theta, x);
    u(i)=X{N_layer+1};
end

%comparison with the exact solution
sol=zeros(1,ntest);
for i=1:ntest
    x=X_test(i);
    sol(i)=f(x);
end

plot(X_test, u, '-r','DisplayName', 'Approximated solution');
hold on;
plot(X_test, sol, '-b','DisplayName', 'exact solution');
title('Graph of y(t)');
legend;
```

APPENDIX A - Script Test2 (1 of 2)

```
% test2

N_layer=1; % number of hidden layer
dim=5; % dimension of any hidden layer
lrate=0.001; % learning rate
f=@(x) cos(x); % initial condition

a=-pi;
b=pi;

% construction of omega and theta
omega = cell(1, N_layer + 1);
theta = cell(1, N_layer + 1);

% Reading the weights and the  biases
for i = 1:N_layer+1
    fileNameOmega = ['omega', num2str(i), '.xlsx'];
    omega{i} = readmatrix(fileNameOmega);
    fileNameTheta = ['theta', num2str(i), '.xlsx'];
    theta{i} = readmatrix(fileNameTheta);
end

ntest=100;
X_test=linspace(a,b, ntest);

% calculation of the error
error=LossFunction(N_layer,X_test,omega,theta,f,a,b);
disp("error= "+error);

% weights update
[errornew,thetanew,omeganew]=WeightsUpdate(N_layer,omega,theta,...
                                                X_test,a,b,f,lrate);
disp("errornew= "+errornew);

% saving the new error and the new weights
if errornew<error
    error=errornew;
    for i = 1:N_layer+1
        fileNameOmega = ['omega', num2str(i), '.xlsx'];
        writematrix(omeganew{i}, fileNameOmega);
        fileNameTheta = ['theta', num2str(i), '.xlsx'];
        writematrix(thetanew{i}, fileNameTheta);
    end
    omega=omeganew;
    theta=thetanew;
end

%calculation of u
u=zeros(1,ntest);
for i=1:ntest
    x=X_test(i);
    X=NeuralNetwork(N_layer, omega, theta, x);
    u(i)=X{N_layer+1};
```

```
end

%comparison with the exact solution
sol=zeros(1,ntest);
for i=1:ntest
    x=X_test(i);
    sol(i)=f(x);
end

plot(X_test, u, '-r','DisplayName', 'Approximated solution');
hold on;
plot(X_test, sol, '-b','DisplayName', 'exact solution');
title('Graph of y(t)');
legend;
```

APPENDIX B - Function SHCNeuralNetwork (1 of 1)

```
function X = SHCNeuralNetwork (N_layer, omega, theta, t)

% sigmoid function
sigmoid = @(u) 1./(1+exp(-u));

% column vector array of node values
X=cell(1,N_layer+1);
X{1}=sigmoid(omega{1}*t+theta{1});

for i=2:N_layer
    % value of nodes in layer i
    X{i}=sigmoid(omega{i}*X{i-1}+theta{i});
end
% activation function is not applied to the last layer
X{N_layer+1}=omega{N_layer+1}*X{N_layer}+theta{N_layer+1};


end
```

APPENDIX B - Function SHCTimeDerivative (1 of 1)

```
function dt = SHCTimeDerivative(N_layer, omega, theta, t)

% calculation of node values
X=SHCNeuralNetwork(N_layer, omega, theta, t);

% calculation of the time derivative
dt=cell(1,N_layer+1);
dt{1}=X{1}.*(1-X{1}).*omega{1};

for i=2:N_layer
    dt{i}=omega{i}*dt{i-1};
    dt{i}=X{i}.*(1-X{i}).*dt{i};
end

dt{N_layer+1}=omega{N_layer+1}*dt{N_layer};

end
```

APPENDIX B - Function SHCLossFunction (1 of 1)

```
function L = SHCLossFunction(N_layer,omega,theta,t,T,y0)

% calculation of the outputs
u=cell(1, length(t));
for i=1:length(t)
    X=SHCNeuralNetwork(N_layer, omega, theta, t(i));
    u{i}=X{N_layer+1};
end

% calculation of the test function
y=cell(1, length(t));
for i=1:length(t)
    y{i}=y0+t(i)*u{i};
end

% calculation of the time derivative of u
du=cell(1,length(t));
for i=1:length(t)
    dt=SHCTimeDerivative(N_layer, omega,theta, t(i));
    du{i}=dt{N_layer+1};
end

% calculation of the time derivative of y
dy=cell(1,length(t));
for i=1:length(t)
    dy{i}=u{i}+t(i)*du{i};
end

% calculation of the loss function
L=0;
for i=1:length(t)
    L=L+(dy{i}-y{i}*(1-y{i}))^2;
end
L=T/length(t)*L;

end
```

APPENDIX B - Function SHCWeightsUpdate (1 of 3)

```
function [error,thetanew,omeganew]=SHCWeightsUpdate(N_layer,omega,theta,...
                                                    t_test,T,y0,lrate)

% parameters initialization
error=1;
errormax=0.0001;
threshold=1;
thresholdmax=100000;
W=cell(1,N_layer+1);
thetanew=theta;
omeganew=omega;

% weights update
while error>errormax && threshold<thresholdmax

    % training nodes
    n_train=10;
    t_train=T*rand(1,n_train);

    % calculation of the outputs
    u=cell(1, length(t_train));
    for i=1:length(t_train)
        X=SHCNeuralNetwork(N_layer,omega,theta,t_train(i));
        u{i}=X{N_layer+1};
    end

    % calculation of the time derivatives of u
    ut=cell(1,length(t_train));
    for i=1:length(t_train)
        dt=SHCTimeDerivative(N_layer,omega,theta,t_train(i));
        ut{i}=dt{N_layer+1};
    end

    % calculation of the test function
    y=cell(1,length(t_train));
    for i=1:length(t_train)
        y{i}=y0+t_train(i)*u{i};
    end

    % calculation of the time derivatives of u
    yt=cell(1,length(t_train));
    for i=1:length(t_train)
        yt{i}=u{i}+t_train(i)*ut{i};
    end

    % initialization of alfa and beta
    alfa=cell(length(t_train),N_layer+1);
    beta=cell(length(t_train),N_layer+1);

    for i=1:length(t_train)
        % weights update of the layer N+1
        X=SHCNeuralNetwork(N_layer,omega,theta,t_train(i));
        alfa{i,N_layer+1}=1;
```

```
    beta{i,N_layer+1}=0;
    dyt=alfa{i,N_layer+1}+t_train(i)*beta{i,N_layer+1};
    dy=t_train(i)*alfa{i,N_layer+1};
    dL=(yt{i}-y{i}.*(1-y{i})).*(dyt-(1-2*y{i}).*dy)*[1;X{N_layer}]';
    W{N_layer+1}=[theta{N_layer+1},omega{N_layer+1}];
    W{N_layer+1}=W{N_layer+1}-lrate*dL;
    theta{N_layer+1}=W{N_layer+1}(:,1);
    omega{N_layer+1}=W{N_layer+1}(:,2:end);

    for k=N_layer:-1:2
        % weights update of the layer k
        X=SHCNeuralNetwork(N_layer, omega, theta, t_train(i));
        dt=SHCTimeDerivative(N_layer, omega,theta, t_train(i));
        dsigma=X{k}.*(1-X{k}).*(1-2*X{k}).*(omega{k}*dt{k-1});
        alfa{i,k}=alfa{i,k+1}.*omega{k+1}*(X{k}.*(1-X{k}));
        beta{i,k}=beta{i,k+1}.*omega{k+1}*(X{k}.*(1-X{k}))+...
            alfa{i,k+1}.*omega{k+1}*(dsigma);
        dyt=alfa{i,k}+t_train(i)*beta{i,k};
        dy=t_train(i)*alfa{i,k};
        dL=(yt{i}-y{i}*(1-y{i}))*(dyt-(1-2*y{i})*dy)*[1;X{k-1}]';
        W{k}=[theta{k},omega{k}];
        W{k}=W{k}-lrate*dL;
        theta{k}=W{k}(:,1);
        omega{k}=W{k}(:,2:end);
    end

