Politecnico di Milano
School of Industrial and Information Engineering
Master of Science in Aeronautical Engineering

Physics-Informed Neural Networks for Shallow Water Equations

Advisor:
Prof. Edie Miglio

Author:
Riccardo Anelli
ID 787052

October 6, 2022

Abstract

In recent years, a class of methods that incorporate Partial Differential Equations (PDEs) into a Neural Network (NN) emerged, these methods are commonly referred to as Physics-Informed Neural Networks (PINNs). Embedding the laws of physics into the learning process of a NN allows to combine the NN's ability to extract patterns, automatically, from large volumes of data, with the theoretical knowledge accumulated in scientific theories. The purpose of this work is to investigate the possibility of using the PINNs to approximate the Shallow Water Equations (SWE), a system of hyperbolic PDEs that simulates free-surface flow problems. A wide variety of benchmark problems, selected for both steady state solutions and Riemann problems of increasing complexity, are examined. Moreover, parametric cases are considered for one-dimensional and two-dimensional transient problems, for the purpose of testing the degree to which the PINNs are a convenient tool to be used when dealing with many-query problems. The extensive PINN application showed remarkably results in terms of accuracy of the numerical solutions produced, regardless of the the presence of viscosity or shock waves. Despite the challenge of a high computational cost, this tool proved to be markedly promising when dealing with many-query problems.

Keywords: Partial Differential Equations, Physics-Informed Neural Networks, Shallow Water Equations, steady state solutions, Riemann problems, many-query problems.

Abstract in lingua italiana

Negli ultimi anni è emersa una classe di metodi che inserisce le equazioni differenziali alle derivate parziali nelle reti neurali, questi metodi sono comunemente denominati Physics-Informed Neural Networks (PINNs). Introdurre le leggi della fisica nel processo di training di una rete neurale permette di combinare la capacità delle reti neurali di estrarre modelli, automaticamente, da una grande quantità di dati, con la conoscenza teorica dei fenomeni naturali. Lo scopo di questo lavoro è analizzare la possibilità di utilizzo dei PINN per approssimare le equazioni Shallow Water, ossia, un sistema di equazioni differenziali alle derivate parziali iperboliche utilizzato in modelli geofluidodinamici e idraulici. Viene esaminata un'ampia varietà di problemi di riferimento, scelti con difficoltà crescente sia per problemi con soluzioni stazionarie che per problemi di Riemann. I problemi non stazionari, sia monodimensionali che bidimensionali, vengono successivamente dotati di un'aggiuntivo spazio dei parametri, con lo scopo di testare l'efficacia dei PINN verso i problemi many-query. La vasta applicazione dei PINN ha prodotto risultati di notevole precisione, senza alcuna dipendenza dalla presenza di viscosità o di onde d'urto. Nonostante la difficoltà rappresentata da un costo computazionale elevato, questo metodo si è dimostrato oltremodo promettente nell'applicazione ai problemi many-query.

Parole chiave: equazioni differenziali alle derivate parziali, Physics-Informed Neural Networks, equazioni Shallow Water, soluzioni stazionarie, problemi di Riemann, problemi many-query.

It only ends once. Anything that happens before that is just progress.

—Jacob

Contents

Introduction

The massive increase of data and computing resources available made possible, over the last 15 years, a significant growth in the field of machine learning—particularly, in deep learning. The term *deep learning* generally refers to neural network methods. These methods are able to extract patterns and models automatically from large volumes of data, and are generally agnostic to the underlying scientific principles driving the variables, thus earning the name of *black box* models. Capitalizing on the neural networks' potential to be universal function approximators [16], this technology yielded state-of-the-art results in disciplines of all sorts [22] such as image recognition [19], natural language processing [12] and cognitive sciences [20].

However, in scientific problems the variables can interact in complex nonstationary and nonlinear ways, and the available dataset is often limited: this makes considerably difficult the challenge of achieving good performances with data-hungry methods. With a small dataset, the neural network may learn relationships that fit good *that* dataset, but do not perform well outside of it. Therefore, black box methods are likely to fail when are applied in scientific problems [21,27].

Considering that there are two possible sources of information in any scientific problem, i.e., scientific knowledge or data, two extremes can be identified in the spectrum of the conceivable approaches: theory-based and data science models [17]. This dichotomy is shown in Fig. 1. Theory-based models (cyan rectangle) make extensive use of scientific theories, consequently, they are well-suited for representing processes that are well understood. For this reason, these methods suffer from certain weakness when applied in problems that are *not* completely understood—e.g., the turbulence. On the other side of the range there are the data science models (magenta rectangle): they require a large amount of available data, while

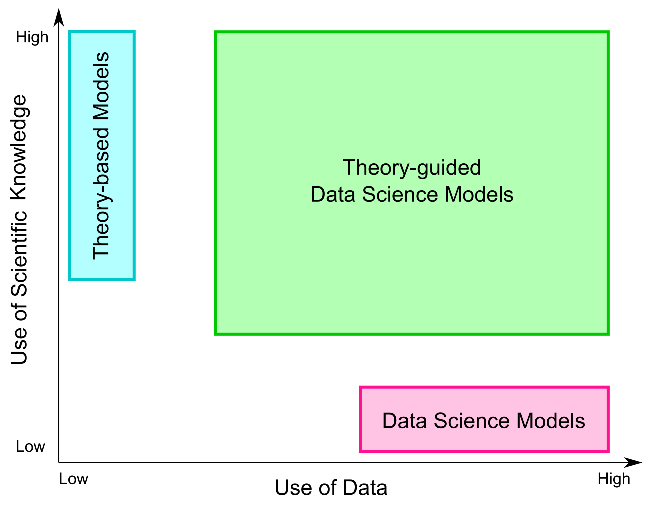


Figure 1: *Theory-based vs. Data Science Models (from Ref. [17]).*

essentially ignoring the theoretical background behind the process. These methods have a wide range of applicability in problems where a plentiful supply of data is available, therefore, they are susceptible to fail where this abundance of data lacks. There's a third approach, the theory-guided data science models (green rectangle), that make effective use of both the sources of information at an equal level. These methods can automatically extract patterns from data *while* making use of the theoretical knowledge accumulated in scientific theories.

Recently, an algorithm that incorporates Partial Differential Equations (PDEs) into a neural network came out, this method is commonly referred to as Physics-Informed Neural Network (PINN). The fundamental step of this approach is embedding a PDE into the loss of the neural network: by constraining the neural network to minimize the PDE residual, the space of admissible solutions the neural network can identify is shrunk to the physical ones.