    % weights update of the layer 1
    X=SHCNeuralNetwork(N_layer,omega,theta,t_train(i));
    dsigma=X{1}.*(1-X{1}).*(1-2*X{1}).*omega{1};
    alfa{i,1}=alfa{i,2}.*omega{2}*(X{1}.*(1-X{1}));
    beta{i,1}=beta{i,2}.*omega{2}*(X{1}.*(1-X{1}))+...
        alfa{i,2}.*omega{2}*(dsigma);
    dyt=alfa{i,1}+t_train(i)*beta{i,1};
    dy=t_train(i)*alfa{i,1};
    dL=(yt{i}-y{i}*(1-y{i}))*(dyt-(1-2*y{i})*dy)*[1;t_train(i)]';
    W{1}=[theta{1},omega{1}];
    W{1}=W{1}-lrate*dL;
    theta{1}=W{1}(:,1);
    omega{1}=W{1}(:,2:end);
end

% error calculation
errornew=SHCLossFunction(N_layer,omega,theta,t_test,T,y0)

% Checking if the error is too big
if errornew>1000
    threshold=thresholdmax-1;
end

% error and weights update
if errornew<error
    error=errornew;
    thetanew=theta;
```

```
        omeganew=omega;
    end

    % threshold update
    threshold=threshold+1

end

end
```

```
% SHC Parameters Optimization

T=5; % time interval [0,T]
y0=1.5; %initial condition

% training times
n_train=21;
t_train=T*rand(1,n_train);

% testing times
n_test=46;
t_test=linspace(0, T, n_test);
error=1;

for n=1:10
    for d=2:10
        for lr=0.001:0.001:0.01

            % construction of omega and theta
            omega=cell(1, n + 1);
            theta=cell(1, n + 1);

            % Initialization of weights by Xavier's method
            W=sqrt(2/(d+2))*randn(d,2);
            theta{1}=W(:,1); % bias 1
            omega{1}=W(:,2:end); % matrice 1 (spazio e tempo)
            for k = 2:n
                W=sqrt(2/(2*d))*randn(d,d+1);
                theta{k}=W(:,1);
                omega{k}=W(:,2:end);
            end
            W=sqrt(2/(1+d))*randn(1,d+1);
            theta{n+1}=W(:,1);
            omega{n+1}=W(:,2:end);

            % weights update
            [errornew,thetanew,omeganew]=SHCWeightsUpdate(n,omega,theta, ...
                                                t_test,T,y0,lrate);

            if errornew<error
                error=errornew; %error update
                %updating the parameters
                N_layer=n;
                dimension=d;
                lrate=lr;

            end
        end
    end
end

disp("error= "+error+", N_layer= "+N_layer+", dimension= "+...
                                            dimension+", lrate= "+lrate );
```

```
% SHC Weights Optimization

T=5; % time interval [0,T]
N_layer=1; % number of hidden layer
dim=3; % dimension of any hidden layer
lrate=0.001; % learning rate
y0=1.5;

% testing times
n_test=50;
t_test=linspace(0, T, n_test);

% parameters initialization
error=0.01;
errormax=0.0005;
threshold=0;
thresholdmax=1000;
W=cell(1,N_layer+1);

while error>errormax && threshold<thresholdmax

    threshold=threshold+1;
    omega = cell(1, N_layer + 1); % array di matrici dei pesi
    theta = cell(1, N_layer + 1); % array di vettori di bias

    % Inizializzazione dei pesi con Xavier
    W = sqrt(2/(dim+2))*randn(dim,2);
    theta{1}=W(:,1); % bias 1
    omega{1}=W(:,2:end); % matrice 1 (spazio e tempo)

    for k=2:N_layer
        W=sqrt(2/ (2*dim))*randn(dim,dim+1);
        theta{k}=W(:, 1);
        omega{k}=W(:, 2:end);
    end

    W=sqrt(2/(1+dim))*randn(1,dim+1);
    theta{N_layer+1}=W(:,1);
    omega{N_layer+1}=W(:,2:end);

    % calculation of the new error
    [errornew,thetanew,omeganew]=SHCWeightsUpdate(N_layer,omega,theta, ...
                                        t_test,T,y0,lrate);

    if errornew<error
        error=errornew; %error update
        for i = 1:N_layer+1
            % saving omega and theta
            fileNameOmega = ['omega', num2str(i), '.xlsx'];
            writematrix(omeganew{i}, fileNameOmega);
            fileNameTheta = ['theta', num2str(i), '.xlsx'];
            writematrix(thetanew{i}, fileNameTheta);
        end
```

```
    end
end
disp("error= "+error);
```

```
% Test1

T=5; % time interval [0,T]
N_layer=1; % number of hidden layer
dim=3; % dimension of any hidden layer
lrate=0.001; % learning rate
y0=1.5; % initial condition


% testing times
n_test=50;
t_test=linspace(0, T, n_test);

% construction of omega and theta
omega=cell(1, N_layer + 1);
theta=cell(1, N_layer + 1);

%  Initialization of weights with Xavier
W = sqrt(2/(dim+2))*randn(dim,2);
theta{1}=W(:,1);
omega{1}=W(:,2:end);

for k=2:N_layer
    W=sqrt(2/ (2*dim))*randn(dim,dim+1);
    theta{k}=W(:, 1);
    omega{k}=W(:, 2:end);
end

W=sqrt(2/(1+dim))*randn(1,dim+1);
theta{N_layer+1}=W(:,1);
omega{N_layer+1}=W(:,2:end);

%calculation of the error
error=SHCLossFunction(N_layer,omega,theta,t_test,T,y0);
disp("error= "+error);

% weights update
[errornew,thetanew,omeganew]=SHCWeightsUpdate(N_layer,omega,theta, ...
                                                t_test,T,y0,lrate);
disp("errornew= "+errornew);

%saving the error and the weights
if errornew<error
    for i = 1:N_layer+1
        % saving omega and theta
        fileNameOmega = ['omega', num2str(i), '.xlsx'];
        writematrix(omeganew{i}, fileNameOmega);
        fileNameTheta = ['theta', num2str(i), '.xlsx'];
        writematrix(thetanew{i}, fileNameTheta);
    end
end

% Comparison of the graph with the exact solution
```

```
y=[];
for n=1:length(t_test)
    output=SHCNeuralNetwork(N_layer,omeganew,thetanew,t_test(n));
    y=[y,y0+ t_test(n)*output{N_layer+1}];
end

solution=@(z) (y0*exp(z))/(1+y0*(exp(z)-1));
sol=[];

for n=1:length(t_test)
    sol=[sol,solution(t_test(n))];
end

err=abs(sol-y).^2;
difference=sum(err)/n_test;


plot(t_test, y, '-r','DisplayName', 'Approximated solution');
hold on;
plot(t_test, sol, '-b','DisplayName', 'Exact solution');
xlabel('time');
ylabel('y(t)');
title('Graph of y(t)');
legend;
```

APPENDIX B - Script Test2 (1 of 2)

```
% Test2

T=5; % time interval [0,T]
N_layer=1; % number of hidden layer
dim=3; % dimension of any hidden layer
lrate=0.0001; % learning rate
y0=1.5; % initial condition

% testing times
n_test=50;
t_test=linspace(0, T, n_test);

% construction of omega and theta
omega=cell(1, N_layer + 1);
theta=cell(1, N_layer + 1);

%reading omega and theta
for i = 1:N_layer+1
    fileNameOmega = ['omega', num2str(i), '.xlsx'];
    omega{i} = readmatrix(fileNameOmega);
    fileNameTheta = ['theta', num2str(i), '.xlsx'];
    theta{i} = readmatrix(fileNameTheta);
end

%calculation of the error
error=SHCLossFunction(N_layer,omega,theta,t_test,T,y0);
disp("error= "+error);

% weights update
[errornew,thetanew,omeganew]=SHCWeightsUpdate(N_layer,omega,theta,...
                                              t_test,T,y0,lrate);
disp("errornew= "+errornew);

%saving the error and the weights
if errornew<error
    for i = 1:N_layer+1
        % saving omega and theta
        fileNameOmega = ['omega', num2str(i), '.xlsx'];
        writematrix(omeganew{i}, fileNameOmega);
        fileNameTheta = ['theta', num2str(i), '.xlsx'];
        writematrix(thetanew{i}, fileNameTheta);
    end
end

% Comparison of the graph with the exact solution
y=[];
for n=1:length(t_test)
    output=SHCNeuralNetwork(N_layer,omeganew,thetanew,t_test(n));
    y=[y,y0+ t_test(n)*output{N_layer+1}];
end

solution=@(z) (y0*exp(z))/(1+y0*(exp(z)-1));
sol=[];
```

```
for n=1:length(t_test)
    sol=[sol,solution(t_test(n))];
end

plot(t_test, y, '-r','DisplayName', 'Approximated solution');
hold on;
plot(t_test, sol, '-b','DisplayName', 'Exact solution');
xlabel('time');
ylabel('y(t)');
title('Graph of y(t)');
legend;
```

APPENDIX C - Function NeuralNetwork (1 of 1)

```
function X = NeuralNetwork (N_layer, omega, theta, X0)

% sigmoid function
sigmoid = @(u) 1./(1+exp(-u));

% column vector array of node values
X=cell(1,N_layer+1);
X{1}=sigmoid(omega{1}*X0+theta{1});

for i=2:N_layer
    % value of nodes in layer i
    X{i}=sigmoid(omega{i}*X{i-1}+theta{i});
end

% activation function is not applied to the last layer
X{N_layer+1}=omega{N_layer+1}*X{N_layer}+theta{N_layer+1};

end
```

```
function dt = TimeDerivative(N_layer, omega, theta, X0)

% calculation of node values
X=NeuralNetwork(N_layer, omega, theta, X0);

% calculation of the time derivative
dt=cell(1,N_layer+1);
dt{1}=X{1}.*(1-X{1}).*omega{1}(:,2);

for i=2:N_layer
    dt{i}=omega{i}*dt{i-1};
    dt{i}=X{i}.*(1-X{i}).*dt{i};
end

dt{N_layer+1}=omega{N_layer+1}*dt{N_layer};

end
```

APPENDIX C - Function SpaceDerivative (1 of 1)

```
function dx = SpaceDerivative(N_layer, omega, theta, X0)

% calculation of node values
X=NeuralNetwork(N_layer, omega, theta, X0);

% calculation of the time derivative
dx=cell(1,N_layer+1);
dx{1}=X{1}.*(1-X{1}).*omega{1}(:,1);

for i=2:N_layer
    dx{i}=omega{i}*dx{i-1};
    dx{i}=X{i}.*(1-X{i}).*dx{i};
end
dx{N_layer+1}=omega{N_layer+1}*dx{N_layer};

end
```

APPENDIX C - Function Laplacian (1 of 1)

```
function lap = Laplacian(N_layer, omega,theta, X0)

%calculation of the array of node values
X=NeuralNetwork(N_layer,omega,theta,X0);

%calculation of the space derivative
dx=SpaceDerivative(N_layer,omega,theta,X0);

%calculation of the laplacian
lap=cell(1,N_layer+1);
lap{1}=X{1}.*(1-X{1}).*(1-2*X{1}).*(omega{1}(:,1).^2);

for i=2:N_layer
    lap{i}=X{i}.*(1-X{1}).*(1-2*X{i}).*((omega{i}*dx{i-1}).^2)+...
        X{i}.*(1-X{i}).*(omega{i}*lap{i-1});
end

lap{N_layer+1} = omega{N_layer + 1}*lap{N_layer};

end
```

APPENDIX C - Function LossFunction (1 of 2)

```
function L = LossFunction(N_layer,omega,theta,X_test,T,a,b,ya,yb,dya,dyb)

% calculation of the outputs
u=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    X=NeuralNetwork(N_layer,omega,theta,X_test(:,i));
    u{i}=X{N_layer+1};
end

% calculation of the trial function
y=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    x=X_test(1,i);
    t=X_test(2,i);
    y{i}=ya(t)*(b-x)/(b-a)+yb(t)*(x-a)/(b-a)+...
        t*(b-x)*(x-a)/(b-a)^2*u{i};
end

% calculation of the time derivative of u
ut=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    dt=TimeDerivative(N_layer,omega,theta,X_test(:,i));
    ut{i}=dt{N_layer+1};
end

% calculation of the space derivative of u
ux=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    dx=SpaceDerivative(N_layer,omega,theta,X_test(:,i));
    ux{i}=dx{N_layer+1};
end

% calculation of the laplacian of u
uxx=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    dxx=Laplacian(N_layer,omega,theta,X_test(:,i));
    uxx{i}=dxx{N_layer+1};
end

% calculation of the time derivative of y
yt=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    x=X_test(1,i);
    t=X_test(2,i);
    yt{i}=dya(t)*(b-x)/(b-a)+dyb(t)*(x-a)/(b-a)+...
        (b-x)*(x-a)/(b-a)^2*u{i}+t*(b-x)*(x-a)/(b-a)^2*ut{i};
end

% calculation of the laplacian of y
yxx=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    x=X_test(1,i);
    t=X_test(2,i);
```

```
    yxx{i}=-t*(2/(b-a)^2)*u{i}+2*t*(-2*x+b+a)/(b-a)^2*ux{i}+...
        t*(b-x)*(x-a)/(b-a)^2*uxx{i};
end

% calculation of the loss function
sum=0;
for i=1:length(X_test(1,:))
    sum=sum+(yt{i}-yxx{i}-y{i}*(1-y{i}))^2;
end
L=(T*(b-a)/length(X_test(1,:)))*sum;

end
```

APPENDIX C - Function WeightsUpdate (1 of 4)

```
function [error,thetanew,omeganew] = WeightsUpdate(N_layer,omega,theta,...
                                X_test,ya,yb,dya,dyb,a,b,T,lrate)

% parameters initialization
error=10;
errormax=0.0001;
threshold=1;
thresholdmax=500;
W=cell(1,N_layer+1);

% weights update
while error>errormax && threshold<thresholdmax

    % training nodes
    ntime_train=50;
    time_train=T*rand(1,ntime_train);

    nspace_train=20;
    space_train=a+(b-a)*rand(1,nspace_train);

    X_train=[];
    for i=1:ntime_train
        for j=1:nspace_train
            X_train=[X_train,[space_train(j),time_train(i)]'];
        end
    end

    % calculation of the outputs
    u=cell(1, length(X_train(1,:)));
    for i=1:length(X_train(1,:))
        X=NeuralNetwork(N_layer, omega, theta, X_train(:,i));
        u{i}=X{N_layer+1};
    end

    % calculation of the trial function
    y=cell(1, length(X_train(1,:)));
    for i=1:length(X_train(1,:))
        x=X_train(1,i);
        t=X_train(2,i);
        y{i}=ya(t)*(b-x)/(b-a)+yb(t)*(x-a)/(b-a)+...
            t*(b-x)*(x-a)/(b-a)^2*u{i};
    end

    % calculation of the time derivative of u
    ut=cell(1,length(X_train(1,:)));
    for i=1:length(X_train(1,:))
        dt=TimeDerivative(N_layer, omega,theta, X_train(:,i));
        ut{i}=dt{N_layer+1};
    end

    % calculation of the space derivative of u
    ux=cell(1,length(X_train(1,:)));
    for i=1:length(X_train(1,:))
```

```
    dx=SpaceDerivative(N_layer, omega,theta, X_train(:,i));
    ux{i}=dx{N_layer+1};
end

% calculation of the laplacian of u
uxx=cell(1,length(X_train(1,:)));
for i=1:length(X_train(1,:))
    dxx=Laplacian(N_layer, omega,theta, X_train(:,i));
    uxx{i}=dxx{N_layer+1};
end