In this work, the feasibility and the effectiveness of this method are investigated through the solution of problems described by the Shallow Water Equations (SWE), a system of hyperbolic PDEs that simulates free-surface flow problems. There is a wide variety of situations, of environmental interest, that can be mathematically represented by this model, such as flow in open channels and rivers, tsunamis, even urban floods [30].

Numerous applications of the SWE include a margin of uncertainty over some parameters' values and, consequently, these problems have to be solved several times for every parameter's value guess: they are called *many-query* problems. Other applications may involve *optimization* problems over models containing physical or geometrical parameters: every optimization task requires multiple runs of the given problem and, every single time that the problem is recalled, the parameters' values are different.

The PINN seems to be a promising tool to deal with this class of problems for one fundamental reason: following one single training session, the PINN is able to *instantly* provide simulations for every requested parameter value. Consequently, the computational cost of the complete parameters space analysis is, basically, the one of a single training session.

The purpose of this thesis is to examine the degree to which the PINN is actually a convenient tool to be used—whether it is on parametric tasks or not—and the inviscid SWE have been selected as a representative prototype for the hyperbolic class of problems.

The work is organized as follows. In Chapter 1 the mathematical background of the physical problems studied in this thesis is covered. First, the expression of the shallow water equations, the hypotheses they are based upon, the variables involved, and the modeling of the forcing terms are detailed. Then, the analysis of the scaling for a given problem described by the shallow water model is provided.

In Chapter 2 the description of the fundamental notions in machine learning are covered. The supervised learning technology is introduced. The definition of artificial neural network, and the explanation of its operating principles are presented. The various differentiation systems, with a focus on the automatic differentiation techniques, are explained. Finally, an overview of the free software available for neural network implementation is presented.

In Chapter 3 the differential problems, and the way they can be solved by NNs are described. First, the partial differential equations, their classification and solution possibilities are presented. Then, the algorithm that makes the NNs capable of effectively solve the problems described by partial differential equations is summarized. Next, the analysis of the sources of error caused by the use of NNs in solving differential equations is provided. Finally, the core of the Python code implemented throughout this work of thesis is tested and validated, on three different classes of problem.

In Chapter 4 the numerical solutions produced by the PINN tool are presented. The work is introduced starting from the one-dimensional steady state cases, proceeding with the transient problems, ending with the two-dimensional transient one. Moreover, parametric problems are built upon the existing ones.

In Appendix A the main structure of all the source codes is described.

Chapter 1

Mathematical model

1.1 Introduction

The Navier–Stokes equations are a comprehensive model that can describe the motion of a three-dimensional *real* fluid. This elaborate approach is appropriate for the simulation of flows dominated by three-dimensional effects, e.g., in strongly meandering rivers. Since the computational cost of a three-dimensional model simulation is still very high, it makes sense to reduce the model for calculations with simpler flow conditions: the depth-averaged two-dimensional flow equations, also called *Shallow Water Equations* (SWE), provide a suitable approximation to model free-surface flow problems.

In spite of the name, the fluid doesn't have to be *water* (for example, the weather forecasting can be done by applying these same equations), indeed, the SWE can be modified in order to simulate different moving fluids, and they can be applied to many free-surface flow scenarios such as:

- tsunamis prediction;
- flash floods prediction;
- atmospheric flows;
- flows around structures;
- planetary flows.

This chapter covers the mathematical background that the physical problems studied in this work of thesis have in common. The expression of the shallow water equations, the hypotheses they are based upon, the variables involved, and the modeling of the forcing terms are the topics detailed in Section 1.2. Then, Section 1.3 provides the analysis of the scaling for a given problem described by the shallow water model.

1.2 Shallow Water Equations (SWE)

The SWE consist in a system of nonlinear PDEs that model the two-dimensional incompressible flow. The derivation of these equation is obtained, mainly, by averaging the Navier–Stokes equations over the depth, and is based on the following *hypotheses*:

- the fluid is incompressible and its density is uniform;
- the vertical dimension is much smaller than the horizontal scale;
- the vertical velocity is zero;
- the vertical dynamics is neglected;
- the pressure distribution is hydrostatic.

Within the ambit of this thesis the equations are written in conservative form, in the Cartesian coordinate system, following the notation illustrated in Fig. 1.1 on page 14. The coordinate z defines the vertical direction, to which the free surface elevation is associated.

The SWE are first implemented in Python holding as unknowns the water height $h(x,y,t)$ and the two horizontal components of the depth-averaged velocity $u(x,y,t)$ and $v(x,y,t)$. This set of variables can be used to define a new array of unknowns called *flows* and defined as $q_x = hu$ and $q_y = hv$. Then, the new unknowns set made up of the water height $h(x,y,t)$ and of the two flows $q_x(x,y,t)$ and $q_y(x,y,t)$ can be used to write the SWE in a second form.

A general expression of the 2D SWE, holding as unknowns h, u and v, that takes into account several complex terms like rain, infiltration and viscous terms, is given in Eq. (1.1) for the sake of completeness. Indeed,

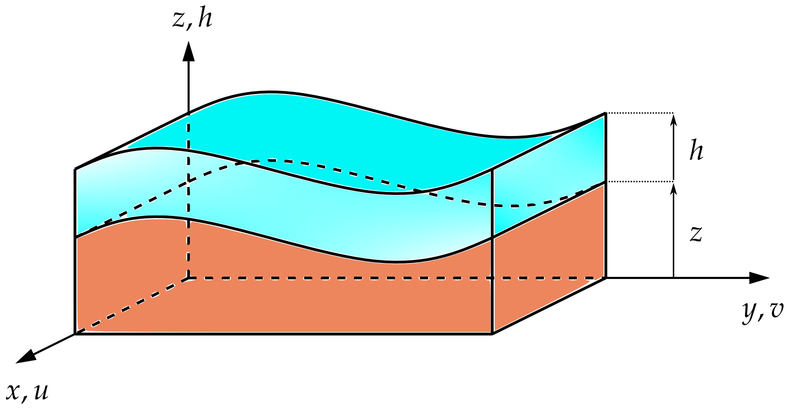


Figure 1.1: *2D SWE. Notation.*

different source terms can be included in the right-hand side of the shallow water system of equations, depending on the application.

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial (hu)}{\partial x} + \dfrac{\partial (hv)}{\partial y} = R - I \\ \dfrac{\partial (hu)}{\partial t} + \dfrac{\partial (hu^2 + \frac{1}{2}gh^2)}{\partial x} + \dfrac{\partial (huv)}{\partial y} = gh(S_{0_x} - S_{f_x}) + \mu S_{d_x} \\ \dfrac{\partial (hv)}{\partial t} + \dfrac{\partial (huv)}{\partial x} + \dfrac{\partial (hv^2 + \frac{1}{2}gh^2)}{\partial y} = gh(S_{0_y} - S_{f_y}) + \mu S_{d_y} \end{cases} \tag{1.1}$$

where:

- the first equation is a mass balance and the other two equations are the momentum balances;
- $g = 9.81\ m/s^2$ is the standard gravitational acceleration;
- $R \geq 0$ is the rain intensity, its physical dimensions are [L/T]. It's a given function of time and space that will be always set to zero in the test cases studied in this thesis;
- I is the infiltration rate, [L/T]. The expression of this term can be given by the Green-Ampt model [14] and won't be taken into account in this work;

- S_{0_x} and S_{0_y}, dimensionless quantities, are the opposite of the slopes in the x, y directions. The topography z could be variable over the time in applications that take into account the *erosion*; in this thesis it will be considered constant in time. Consequently, these two terms are defined as:

$$S_{0_x} = -\frac{\partial z(x,y)}{\partial x}, \; S_{0_y} = -\frac{\partial z(x,y)}{\partial y} \tag{1.2}$$

- S_{f_x} and S_{f_y}, dimensionless quantities, are the terms that model the friction between water and ground. The friction laws behind these terms are based on experiential evidences and can take two different shapes. The first expression is based on the Manning-Strickler's model:

$$S_{f_x} = n^2 \frac{u|\mathbf{u}|}{h^{4/3}} = n^2 \frac{q_x|\mathbf{q}|}{h^{10/3}} \tag{1.3}$$

$$S_{f_y} = n^2 \frac{v|\mathbf{u}|}{h^{4/3}} = n^2 \frac{q_y|\mathbf{q}|}{h^{10/3}} \tag{1.4}$$

where n is the Manning's coefficient, $\mathbf{u} = [u, v]^T$ is the velocity vector and $\mathbf{q} = [q_x, q_y]^T$ is the flow vector. This model doesn't work for applications that allow for dry zones. This issue is solved using the Darcy-Weisbach's model. The expressions of the Darcy-Weisbach's laws are:

$$S_{f_x} = C_f \frac{u|\mathbf{u}|}{h} = C_f \frac{q_x|\mathbf{q}|}{h^3} \tag{1.5}$$

$$S_{f_y} = C_f \frac{v|\mathbf{u}|}{h} = C_f \frac{q_y|\mathbf{q}|}{h^3} \tag{1.6}$$

with $C_f = f/(8g) = 1/C^2$, where f is a dimensionless coefficient and C is the Chézy's friction coefficient;

- μS_{d_x} and μS_{d_y} are the viscous terms. They can be modeled by the Laplace operator and μ is the kinematic viscosity of the fluid. No viscous term will be taken into account in this work.

The conservative variables, derived with respect to the time, and the components of the flux tensor, can be conveniently organized into vectors:

$$\mathbf{U} = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, \ \mathbf{E} = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}, \ \mathbf{G} = \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix} \tag{1.7}$$

In the same way, all the source terms in the right-hand side of the system can be organized into a vector:

$$\mathbf{S} = \begin{bmatrix} R - I \\ gh(S_{0_x} - S_{f_x}) + \mu S_{d_x} \\ gh(S_{0_y} - S_{f_y}) + \mu S_{d_y} \end{bmatrix} \tag{1.8}$$

Writing the SWE adopting these vectors results in a single, compact, vector equation:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{E}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = \mathbf{S} \tag{1.9}$$

1.2.1 Boundary conditions

The applications of the SWE that allow for Laplacian terms require the boundary conditions to be set in every border of the differential problem, along with the initial condition in case of transitory problems. On the other hand, if the SWE are applied to the *perfect* fluid model and if the condition $h > 0$ is met, the problem is hyperbolic and the boundary conditions treatment must comply with a suitable set of requisites explained right away.

Let $\mathbf{n}$ be the outward unit normal vector on the border of the computational domain, so that:

- the condition $\mathbf{u} \cdot \mathbf{n} > 0$ denotes an *outflow* point on the border, the fluid exits the domain through an outflow point;
- the condition $\mathbf{u} \cdot \mathbf{n} < 0$ denotes an *inflow* point on the border, the fluid enters the domain through an inflow point.

The flows described by the SWE for a perfect fluid can be classified according to the value of a dimensionless number, the Froude number (Fr), defined as:

$$\mathrm{Fr} = \frac{|\mathbf{u}|}{\sqrt{gh}} \tag{1.10}$$

- if $\mathrm{Fr} < 1$ the flow is known to be *subcritical*, the information can travel upstream as well as downstream;
- if $\mathrm{Fr} > 1$ the flow is known to be *supercritical*, the information cannot travel upstream;
- if $\mathrm{Fr} = 1$ the flow is known to be *critical*, the upstream propagating information remain stationary.

As explained in [11], the combination between the type of border and the value of the Froude number can generate four possible situations on every point of the border:

- if $\mathrm{Fr} > 1$ and $\mathbf{u} \cdot \mathbf{n} < 0$, three conditions have to be set;
- if $\mathrm{Fr} \leq 1$ and $\mathbf{u} \cdot \mathbf{n} < 0$, two conditions have to be set;
- if $\mathrm{Fr} < 1$ and $\mathbf{u} \cdot \mathbf{n} > 0$, one condition has to be set;
- if $\mathrm{Fr} \geq 1$ and $\mathbf{u} \cdot \mathbf{n} > 0$, no conditions can be set.

1.3 Scaling of the SWE

The problems examined throughout this work are often defined in big spatial domains, and develop within large time scales. Furthermore, some of these problems involve infinitesimal solutions. These scenarios conflict with the technology used to solve each case study, since the NN best deals with both domains and solutions belonging to the $\mathcal{O}(1)$ scale. The effective strategy that can get every single problem to fit with the solver is the *scaling*.

The standard practice begins with the identification of suitable reference values for the geometry of the problem, for its temporal domain and for the variables, specifically **L**, **T**, **H**, **U**. The optimal outcome of this process would be a scaled problem defined in an unitary geometrical domain, lasting one second, involving solutions with both range and upper limit unitary. The original SWE in Eq. (1.1) are scaled into a new system of equation that is tagged with the *tilde* symbol. It is relevant pointing out that initial and boundary data have to be scaled as well, but here only the equations are considered for the sake of simplicity.