% calculation of the time derivative of y
yt=cell(1,length(X_train(1,:)));
for i=1:length(X_train(1,:))
    x=X_train(1,i);
    t=X_train(2,i);
    yt{i}=dya(t)*(b-x)/(b-a)+dyb(t)*(x-a)/(b-a)+...
        (b-x)*(x-a)/(b-a)^2*u{i}+t*(b-x)*(x-a)/(b-a)^2*ut{i};
end

% calculation of the laplacian of y
yxx=cell(1,length(X_train(1,:)));
for i=1:length(X_train(1,:))
    x=X_train(1,i);
    t=X_train(2,i);
    yxx{i}=-t*(2/(b-a)^2)*u{i}+2*t*(-2*x+b+a)/(b-a)^2*ux{i}+...
        t*(b-x)*(x-a)/(b-a)^2*uxx{i};
end

% initialization of alfa, betax, betat and gamma
alfa=cell(length(X_train(1,:)),N_layer+1);
betax=cell(length(X_train(1,:)),N_layer+1);
betat=cell(length(X_train(1,:)),N_layer+1);
gamma=cell(length(X_train(1,:)),N_layer+1);


for i=1:length(X_train(1,:))
    % weights update of the layer N+1
    X=NeuralNetwork(N_layer,omega,theta,X_train(:,i));
    alfa{i,N_layer+1}=1;
    betax{i,N_layer+1}=0;
    betat{i,N_layer+1}=0;
    gamma{i,N_layer+1}=0;
    x=X_train(1,i);
    t=X_train(2,i);
    dy=t*(b-x)*(x-a)/(b-a)^2;
    dyt=(b-x)*(x-a)/(b-a)^2;
    dyxx=-t*2/(b-a)^2;
    dL=(yt{i}-yxx{i}-y{i}.*(1-y{i})).*...
                (dyt-dyxx-(1-2*y{i}).*dy)*[1;X{N_layer}]';
    W{N_layer+1}=[theta{N_layer+1},omega{N_layer+1}];
    W{N_layer+1}=W{N_layer+1}-lrate*dL;
    theta{N_layer+1}=W{N_layer+1}(:,1);
```

```
omega{N_layer+1}=W{N_layer+1}(:,2:end);

for k=N_layer:-1:2
    % weights update of the layer k
    X=NeuralNetwork(N_layer,omega,theta,X_train(:,i));
    timederiv=TimeDerivative(N_layer,omega,theta,X_train(:,i));
    spacederiv=SpaceDerivative(N_layer,omega,theta,X_train(:,i));
    lap=Laplacian(N_layer,omega,theta,X_train(:,i));
    dsigmax=X{k}.*(1-X{k}).*(1-2*X{k}).*(omega{k}*spacederiv{k-1});
    dsigmat=X{k}.*(1-X{k}).*(1-2*X{k}).*(omega{k}*timederiv{k-1});
    dsigmax2=X{k}.*(1-X{k}).*(1-6*X{k}.*(1-X{k})).*...
        (omega{k}*spacederiv{k-1}).^2+X{k}.*(1-X{k}).*...
        (1-2*X{k}).*(omega{k}*lap{k-1});
    alfa{i,k}=alfa{i,k+1}.*omega{k+1}*(X{k}.*(1-X{k}));
    betax{i,k}=betax{i,k+1}.*omega{k+1}*(X{k}.*(1-X{k}))+...
                alfa{i,k+1}.*(omega{k+1}*dsigmax);
    betat{i,k}=betat{i,k+1}.*omega{k+1}*(X{k}.*(1-X{k}))+...
                alfa{i,k+1}.*(omega{k+1}*dsigmat);
    gamma{i,k}=gamma{i,k+1}.*omega{k+1}*(X{k}.*(1-X{k}))+...
                2*betax{i,k+1}.*(omega{k+1}*dsigmax)+...
                alfa{i,k+1}.*(omega{k+1}*dsigmax2);
    x=X_train(1,i);
    t=X_train(2,i);
    dy=t*(b-x)*(x-a)/(b-a)^2*alfa{i,k};
    dyt=(b-x)*(x-a)/(b-a)^2*alfa{i,k}+...
       t*(b-x)*(x-a)/(b-a)^2*betat{i,k};
    dyxx=-t*2/(b-a)^2*alfa{i,k}+2*t*(-2*x+b+a)/(b-a)^2*betax{i,k}+...
        t*(b-x)*(x-a)/(b-a)^2*gamma{i,k};
    dL=(yt{i}-yxx{i}-y{i}.*(1-y{i})).*...
        (dyt-dyxx-(1-2*y{i}).*dy)*[1;X{k-1}]';
    W{k}=[theta{k},omega{k}];
    W{k}=W{k}-lrate*dL;
    theta{k}=W{k}(:,1);
    omega{k}=W{k}(:,2:end);
end

% weights update of the layer 1
X=NeuralNetwork(N_layer,omega,theta,X_train(:,i));
dsigmax=X{1}.*(1-X{1}).*(1-2*X{1}).*omega{1}(:,1);
dsigmat=X{1}.*(1-X{1}).*(1-2*X{1}).*omega{1}(:,2);
dsigmax2=X{1}.*(1-X{1}).*(1-6*X{1}.*(1-X{1})).*omega{1}(:,1).^2;
alfa{i,1}=alfa{i,2}.*omega{2}*(X{1}.*(1-X{1}));
betax{i,1}=betax{i,2}.*omega{2}*(X{1}.*(1-X{1}))+...
    alfa{i,2}.*(omega{2}*dsigmax);
betat{i,1}=betat{i,2}.*omega{2}*(X{1}.*(1-X{1}))+...
            alfa{i,2}.*(omega{2}*dsigmat);
gamma{i,1}=gamma{i,2}.*omega{2}*(X{1}.*(1-X{1}))+...
            2*betax{i,2}.*(omega{2}*dsigmax)+...
            alfa{i,2}.*(omega{2}*dsigmax2);
x=X_train(1,i);
t=X_train(2,i);
dy=t*(b-x)*(x-a)/(b-a)^2*alfa{i,1};
dyt=(b-x)*(x-a)/(b-a)^2*alfa{i,1}+t*(b-x)*(x-a)/(b-a)^2*betat{i,1};
```

```
        dyxx=-t*2/(b-a)^2*alfa{i,1}+2*t*(-2*x+b+a)/(b-a)^2*betax{i,1}+...
            t*(b-x)*(x-a)/(b-a)^2*gamma{i,1};
        dL=(yt{i}-yxx{i}-y{i}.*(1-y{i})).*...
            (dyt-dyxx-(1-2*y{i}).*dy)*[1;X_train(:,i)]';
        W{1}=[theta{1},omega{1}];
        W{1}=W{1}-lrate*dL;
        theta{1}=W{1}(:,1);
        omega{1}=W{1}(:,2:end);
    end
    % error calculation
    errornew=LossFunction(N_layer,omega,theta,X_test,T,a,b,ya,yb,dya,dyb);

     % error update
    if errornew<error
        error=errornew;
        thetanew=theta;
        omeganew=omega;
    end

    % threshold update
    threshold=threshold+1;

end

end
```

```
% test

T=5; % time interval [0,T]
N_layer=1; % number of hidden layer
dim=2; % dimension of any hidden layer
lrate=0.001; % learning rate
y0=1.5; % initial condition
a=-3;
b=3;

%boundary conditions
ya= @(t) (y0*exp(t))/(1+y0*(exp(t)-1));
yb=ya;
dya= @(t) (y0*exp(t)*(1-y0))/((1+y0*(exp(t)-1))^2);
dyb=dya;

% testing nodes
ntime_test=50;
time_test=linspace(0, T, ntime_test);

nspace_test=20;
space_test=linspace(a,b, nspace_test);

X_test=[];
for i=1:ntime_test
    for j=1:nspace_test
        X_test=[X_test,[space_test(j),time_test(i)]'];
    end
end

% construction of omega and theta
omega=cell(1, N_layer + 1);
theta=cell(1, N_layer + 1);