The relations between the original quantities and the scaled ones are:

- $x = \mathbf{L}\,\tilde{x}$
- $y = \mathbf{L}\,\tilde{y}$
- $z = \mathbf{H}\,\tilde{z}$
- $t = \mathbf{T}\,\tilde{t}$
- $h = \mathbf{H}\,\tilde{h}$
- $u = \mathbf{U}\,\tilde{u}$
- $v = \mathbf{U}\,\tilde{v}$

Substituting in Eq. (1.1) and making explicit the right-hand side terms:

$$\begin{cases} \dfrac{\mathbf{H}}{\mathbf{T}}\dfrac{\partial \tilde{h}}{\partial \tilde{t}} + \dfrac{\mathbf{HU}}{\mathbf{L}}\dfrac{\partial(\tilde{h}\tilde{u})}{\partial \tilde{x}} + \dfrac{\mathbf{HU}}{\mathbf{L}}\dfrac{\partial(\tilde{h}\tilde{v})}{\partial \tilde{y}} = 0 \\[2ex] \dfrac{\mathbf{HU}}{\mathbf{T}}\dfrac{\partial(\tilde{h}\tilde{u})}{\partial \tilde{t}} + \dfrac{\mathbf{HU}^2}{\mathbf{L}}\dfrac{\partial(\tilde{h}\tilde{u}^2 + \frac{1}{2}\tilde{g}\tilde{h}^2)}{\partial \tilde{x}} + \dfrac{\mathbf{HU}^2}{\mathbf{L}}\dfrac{\partial(\tilde{h}\tilde{u}\tilde{v})}{\partial \tilde{y}} = \\ = -\dfrac{\mathbf{U}^2}{\mathbf{H}}\mathbf{H}\dfrac{\mathbf{H}}{\mathbf{L}}\tilde{g}\tilde{h}\dfrac{\partial \tilde{z}}{\partial \tilde{x}} - \dfrac{\mathbf{U}^2}{\mathbf{H}}\mathbf{U}^2\tilde{g}C_f\tilde{u}|\tilde{\mathbf{u}}| \\[2ex] \dfrac{\mathbf{HU}}{\mathbf{T}}\dfrac{\partial(\tilde{h}\tilde{v})}{\partial \tilde{t}} + \dfrac{\mathbf{HU}^2}{\mathbf{L}}\dfrac{\partial(\tilde{h}\tilde{u}\tilde{v})}{\partial \tilde{x}} + \dfrac{\mathbf{HU}^2}{\mathbf{L}}\dfrac{\partial(\tilde{h}\tilde{v}^2 + \frac{1}{2}\tilde{g}\tilde{h}^2)}{\partial \tilde{y}} = \\ = -\dfrac{\mathbf{U}^2}{\mathbf{H}}\mathbf{H}\dfrac{\mathbf{H}}{\mathbf{L}}\tilde{g}\tilde{h}\dfrac{\partial \tilde{z}}{\partial \tilde{y}} - \dfrac{\mathbf{U}^2}{\mathbf{H}}\mathbf{U}^2\tilde{g}C_f\tilde{v}|\tilde{\mathbf{u}}| \end{cases} \tag{1.11}$$

where $\tilde{g} = \frac{\mathbf{H}}{\mathbf{U}^2}g$. After simplification we obtain:

$$\begin{cases} \dfrac{\mathbf{L}}{\mathbf{TU}}\dfrac{\partial \tilde{h}}{\partial \tilde{t}} + \dfrac{\partial(\tilde{h}\tilde{u})}{\partial \tilde{x}} + \dfrac{\partial(\tilde{h}\tilde{v})}{\partial \tilde{y}} = 0 \\ \\ \dfrac{\mathbf{L}}{\mathbf{TU}}\dfrac{\partial(\tilde{h}\tilde{u})}{\partial \tilde{t}} + \dfrac{\partial(\tilde{h}\tilde{u}^2 + \frac{1}{2}\tilde{g}\tilde{h}^2)}{\partial \tilde{x}} + \dfrac{\partial(\tilde{h}\tilde{u}\tilde{v})}{\partial \tilde{y}} = \\ = -\tilde{g}\tilde{h}\dfrac{\partial \tilde{z}}{\partial \tilde{x}} - \dfrac{\mathbf{LU^2}}{\mathbf{H^2}}\tilde{g}C_f\tilde{u}|\tilde{\mathbf{u}}| \\ \\ \dfrac{\mathbf{L}}{\mathbf{TU}}\dfrac{\partial(\tilde{h}\tilde{v})}{\partial \tilde{t}} + \dfrac{\partial(\tilde{h}\tilde{u}\tilde{v})}{\partial \tilde{x}} + \dfrac{\partial(\tilde{h}\tilde{v}^2 + \frac{1}{2}\tilde{g}\tilde{h}^2)}{\partial \tilde{y}} = \\ = -\tilde{g}\tilde{h}\dfrac{\partial \tilde{z}}{\partial \tilde{y}} - \dfrac{\mathbf{LU^2}}{\mathbf{H^2}}\tilde{g}C_f\tilde{v}|\tilde{\mathbf{u}}| \end{cases} \tag{1.12}$$

If:

$$\mathbf{U} = \frac{\mathbf{L}}{\mathbf{T}} \quad \text{and} \quad \mathbf{H} = \sqrt{\frac{\mathbf{L^3}}{\mathbf{T^2}}} \tag{1.13}$$

then:

$$\frac{\mathbf{L}}{\mathbf{TU}} = 1 \quad \text{and} \quad \frac{\mathbf{LU^2}}{\mathbf{H^2}} = 1 \tag{1.14}$$

and the mathematical expressions used for the scaled problem—made up of equations, boundary conditions and initial conditions—and for the original unscaled one are *identical*. Actually, the possibility of taking advantage of this convenient choice for the reference values depends on the problem at hand. Since the unit values of the scaling groups in Eq. (1.14) are obtained enforcing the relations in Eq. (1.13), not *all* the reference values **L**, **T**, **H**, **U** can be chosen arbitrarily. So:

- if this choice of reference values allows to end up with a scaled problem defined in a $\mathcal{O}(1)$ domain, for $\mathcal{O}(1)$ variables, it's ok;
- if this choice of reference values brings to a scaled problem with variables values too far from the $\mathcal{O}(1)$ scale, then the best option would be choosing arbitrarily and suitably *all* the reference values, ending up with a scaled mathematical expression different from the original one.

Chapter 2

Machine Learning

2.1 Introduction

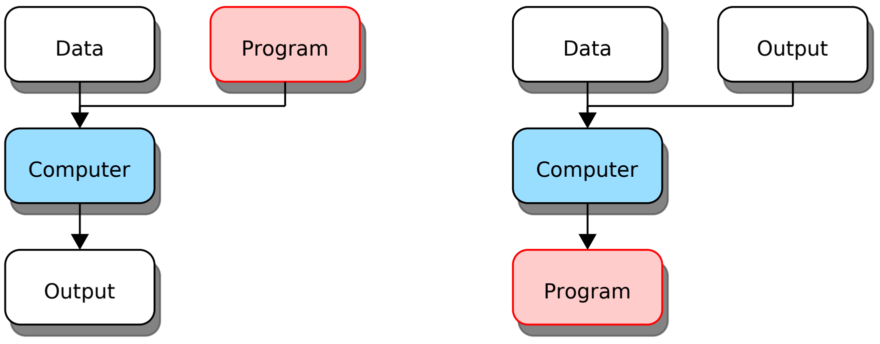
Machine Learning (ML) is the scientific discipline that the American pioneer Arthur Samuel defined, back in 1959, as the field of study that gives computers the ability to learn without being explicitly programmed. It comes under the broad umbrella of Artificial Intelligence (AI). Despite a terminology that suggests an ability to think by the machines, there are no magic human-like tools on the table: the process behind every machine learning based design, as remarkable as it may be, is nothing more than the elaboration of a computation.

As described in [15], the traditional programming procedure requires a program—a *code*—to be written; it's successively processed by the computer together with the data and the output is provided. The machine learning approach is different: the computer gets the output and the data, then it provides the program. This distinction is depicted in Fig. 2.1.

The whole ML discipline can be divided into three main fields of study:

- supervised learning
- unsupervised learning
- reinforcement learning

This thesis will cover only supervised learning tasks (see [4, 10] for an overview of the unsupervised and reinforcement learning techniques).



(a) *Traditional programming* **(b)** *Machine learning*

Figure 2.1: *Introduction. Comparison between approaches.*

The main approach that will be examined and used in order to deal with this kind of problems will be the Artificial Neural Network (ANN).

This chapter covers the description of the fundamental notions in machine learning. The supervised learning technology is introduced in Section 2.2. The definition of artificial neural network and the explanation of its operating principles are presented in Section 2.3. The various differentiation systems, with a focus on the automatic differentiation techniques, are explained in Section 2.4. Finally, an overview of the free software available for neural network purposes is presented in Section 2.5.

2.2 Supervised learning

The data-set that enters a supervised learning algorithm is a collection of N labeled examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$. Each one of these elements comes in the form of a row vector made up of an n-dimensional vector $\mathbf{x}$ called *feature vector* and of another element, the *label* y_i, appended to the previous vector. As suggested by its denomination, the feature vector $\mathbf{x}$ is made up of elements called *features*, they are denoted as f_j for $j = 1, \ldots, n$. A classic example of labeled element is illustrated in Fig. 2.2.

The vector $\mathbf{x}$ can be represented as a *datapoint*, i.e. a point in the n-dimensional space where the axes are the features: this goes to show that the dimensionality of a supervised learning problem matches the

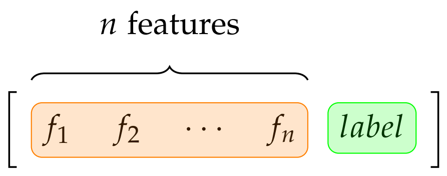


Figure 2.2: *Supervised learning. Labeled example.*

number of features. The nature of these features can be different, so are the properties they can have:

- **numerical features**. This class of features consist of pure numerical values. There are two properties that these kind of data have. First, they can be put in order: 1 is before 5, 5 is before 25 and so forth. Second, calculations can be performed with these values.
- **ordinal features**. These data come apparently in the form of *first*, *second*, and so on. The algorithm dealing with this data can take advantage of the first property but not of the second one.
- **categorical features**. Here there is only the names of the data—for instance, information like *colours*—and nothing can be deduced from them. A machine learning algorithm can't be given pure categorical data as they are, so they must be properly converted. The conversion technique is called *one-hot encoding*. While this transformation increases the dimensionality of the problem, yet it allows the algorithm to manage categorical data. For further details about the one-hot encoding, see [34].

The aim of a supervised learning algorithm is to produce a program—a *model*—able to predict the label y of a given feature vector $\mathbf{x}$. For instance, a model created using a data-set of text messages could take as input an incoming e-mail and output information that allows deducing whether or not it is an unwanted commercial bulk email. The whole supervised learning branch is commonly broke down in two categories:

- **classification**. This is the problem of automatically assigning the label to an unlabeled example. For instance, recognising a dog in an image can be seen as classifying the image in one of the two classes: 'has a dog' or 'does not have a dog'. The label is a member of a finite set of classes. If the size of this set is two, the problem is referred to as *binary classification*; if it's larger than two, then it's a *multiclass classification* problem.

- **regression**. This is the problem of assigning a real-valued label to an unlabeled example, this specific kind of label is called *target*. For instance, the problem of estimating the house price based on an array of house features—such as the location, the measurement of the surfaces or the number of bedrooms—is an example of regression. This kind of problems are solved by regression learning algorithms: they take in a collection of labeled examples and build a model that can take an unlabeled example as input and output the target.

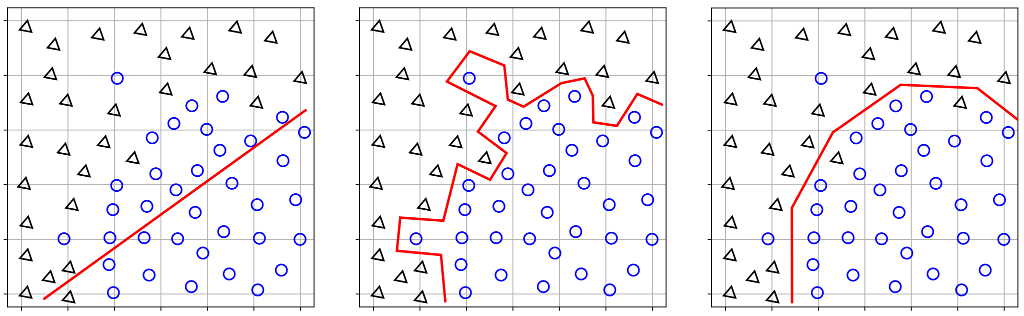
2.2.1 Classification

Since this thesis will focus on the *classification* learning branch only, the most classical approaches of this technology are introduced in this section.