% Xavier initialization
W=sqrt(2/(dim + 2))*randn(dim, 3);
theta{1}=W(:, 1); % bias 1
omega{1}=W(:, 2:end); % matrice 1 (spazio e tempo)

for k=2:N_layer
    W=sqrt(2/(2*dim))*randn(dim,dim+1);
    theta{k}=W(:,1);
    omega{k}=W(:,2:end);
end

W=sqrt(2/(1+dim))*randn(1,dim+1);
theta{N_layer+1}=W(:,1);
omega{N_layer+1}=W(:,2:end);

%calculation of the error
error=LossFunction(N_layer,omega,theta,X_test,T,a,b,ya,yb,dya,dyb);
disp("error= "+error);
```

```
% weights update
[errornew,thetanew,omeganew]=WeightsUpdate(N_layer,omega,theta, ...
                                X_test,ya,yb,dya,dyb,a,b,T,lrate);
disp("errornew= "+errornew);

%saving the error and the new weights
if errornew<error
    omega=omeganew;
    theta=thetanew;
    for i = 1:N_layer+1
        % saving omega and theta
        fileNameOmega = ['omega', num2str(i), '.xlsx'];
        writematrix(omeganew{i}, fileNameOmega);
        fileNameTheta = ['theta', num2str(i), '.xlsx'];
        writematrix(thetanew{i}, fileNameTheta);
    end
end


% Comparison of the graph with the exact solution on the bound

% Solution evaluated on the boundary a
bya=[];
for i=1:length(X_test(1,:))
    X=NeuralNetwork(N_layer, omega, theta, X_test(:,i));
    if X_test(1,i)==a
        x=a;
        t=X_test(2,i);
        bya=[bya,ya(t)*(b-x)/(b-a)+yb(t)*(x-a)/(b-a)+...
            t*(b-x)*(x-a)/(b-a)^2*X{N_layer+1}];
    end
end

% Solution evaluated on the boundary b
byb=[];
for i=1:length(X_test(1,:))
    X=NeuralNetwork(N_layer, omega, theta, X_test(:,i));
    if X_test(1,i)==b
        x=b;
        t=X_test(2,i);
        byb=[byb,ya(t)*(b-x)/(b-a) + yb(t)*(x-a)/(b-a)+...
            t*(b-x)*(x-a)/(b-a)^2*X{N_layer+1}];
    end
end

% exact solution on the boundary
solution=@(z) (y0*exp(z))/(1+y0*(exp(z)-1));
sol=[];

for n=1:length(time_test)
    sol=[sol,solution(time_test(n))];
end
```

```
%error on the boundary a
erra=abs(sol-bya);
errorA=sum(erra)/ntime_test;
disp("errorA= "+errorA);

%error on the boundary b
errb=abs(sol-byb);
errorB=sum(errb)/ntime_test;
disp("errorB= "+errorB);

%plot of the comparison with the exact solution on the bound
figure;
plot(time_test, sol, '-b','DisplayName', 'Exact solution');
hold on;
plot(time_test, bya, '-r','DisplayName', 'Approximated solution on a');
plot(time_test, byb, '-g','DisplayName', 'Approximated solution on b');
xlabel('time');
ylabel('y(t)');
title('Graph of y(t)');
legend;


%construction of the approximated solution
Y=[];
for i=1:nspace_test
    for j=1:ntime_test
        X_test=[space_test(i),time_test(j)]';
        X=NeuralNetwork(N_layer, omega, theta, X_test);
        x=X_test(1);
        t=X_test(2);
        Y(i,j)=ya(t)*(b-x)/(b-a)+yb(t)*(x-a)/(b-a)+...
            t*((b-x)*(x-a))/((b-a)^2)*X{N_layer+1};
    end
end

Y=Y';

%graph of the approximated solution
figure;
[X,T]=meshgrid(space_test, time_test);
surf(X,T,Y);
xlabel('x (space)');
ylabel('t (time)');
zlabel('y(x,t)');
```

APPENDIX D - Function GCNeuralNetwork (1 of 1)

```
function X = GCNeuralNetwork (N_layer, omega, theta, X0)

% sigmoid function
sigmoid = @(u) 1./(1+exp(-u));

% column vector array of node values
X=cell(1,N_layer+1);
X{1}=sigmoid(omega{1}*X0+theta{1});

for i=2:N_layer
    % value of nodes in layer i
    X{i}=sigmoid(omega{i}*X{i-1}+theta{i});
end
% activation function is not applied to the last layer
X{N_layer+1}=omega{N_layer+1}*X{N_layer}+theta{N_layer+1};

end
```

APPENDIX D - Function GCTimeDerivative (1 of 1)

```
function dt = GCTimeDerivative(N_layer, omega, theta, X0)

% calculation of node values
X=GCNeuralNetwork(N_layer, omega, theta, X0);

% calculation of the time derivative
dt=cell(1,N_layer+1);
dt{1}=X{1}.*(1-X{1}).*omega{1}(:,2);

for i=2:N_layer
    dt{i}=X{i}.*(1-X{i}).*omega{i}*dt{i-1};
end
dt{N_layer+1}=omega{N_layer+1}*dt{N_layer};

end
```

APPENDIX D - Function GCSpaceDerivative (1 of 1)

```
function dx = GCSpaceDerivative(N_layer, omega, theta, X0)

% calculation of node values
X=GCNeuralNetwork(N_layer, omega, theta, X0);

% calculation of the time derivative
dx=cell(1,N_layer+1);
dx{1}=X{1}.*(1-X{1}).*omega{1}(:,1);

for i=2:N_layer
    dx{i}=X{i}.*(1-X{i}).*omega{i}*dx{i-1};
end
dx{N_layer+1}=omega{N_layer+1}*dx{N_layer};

end
```

APPENDIX D - Function GCLaplacian (1 of 1)

```
function lap = GCLaplacian(N_layer, omega,theta, X0)

%calculation of the array of node values
X= GCNeuralNetwork(N_layer, omega, theta, X0);

%calculation of the derivative
dx = GCSpaceDerivative(N_layer, omega,theta, X0);

%calculation of the laplacian
lap=cell(1,N_layer+1);
lap{1}=X{1}.*(1-X{1}).*(1-2*X{1}).*(omega{1}(:,1).^2);

for i=2:N_layer
    lap{i}=X{i}.*(1-X{1}).*(1-2*X{i}).*((omega{i}*dx{i-1}).^2)+...
        X{i}.*(1-X{i}).*(omega{i}*lap{i-1});
end

lap{N_layer+1} = omega{N_layer + 1}*lap{N_layer};

end
```

APPENDIX D - Function GCLossFunction (1 of 2)

```
function L=GCLossFunction(N_layer,omega,theta,X_test,X_test0,T,a,b,y0,ya,dya)

X_test=[X_test,X_test0];

% calculation of the outputs
u=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    X=GCNeuralNetwork(N_layer,omega,theta,X_test(:,i));
    u{i}=X{N_layer+1};
end

% calculation of the trial function
y=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    x=X_test(1,i);
    t=X_test(2,i);
    y{i}=ya(t)+(b-x)*(x-a)/(b-a)^2*u{i};
end

% calculation of the time derivative of u
ut=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    dt=GCTimeDerivative(N_layer,omega,theta,X_test(:,i));
    ut{i}=dt{N_layer+1};
end

% calculation of the space derivative of u
ux=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    dx=GCSpaceDerivative(N_layer,omega,theta,X_test(:,i));
    ux{i}=dx{N_layer+1};
end

% calculation of the laplacian of u
uxx=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    dxx=GCLaplacian(N_layer,omega,theta,X_test(:,i));
    uxx{i}=dxx{N_layer+1};
end

% calculation of the time derivative of y
yt=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    x=X_test(1,i);
    t=X_test(2,i);
    yt{i}=dya(t)+(b-x)*(x-a)/(b-a)^2*ut{i};
end

% calculation of the laplacian of y
yxx=cell(1,length(X_test(1,:)));
for i=1:length(X_test(1,:))
    x=X_test(1,i);
    yxx{i}=-2/(b-a)^2*u{i}+2*(-2*x+b+a)/(b-a)^2*ux{i}+...
```