The challenge of a classification problem is to produce an accurate classifier. The typical machine learning approach—considering a binary classification case—is to supply the algorithm with k labeled examples of one class together with h labeled examples of another class, then to let it determine whether a brand-new *unlabeled* example is more similar to the k-type labeled examples or the h-type ones.

The classification performed by the algorithm can be seen as a boundary in the n-dimensional space that divides the datapoints according to what the algorithm is asked to do, e.g., if it's a binary classification task, the boundary will conveniently divide the whole set of points into two groups—the k-type point one side of the boundary, the h-type the other side. This boundary, the object that partitions the workspace, is called *hyperplane*. Once the hyperplane is set, the type prediction for the brand-new element provided to the algorithm is automatic: the side of the boundary the new element falls in the n-dimensional space is all the algorithm needs to look in order to assign the label.

A suitable selection of hyperplanes is shown in Fig. 2.3 in order to point out some typical issues that have to be taken into account:



(a) *Underfitting* **(b)** *Overfitting* **(c)** *Good fit*

Figure 2.3: *Classification. Fitting issues,* ***(a)****-****(b)****, good fit case,* ***(c)****.*

- the hyperplane **(a)** is not capable of separating effectively the two groups of elements, it *underfits* the data. The reason behind this problem can be twofold: the excessive simplicity of the algorithm compared to the characteristics of the data-set, or the scarcity of information brought by the data-set.

- the hyperplane **(b)** splits perfectly the elements, it divides the blue circles from the black triangles without any error, yet it actually *overfits* the data. This is a problem, because the algorithm has learned to predict perfectly a data-set that is not perfect, i.e. it has learned to predict imperfections along with the fundamental information brought by the data-set.

- the hyperplane **(c)** is a reasonable separator. It is not strictly correct, yet it is in excellent accordance with the prevalent arrangement of the data.

The goal of the training process is to build an excellent separator, but perfection is not on the table: it's the trend of the elements that are actually relevant that the algorithm is after. A hundred percent accurate hyperplane for the whole training set turns out to be be too intricate, and eventually wrong, because it would perfectly fit a set of data that inevitably carry some random oscillations. Looking for perfection when operating with imperfect data it's ineffective, there must be a trade-off between the complexity of the hyperplane and the accuracy of the algorithm.

The ability to build the hyperplane is *learned* by the algorithm through a process that takes the name of *training*, at this point the algorithm doesn't give any output. The operation of generating the label to the brand-new unlabeled elements takes place in the following phase that is called the *predicting* phase.

2.3 Artificial neural networks

The artificial neural network is a supervised learning algorithm. The feedforward neural network (FNN), the simplest neural network architecture, applies linear and nonlinear transformations to the inputs. As the name suggests, the neural network is a framework made up of basic elements, called *neurons*, that actively execute in the network by taking in input data, by processing it, and by producing an output. The two elements that constitute each neuron—the bias and the activation function—and the details of the neuron's operations are defined and described right away.

The neurons are arranged in *layers*. The information enters the neural network by way of the input-layer neurons, then it moves towards the output-layer neurons passing through the hidden-layers neurons. The number of the *input-layer* neurons matches the number of the features in the input vector, while the number of the *output-layer* neurons matches the dimensionality of the label. A neural network consisting of more than one hidden layer is called deep neural network; the subfield of machine learning with deep neural networks is called *deep learning*.

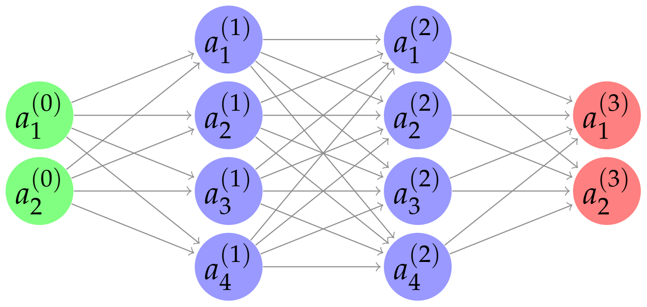


Figure 2.4: *Example of deep neural network.*

A typical neural network architecture is illustrated in Fig. 2.4. Every neuron is labeled with a superscript indicating the layer, starting from number 0 for the input layer, then number 1 for the following layer and so forth. A subscript indicates which neuron of that layer it is. Every neuron belonging to one layer is connected with all the neurons belonging to the previous layer, as well as with all the neurons belonging to the following layer. Neurons belonging to the same layer are not interconnected.

The elements which are not given as inputs are the *parameters*. A first set of parameters—they're real numbers, they're called *weights*—is associated with each and every neuron-to-neuron connection existing in the neural network. A second set of parameters is associated with the neurons belonging to any layer other than the input one, they are real numbers as well and they are called *biases*.

Weights and biases play a crucial role. They are both randomly set by the algorithm at the start of the process, then they are properly modified in order to make the network work its optimization problem out: the task of finding an optimal set of weights and biases is performed through a *training* process.

2.3.1 Operation and representation

This section is devoted to the elaboration of the network's mode of operation. While all the calculus will be gradually broken down, a schematic representation of an isolated neuron is shown in Fig. 2.5.

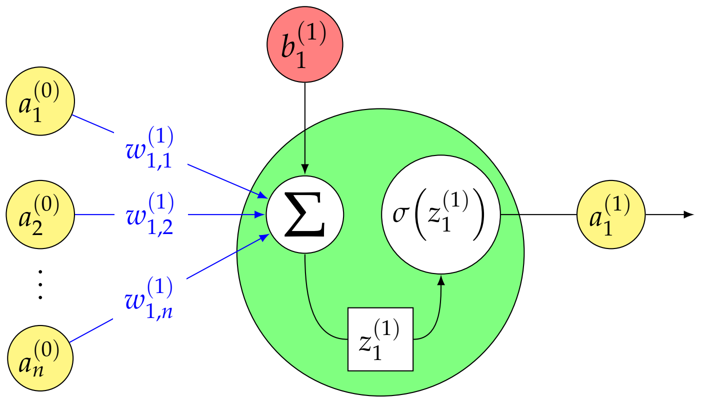


Figure 2.5: *Neuron $a_1^{(1)}$. Inner components, operations, input neurons.*

A machine learning algorithm accepts a collection of data and output: in the neural network case the data are the features, and they are fed to the input-layer neurons; the output are the labels, and they are compared with the output-layer neurons. The operating principle of a neural network begins with the feeding of the data into the input-layer neurons: each of those neurons get one numerical value. The information contained in every input-layer neuron propagates through the neural network according to a basic principle: the values contained in the neurons belonging to one layer determine the values contained in the neurons of the next layer.

The first step to get the value for one neuron belonging to the layer next to the input layer, is computing the weighted sum of all the input layer neurons values according to the weights, plus the bias:

$$\begin{aligned} z_1^{(1)} &= w_{1,1}^{(1)}\, a_1^{(0)} + w_{1,2}^{(1)}\, a_2^{(0)} + \cdots + w_{1,n}^{(1)}\, a_n^{(0)} + b_1^{(1)} \\ &= \sum_{k=1}^{n} w_{1,k}^{(1)}\, a_k^{(0)} + b_1^{(1)} \end{aligned} \tag{2.1}$$

where:

- $z_1^{(1)}$ is the *logit*, a number still to be handled in order to get a valid neuron value, it's associated with the first hidden layer's neuron;
- $a_k^{(0)}$ is the value stored in the k-th input layer's neuron, it's called *activation*;
- n is the number of neurons belonging to the input layer;
- $w_{1,k}^{(1)}$ is the weight associated to the connection between $a_k^{(0)}$ and the first hidden layer's neuron;
- $b_1^{(1)}$ is the bias associated with the first hidden layer's neuron.

This is just for one neuron; every other neuron in the same layer is connected to all the neurons of the previous layer and has its own weights, plus one bias, associated. Such a sophisticated computational procedure can be written in a more compact way: the weights are organized into a matrix where each row corresponds to the connections between one layer and a particular neuron in the next layer; the biases are organized into a column as a vector, so are all the neurons values from every layer. The number of neurons belonging to the layer next to the input layer is m.

$$\begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ \vdots \\ z_m^{(1)} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & \dots & w_{1,n}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & \dots & w_{2,n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(1)} & w_{m,2}^{(1)} & \dots & w_{m,n}^{(1)} \end{bmatrix} \begin{bmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_m^{(1)} \end{bmatrix} \tag{2.2}$$

At this point, a nonlinear transformation is performed on this vector by means of a nonlinear function, called *activation function*, that will transform the weighted sum into the actual value stored in the neuron—the *activation*. The application of the nonlinearity to the logit will determine the intensity a particular neuron will be activated with, if at all. This will enable the network to learn complex mapping functions. Three popular functions that do this are the *rectified linear unit* (ReLU), the *sigmoid function* (S) and the *hyperbolic tangent*:

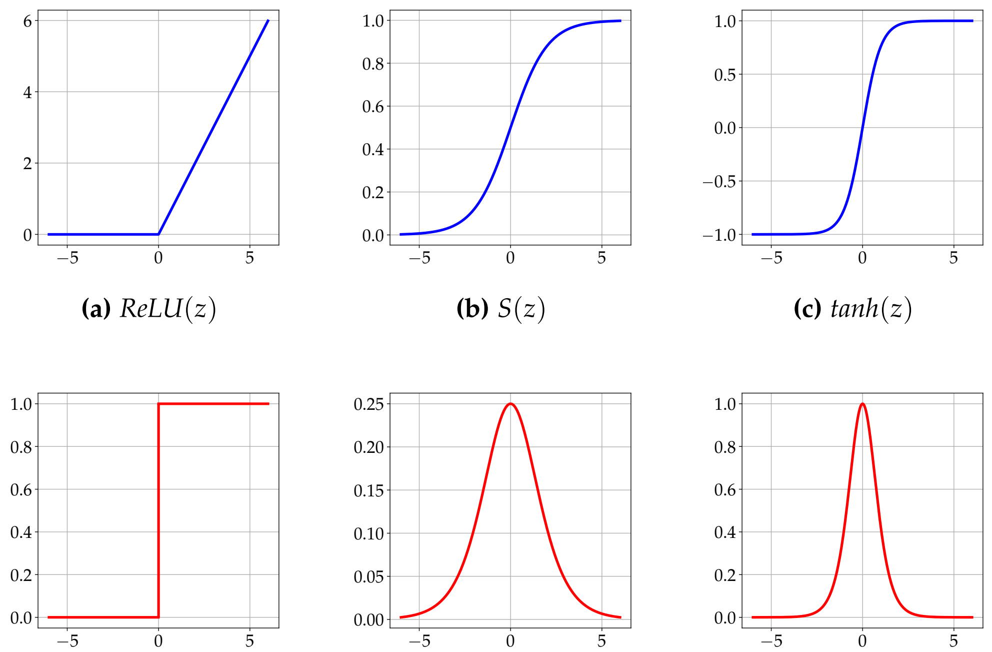
$$\mathrm{ReLU}(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases} \tag{2.3}$$

$$\mathrm{S}(z) = \frac{1}{1+e^{-z}} \tag{2.4}$$

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.5}$$

These functions, along with their derivative, are plotted in Fig. 2.6 on page 29. Several factors have an influence on the choice of the appropriate activation function, such as the properties of the function, the properties of the function's derivative or the structure of the neural network.

Another element that influence this choice is the type of classification problem the network is supposed to learn: whether it is a binary classification task or a multi-class one, the activation function that will lead to the better result may be different [10].



(a) $ReLU(z)$ **(b)** $S(z)$ **(c)** $tanh(z)$

(d) $step(z)$ **(e)** $\frac{d}{dz}S(z)$ **(f)** $\frac{d}{dz}tanh(z)$

Figure 2.6: *Activation functions. Three popular functions, **(a)-(b)-(c)**, corresponding derivatives, **(d)-(e)-(f)**.*

Starting with a thorough analysis of the plots, some observation can be made:

- **ReLU**. This function is a simple computation that returns the logit provided as input directly, or the value zero, and the neuron does not activate, if the input is zero or less. This computation results in a sparse network and a faster data processing. Another major advantage of this function over the $\mathrm{S}(z)$ and the $\tanh(z)$, that require the use of an exponential calculation, is that the ReLU is easy to implement, requiring just a *max* function.

  On the other hand, the limitations come from its gradient, that is zero whenever the neuron is not active. This can bring about problems in gradient-based optimization algorithm, since they will not adjust the weights of a neuron that never activates initially. Some extensions to the ReLU overcome this problem, e.g., the Leaky ReLU, that modifies the function to allow small negative values when the input is less than zero in order to avoid the case of *dead* neurons [10].

- **sigmoid function**. This function is smooth and continuously differentiable, very negative inputs end up close to 0 and very positive inputs end up close to 1. It outputs only positive signals. Since the derivative is smooth and continuously differentiable as well, gradient-based optimization algorithm will benefit from that.