```
        (b-x)*(x-a)/(b-a)^2*uxx{i};
end

% calculation of the loss function
L1=0;
L2=0;
for i=1:length(X_test(1,:))
    x=X_test(1,i);
    t=X_test(2,i);
    if t~=0
        L1=L1+(yt{i}-yxx{i}-y{i}*(1-y{i}))^2;
    else
        L2=L2+(y{i}-y0(x))^2;
    end
end
L1=0.1*T*(b-a)/length(X_test(1,:))*L1;
L2=0.9*(b-a)/length(X_test0(1,:))*L2;
L=L1+L2;

end
```

```
function [error,thetanew,omeganew]=GCWeightsUpdate(N_layer,omega,theta,...
                        X_test,X_test0,y0,ya,dya,a,b,T,lrate)

% parameters initialization
error=10;
errormax=0.005;
threshold=1;
thresholdmax=10000;
W=cell(1,N_layer+1);
max=0;
errornew=1000;
omeganew=omega;
thetanew=theta;
% weights update
while error>errormax && threshold<thresholdmax
    % training nodes
    ntime_train=10;
    time_train=T*rand(1,ntime_train);

    nspace_train=10;
    space_train=a+(b-a)*rand(1,nspace_train);

    X_train=[];
    for i=1:ntime_train
        for j=1:nspace_train
            X_train=[X_train,[space_train(j),time_train(i)]'];
        end
    end

    nspace_train0=200;
    space_train0=a+(b-a)*rand(1,nspace_train0);

    X_train0=[];
    for i=1:nspace_train0
        X_train0=[X_train0,[space_train0(i),0]'];
    end

    X_train=[X_train,X_train0];

    X_train=X_train(:, randperm(size(X_train, 2)));

    % calculation of the outputs
    u=cell(1,length(X_train(1,:)));
    for i=1:length(X_train(1,:))
        X=GCNeuralNetwork(N_layer,omega,theta,X_train(:,i));
        u{i}=X{N_layer+1};
    end

    % calculation of the trial function
    y=cell(1,length(X_train(1,:)));
    for i=1:length(X_train(1,:))
        x=X_train(1,i);
        t=X_train(2,i);
```

```
    y{i}=ya(t)+(b-x)*(x-a)/(b-a)^2*u{i};
end

% calculation of the time derivative of u
ut=cell(1,length(X_train(1,:)));
for i=1:length(X_train(1,:))
    dt=GCTimeDerivative(N_layer,omega,theta,X_train(:,i));
    ut{i}=dt{N_layer+1};
end

% calculation of the space derivative of u
ux=cell(1,length(X_train(1,:)));
for i=1:length(X_train(1,:))
    dx=GCSpaceDerivative(N_layer,omega,theta,X_train(:,i));
    ux{i}=dx{N_layer+1};
end

% calculation of the laplacian of u
uxx=cell(1,length(X_train(1,:)));
for i=1:length(X_train(1,:))
    dxx=GCLaplacian(N_layer,omega,theta,X_train(:,i));
    uxx{i}=dxx{N_layer+1};
end

% calculation of the time derivative of y
yt=cell(1,length(X_train(1,:)));
for i=1:length(X_train(1,:))
    x=X_train(1,i);
    t=X_train(2,i);
    yt{i}=dya(t)+(b-x)*(x-a)/(b-a)^2*ut{i};
end

% calculation of the laplacian of y
yxx=cell(1,length(X_train(1,:)));
for i=1:length(X_train(1,:))
    x=X_train(1,i);
    yxx{i}=-2/(b-a)^2*u{i}+2*(-2*x+b+a)/(b-a)^2*ux{i}+...
        (b-x)*(x-a)/(b-a)^2*uxx{i};
end

% initialization of alfa, betax, betat and gamma
alfa=cell(length(X_train(1,:)),N_layer+1);
alfa0=cell(nspace_train0,N_layer+1);
betax=cell(length(X_train(1,:)),N_layer+1);
betat=cell(length(X_train(1,:)),N_layer+1);
gamma=cell(length(X_train(1,:)),N_layer+1);

for i=1:length(X_train(1,:))
    % weights update of the layer N+1
    X=GCNeuralNetwork(N_layer,omega,theta,X_train(:,i));
    x=X_train(1,i);
    t=X_train(2,i);
    if t~=0
```

```
            alfa{i,N_layer+1}=1;
            betax{i,N_layer+1}=0;
            betat{i,N_layer+1}=0;
            gamma{i,N_layer+1}=0;
            dy=(b-x)*(x-a)/(b-a)^2;
            dyt=0;
            dyxx=-2/(b-a)^2;
            dL1=2*(yt{i}-yxx{i}-y{i}.*(1-y{i}))*(dyt-dyxx-(1-2*y{i})*dy)*[1;X↙
{N_layer}]';
            W{N_layer+1}=[theta{N_layer+1},omega{N_layer+1}];
            W{N_layer+1}=W{N_layer+1}-lrate*dL1;
            theta{N_layer+1}=W{N_layer+1}(:,1);
            omega{N_layer+1}=W{N_layer+1}(:,2:end);
        else
            alfa0{i,N_layer+1}=1;
            dy=(b-x)*(x-a)/(b-a)^2;
            dL2=2*(y0(a)+(b-x)*(x-a)/(b-a)^2*X{N_layer+1}-y0(x))*dy*[1;X↙
{N_layer}]';
            W{N_layer+1}=[theta{N_layer+1},omega{N_layer+1}];
            W{N_layer+1}=W{N_layer+1}-lrate*dL2;
            theta{N_layer+1}=W{N_layer+1}(:,1);
            omega{N_layer+1}=W{N_layer+1}(:,2:end);
        end

        for k=N_layer:-1:2
            % weights update of the layer k
            X=GCNeuralNetwork(N_layer,omega,theta,X_train(:,i));
            x=X_train(1,i);
            t=X_train(2,i);
            if t~=0
                timeder=GCTimeDerivative(N_layer,omega,theta,X_train(:,i));
                spaceder=GCSpaceDerivative(N_layer,omega,theta,X_train(:,i));
                lap=GCLaplacian(N_layer,omega,theta,X_train(:,i));
                dsigmax=X{k}.*(1-X{k}).*(1-2*X{k}).*(omega{k}*spaceder{k-1});
                dsigmat=X{k}.*(1-X{k}).*(1-2*X{k}).*(omega{k}*timeder{k-1});
                dsigmax2=X{k}.*(1-X{k}).*(1-6*X{k}.*(1-X{k})).*...
                    (omega{k}*spaceder{k-1}).^2+X{k}.*(1-X{k}).*...
                    (1-2*X{k}).*(omega{k}*lap{k-1});
                alfa{i,k}=alfa{i,k+1}.*(omega{k+1}*(X{k}.*(1-X{k})));
                betax{i,k}=betax{i,k+1}.*(omega{k+1}*(X{k}.*(1-X{k})))+...
                    alfa{i,k+1}.*(omega{k+1}*dsigmax);
                betat{i,k}=betat{i,k+1}.*(omega{k+1}*(X{k}.*(1-X{k})))+...
                            alfa{i,k+1}.*(omega{k+1}*dsigmat);
                gamma{i,k}=gamma{i,k+1}.*(omega{k+1}*(X{k}.*(1-X{k})))+...
                            2*betax{i,k+1}.*(omega{k+1}*dsigmax)+...
                            alfa{i,k+1}.*(omega{k+1}*dsigmax2);
                dy=(b-x)*(x-a)/(b-a)^2*alfa{i,k};
                dyt=(b-x)*(x-a)/(b-a)^2*betat{i,k};
                dyxx=-2/(b-a)^2*alfa{i,k}+...
                    2*(-2*x+b+a)/(b-a)^2*betax{i,k}+...
                    (b-x)*(x-a)/(b-a)^2*gamma{i,k};
                dL1=2*(yt{i}-yxx{i}-y{i}.*(1-y{i}))*...
                                (dyt-dyxx-(1-2*y{i})*dy)*[1;X{k-1}]';
```

```
                W{k}=[theta{k},omega{k}];
                W{k}=W{k}-lrate*dL1;
                theta{k}=W{k}(:,1);
                omega{k}=W{k}(:,2:end);
            else
                alfa0{i,k}=alfa0{i,k+1}.*(omega{k+1}*(X{k}.*(1-X{k})));
                dy=(b-x)*(x-a)/(b-a)^2*alfa0{i,k};
                dL2=2*(y0(a)+(b-x)*(x-a)/(b-a)^2*X{N_layer+1}-y0(x))*dy*[1;X{k-
1}]';
                W{k}=[theta{k},omega{k}];
                W{k}=W{k}-lrate*dL2;
                theta{k}=W{k}(:,1);
                omega{k}=W{k}(:,2:end);
            end
        end
        % weights update of the layer 1
        X=GCNeuralNetwork(N_layer,omega,theta,X_train(:,i));
        x=X_train(1,i);
        t=X_train(2,i);
        if t~=0
            dsigmax=X{1}.*(1-X{1}).*(1-2*X{1}).*omega{1}(:,1);
            dsigmat=X{1}.*(1-X{1}).*(1-2*X{1}).*omega{1}(:,2);
            dsigmax2=X{1}.*(1-X{1}).*(1-6*X{1}.*(1-X{1})).*omega{1}(:,1).^2;
            alfa{i,1}=alfa{i,2}.*omega{2}*(X{1}.*(1-X{1}));
            betax{i,1}=betax{i,2}.*omega{2}*(X{1}.*(1-X{1}))+...
                alfa{i,2}.*(omega{2}*dsigmax);
            betat{i,1}=betat{i,2}.*omega{2}*(X{1}.*(1-X{1}))+...
                        alfa{i,2}.*(omega{2}*dsigmat);
            gamma{i,1}=gamma{i,2}.*omega{2}*(X{1}.*(1-X{1}))+...
                        2*betax{i,2}.*(omega{2}*dsigmax)+...
                        alfa{i,2}.*(omega{2}*dsigmax2);
            x=X_train(1,i);
            dy=(b-x)*(x-a)/(b-a)^2*alfa{i,1};
            dyt=(b-x)*(x-a)/(b-a)^2*betat{i,1};
            dyxx=-2/(b-a)^2*alfa{i,1}+...
                2*(-2*x+b+a)/(b-a)^2*betax{i,1}+...
                (b-x)*(x-a)/(b-a)^2*gamma{i,1};
            dL1=(yt{i}-yxx{i}-y{i}.*(1-y{i})).*...
                        (dyt-dyxx-(1-2*y{i}).*dy)*[1;X_train(:,i)]';
            W{1}=[theta{1},omega{1}];
            W{1}=W{1}-lrate*dL1;
            theta{1}=W{1}(:,1);
            omega{1}=W{1}(:,2:end);
        else
            alfa0{i,1}=alfa0{i,2}.*(omega{2}*(X{1}.*(1-X{1})));
            dy=(b-x)*(x-a)/(b-a)^2*alfa0{i,1};
            dL2=2*(y0(a)+(b-x)*(x-a)/(b-a)^2*X{N_layer+1}-y0(x))*dy*[1;X_train(:,
i)]';
            W{1}=[theta{1},omega{1}];
            W{1}=W{1}-lrate*dL2;
            theta{1}=W{1}(:,1);
            omega{1}=W{1}(:,2:end);
        end
```

APPENDIX D - Function GCWeightsUpdate (5 of 5)

```
    end

    errorpred=errornew;
    % error calculation
    errornew=GCLossFunction(N_layer,omega,theta,X_test,X_test0,T,a,b,y0,ya,dya);

    % error update
    if errornew<error
        error=errornew;
        omeganew=omega;
        thetanew=theta;
    end

    if errornew>errorpred
        max=max+1;
    end

    if errornew>10000
        threshold=thresholdmax-1;
    end

    % threshold update
    threshold=threshold+1;

end

end
```

APPENDIX D - Script GC Parameters Optimization (1 of 2)

```
% GC Parameters Optimization
T=5;
N_layer=3; % number of hidden layer
dim=10; % dimension of any hidden layer
lrate=0.01; % learning rate
y0=@(x) cos(pi*x/3)+2.5; % initial condition

a=-3;
b=3;

%boundary conditions
ya= @(t) y0(a)*exp(t)/(1+y0(a)*(exp(t)-1));
dya= @(t) (y0(a)*exp(t)*(1-y0(a)))/((1+y0(a)*(exp(t)-1))^2);

yb= @(t) (y0(b)*exp(t))/(1+y0(b)*(exp(t)-1));
dyb= @(t) (y0(b)*exp(t)*(1-y0(b)))/((1+y0(b)*(exp(t)-1))^2);

% testing nodes
ntime_test=50;
time_test=linspace(0, T, ntime_test);

nspace_test=20;
space_test=linspace(a,b, nspace_test);

X_test=[];
for i=1:ntime_test
    for j=1:nspace_test
        X_test=[X_test,[space_test(j),time_test(i)]'];
    end
end

nspace_test0=100;
space_test0=linspace(a,b, nspace_test0);

X_test0=[];
for i=1:nspace_test0
    X_test0=[X_test0,[space_test0(i),0]'];
end


error=1;
errormax=0.005;

for n=1:4
    for d=2:20

        % construction of omega and theta
        omega = cell(1, n + 1);
        theta = cell(1, n + 1);

        % Initialization of weights with Xavier
        W =sqrt(2/(d+2))*randn(d,3);
        theta{1} = W(:, 1); % bias 1
```

APPENDIX D - Script GC Parameters Optimization (2 of 2)

```
        omega{1} = W(:, 2:end); % matrice 1 (spazio e tempo)

        for k = 2:n
            W = sqrt(2/(2*d))*randn(d,d+1);
            theta{k}=W(:,1);
            omega{k}=W(:,2:end);
        end

        W = sqrt(2/(1+d))*randn(1,d+1);
        theta{n+1}=W(:,1);
        omega{n+1}=W(:,2:end);

        %Since the initial condition is an even function, we impose symmetry
        for i=1:d
            omega{1}(d-i+1,1)=-omega{1}(i,1);
            omega{1}(d-i+1,2)=omega{1}(i,2);
            theta{1}(d-i+1)=theta{1}(i);
            omega{2}(:,d-i+1)=omega{2}(:,i);
        end

        % weights update
        [errornew,thetanew,omeganew]=GCWeightsUpdate(n,omega,theta,...
                X_test,X_test0,y0,ya,dya,a,b,T,lrate);

        %saving the error and the new weights
        if errornew<error
            error=errornew;
            for i = 1:n+1
                fileNameOmega = ['omega', num2str(i), '.xlsx'];
                writematrix(omeganew{i}, fileNameOmega);
                fileNameTheta = ['theta', num2str(i), '.xlsx'];
                writematrix(thetanew{i}, fileNameTheta);
            end

            %updating the parameters
            N_layer=n;
            dim=d;
        end
    end
end

disp("error= "+error+", N_layer= "+N_layer+", dimension= "+...
                                                dim+", lrate= "+lrate );
```

APPENDIX D - Script GC Weights Optimization (1 of 2)

```
% GC Weights Optimization

T=5; % time interval [0,T]
N_layer=2; % number of hidden layer
dim=19; % dimension of any hidden layer
lrate=0.01; % learning rate
y0=@(x) cos(pi*x/3)+2.5; % initial condition

a=-3;
b=3;

ya= @(t) y0(a)*exp(t)/(1+y0(a)*(exp(t)-1));
dya= @(t) y0(a)*exp(t)*(1-y0(a))/(1+y0(a)*(exp(t)-1))^2;

yb= @(t) (y0(b)*exp(t))/(1+y0(b)*(exp(t)-1));
dyb= @(t) (y0(b)*exp(t)*(1-y0(b)))/((1+y0(b)*(exp(t)-1))^2);

% testing nodes
ntime_test=100;
time_test=linspace(0, T, ntime_test);

nspace_test=50;
space_test=linspace(a,b, nspace_test);

X_test=[];
for i=1:ntime_test
    for j=1:nspace_test
        X_test=[X_test,[space_test(j),time_test(i)]'];
    end
end

nspace_test0=100;
space_test0=linspace(a,b, nspace_test0);

X_test0=[];
for i=1:nspace_test0
    X_test0=[X_test0,[space_test0(i),0]'];
end


% parameters initialization
error=10;
errormax=0.001;
threshold=0;
thresholdmax=1000;
W=cell(1,N_layer+1);

while error>errormax && threshold<thresholdmax

    threshold=threshold+1;

    omega = cell(1, N_layer + 1); % array di matrici dei pesi
    theta = cell(1, N_layer + 1); % array di vettori di bias
```

APPENDIX D - Script GC Weights Optimization (2 of 2)

```
    % Initialization of weights with Xavier
    W = sqrt(2/(dim+2))*randn(dim,3);
    theta{1} = W(:, 1);
    omega{1} = W(:, 2:end);

    for k = 2:N_layer
        W = sqrt(2/(2*dim))*randn(dim,dim+1);
        theta{k}=W(:,1);
        omega{k}=W(:,2:end);
    end

    W = sqrt(2/(1+dim))*randn(1,dim+1);
    theta{N_layer+1}=W(:,1);
    omega{N_layer+1}=W(:,2:end);

    %Since the initial condition is an even function, we impose symmetry
    for i=1:dim
        omega{1}(dim-i+1,1)=-omega{1}(i,1);
        omega{1}(dim-i+1,2)=omega{1}(i,2);
        theta{1}(dim-i+1)=theta{1}(i);
        omega{2}(:,dim-i+1)=omega{2}(:,i);
    end

    % weights update
    [errornew,thetanew,omeganew]=GCWeightsUpdate(N_layer,omega,theta,...
                        X_test,X_test0,y0,ya,dya,a,b,T,lrate);

    %saving the error and the new weights
    if errornew<error
        error=errornew;
        for i = 1:N_layer+1
            fileNameOmega = ['omega', num2str(i), '.xlsx'];
            writematrix(omeganew{i}, fileNameOmega);
            fileNameTheta = ['theta', num2str(i), '.xlsx'];
            writematrix(thetanew{i}, fileNameTheta);
        end
    end
end

disp("error= "+error);
```

APPENDIX D - Script GC Test (1 of 3)

```
% test
T=5;
N_layer=2; % number of hidden layer
dim=19; % dimension of any hidden layer
lrate=0.001; % learning rate
y0=@(x) cos(pi*x/3)+2.5; % initial condition

a=-3;
b=3;

ya= @(t) y0(a)*exp(t)/(1+y0(a)*(exp(t)-1));
dya= @(t) (y0(a)*exp(t)*(1-y0(a)))/((1+y0(a)*(exp(t)-1))^2);

yb= @(t) (y0(b)*exp(t))/(1+y0(b)*(exp(t)-1));
dyb= @(t) (y0(b)*exp(t)*(1-y0(b)))/((1+y0(b)*(exp(t)-1))^2);

% testing nodes
ntime_test=50;
time_test=linspace(0, T, ntime_test);

nspace_test=20;
space_test=linspace(a,b, nspace_test);

X_test=[];
for i=1:ntime_test
    for j=1:nspace_test
        X_test=[X_test,[space_test(j),time_test(i)]'];
    end
end

nspace_test0=100;
space_test0=linspace(a,b, nspace_test0);

X_test0=[];
for i=1:nspace_test0
    X_test0=[X_test0,[space_test0(i),0]'];
end

% construction of omega and theta
omega = cell(1, N_layer + 1); % array di matrici dei pesi
theta = cell(1, N_layer + 1); % array di vettori di bias

% Inizializzazione dei pesi con Xavier
for i = 1:N_layer+1
    fileNameOmega = ['omega', num2str(i), '.xlsx'];
    omega{i} = readmatrix(fileNameOmega);
    fileNameTheta = ['theta', num2str(i), '.xlsx'];
    theta{i} = readmatrix(fileNameTheta);
end

error=GCLossFunction(N_layer,omega,theta,X_test,X_test0,T,a,b,y0,ya,dya);
disp("error= "+ error);
```

```
% weights update
[errornew,thetanew,omeganew]=GCWeightsUpdate(N_layer,omega,theta, ...
                        X_test,X_test0,y0,ya,dya,a,b,T,lrate);
disp("errornew= "+errornew);
if errornew<error
    error=errornew;
    for i = 1:N_layer+1
        fileNameOmega = ['omega', num2str(i), '.xlsx'];
        writematrix(omeganew{i}, fileNameOmega);
        fileNameTheta = ['theta', num2str(i), '.xlsx'];
        writematrix(thetanew{i}, fileNameTheta);
    end
    omega=omeganew;
    theta=thetanew;
end


% calculation of the outputs
u=cell(1,length(X_test0(1,:)));
for i=1:length(X_test0(1,:))
    X=GCNeuralNetwork(N_layer,omega,theta,X_test0(:,i));
    u{i}=X{N_layer+1};
end


y=[];
for i=1:nspace_test0
    x=X_test0(1,i);
    y=[y,ya(0)+(b-x)*(x-a)/(b-a)^2*u{i}];
end

sol=[];
for i=1:nspace_test0
    x=X_test0(1,i);
    sol=[sol,y0(x)];
end

figure;
plot(X_test0, y, '-r','DisplayName', 'Approximated solution');
hold on;
plot(X_test0, sol, '-b','DisplayName', 'exact solution');
xlabel('x (space)');
ylabel('y(x,0)');
title('Graph of y(t)');
legend;


% Comparison of the graph with the exact solution to the bound
Y=[];
for j=1:ntime_test
    for i=1:nspace_test
        xtest=[space_test(i),time_test(j)]';
        X=GCNeuralNetwork(N_layer, omega, theta, xtest);
```

```
        x=space_test(i);
        t=time_test(j);
        Y(i,j)=ya(t)+(b-x)*(x-a)/(b-a)^2*X{N_layer+1};
    end
end

Y=Y';
figure;
[X,T]=meshgrid(space_test, time_test);
surf(X,T,Y);
xlabel('x (space)');
ylabel('t (time)');
zlabel('u(x,t)');
```

Bibliography

[1] Borzì A., Modelling with Ordinary Differential Equations A Comprehensive Approach, CRC Press.

[2] Evans C. L., Partial Differential Equations, American Mathematical Society.

[3] Faronius H. K., Solving Partial Differential Equations With Neural Networks, https://uu.diva-portal.org/smash/record.jsf?pid=diva2%3A1746454&dswid=424 .

[4] Friedlander F. G., Joshi M., Introduction to the Theory of Distributions, Cambridge University Press.

[5] Goodfellow I., Bengio Y., Courville A., Deep Learning, The MIT Press.

[6] Han J., Kamber M., Pei J., Data Mining: Concepts and Techniques, Morgan Kaufmann.

[7] Mascali G., Note di Fisica Matematica Avanzata due.

[8] Perko L., Differential Equations and Dynamical Systems, Springer

[9] Pinkus A., Approximation theory of the MLP model in neural networks, Cambridge University Press.

[10] Shewchuk J. R., An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf .

[11] Van der Meer R., Oosterlee C., Borovykh A., (2021), Optimally weighted loss functions for solving PDEs with Neural Networks, Journal of Computational and Applied Mathematics, https://arxiv.org/abs/2002.06269 .

[12] Van der Meer R., Solving Partial Differential Equations with Neural Networks.

[13] Vostrecov, B.A.K.M.A., Approximation of continuous functions by superpositions of plane waves (1961), http://resolver.tudelft.nl/uuid:c77e1bcc-7212-4234-af34-6586b628ab1c .

[14] Yadav N., Yadav A., Kumar M., An Introduction to Neural Network Methods for Differential Equations, Springer.