  The disadvantage is that while the derivative is very high when the logit values between -3 and 3, it becomes flat away from that and the training algorithm comes to a halt: this is the *vanishing gradient problem*.

- **hyperbolic tangent**. This function is smooth and continuously differentiable as well as the $S(z)$, yet it becomes flat more quickly and it can return both positive and negative signals. The derivative is smooth but it is steeper than the derivative of $S(z)$, so the hyperbolic tangent function also has the vanishing gradient problem.

Different layers may have different activation functions, but all neurons of the same layer apply the same nonlinearity to its logits. Once the activation function set is selected for the particular application—often by testing different functions and eliminating the ones that do not work—a final step in the mathematical representation can be made, as the nonlinear activation function is wrapped around the whole matrix framework. The function is applied element-wisely to the m-dimensional resulting vector. For the sake of simplicity, only the symbol σ is used to identify the activation function.

$$\begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & \dots & w_{1,n}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & \dots & w_{2,n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(1)} & w_{m,2}^{(1)} & \dots & w_{m,n}^{(1)} \end{bmatrix} \begin{bmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_m^{(1)} \end{bmatrix} \right) \tag{2.6}$$

The algorithm iterates this information propagation procedure up to the output layer of the neural network. A schematic representation of an isolated neuron is shown in Fig. 2.5 on page 26.

Summarizing the whole process, let $\mathcal{N}^L(\mathbf{x}) : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ be a L-layer neural network, meaning it's a $(L-1)$-hidden layer neural network: there are $\mathcal{N}_0 = d_{in}$ neurons in the input layer, $\mathcal{N}_L = d_{out}$ neurons in the output layer, $\mathcal{N}_\ell$ neurons in the ℓ-th layer. Let $\mathbf{W}^\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ and $\mathbf{b}^\ell \in \mathbb{R}^{N_\ell}$ be the weight matrix and the bias vector in the ℓ-th layer, respectively. The FNN is defined as follows:

$$\begin{array}{ll} \text{input layer:} & \mathcal{N}^0(\mathbf{x}) = \mathbf{x} \in \mathbb{R}^{d_{in}} \\ \text{hidden layers:} & \mathcal{N}^\ell(\mathbf{x}) = \sigma(\mathbf{W}^\ell \mathcal{N}^{\ell-1}(\mathbf{x}) + \mathbf{b}^\ell) \in \mathbb{R}^{N_\ell}, \quad \text{for } 1 \le \ell \le L-1 \\ \text{output layer:} & \mathcal{N}^L(\mathbf{x}) = \mathbf{W}^L \mathcal{N}^{L-1}(\mathbf{x}) + \mathbf{b}^L \in \mathbb{R}^{d_{out}} \end{array}$$

2.3.2 Loss function

In order to get the neural network to operate effectively, a training process is mandatory. The concept is to measure the error the network commits and then modify its parameters set to improve its perfomance and make this error very small; this is done through an algorithm called *backpropagation*, which happens to be the reverse of the forward pass described above.

The prerequisite is a whole bunch of training data, i.e. a set of feature vectors to provide as input of the network along with labels for what every feature vector is supposed to be. The goal of the training process is to generalize the behaviour of the network beyond the training data, to new *unlabeled* input vectors.

The pattern of the output layer neurons values is compared with the desired pattern for that specific input vector—the desired pattern is exactly the label, suitably made up of the same number of elements of the output layer. What is done is define a *loss* function $\mathcal{L}$ as a way to measure the dissimilarity between the desired output layer pattern and the actual one, hence, this is a way to quantify how good or bad the neural network is performing. The loss of a single training element $\mathcal{L}_h$ is defined by adding up the squares of the differences between each of those output layer neurons values and the desired values:

$$\mathcal{L}_h = \sum_{k=1}^{n} (a_k - y_k)^2 \tag{2.7}$$

where:

- n is the number of neurons belonging to the output layer;
- a_k is the activation of the k-th output layer's neuron;
- y_k is the desired value for a_k.

The procedure is repeated for all the training elements at disposal, then the average loss over the whole training set at disposal is considered. Hence the loss $\mathcal{L}$, the end result of the computation, is a function that takes in all the weights and biases and it returns one number describing how suitable the weights and biases are.

$$\mathcal{L} = \frac{1}{N} \sum_{h=1}^{N} \left(\sum_{k=1}^{n} (a_k - y_k)^2 \right)_h \tag{2.8}$$

where N is the number of the whole bunch of training samples.

The loss is small when the network operates correctly but, as might be expected, the network is going to perform badly on the firsts cycles of training inputs since it's working with randomly initialized parameters. The scalar information provided from the loss function is used in order to properly modify these parameters. The *aim* is to find the value for all these parameters that minimizes the loss and, moreover, every single tweak to the parameters is made with the purpose of causing the fastest decrease to the loss. This process is called *learning*.

2.3.3 Gradient descent

The *gradient* is a mathematical operator that provides the direction of the steepest ascent of a function, consequently, the *negative* gradient of the loss function gives the direction that decreases this function most quickly starting from the point the gradient is computed. Naturally enough, the algorithm for minimizing the loss function computes its negative gradient, it takes a small step in that direction afterwards and then repeats the same procedure over and over.

The input of the loss function is a large set of variables. There are many possible local minimums that the gradient-based algorithm might land in depending on which random input it starts, and there is no guarantee that the local minimum it gets to is going to be the absolute *global* minimum of the loss function. It's important to restate that the loss function involves an average over all the training data, therefore, minimizing it yields to a better performance on all the samples.

All the weights and the biases can be thought as scalars orderly arranged in a vector $\mathbf{W}$, the gradient of the loss function $\boldsymbol{\nabla}\mathcal{L}(\mathbf{W})$ is a vector the same size of its input $\mathbf{W}$. The algorithm will modify every single weight and every single bias by a quantity proportional to the corresponding value in the negative gradient vector:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \boldsymbol{\nabla}\mathcal{L}(\mathbf{W}) \tag{2.9}$$

where η is a constant called *learning rate*, a coefficient that controls the step size towards the local minimum. The process of repeatedly adjust the input of the loss function by a multiple of its negative gradient is called *gradient descent*. If the learning rate η is very small the convergence will take more time, but this will keep the minimization process under control in order to avoid overshooting. On the other hand, an high value of η might cause the algorithm to overshoot and to miss the local minimum: find the best way to pick the step size is still an open research problem. This issue is depicted in Fig. 2.7 on page 34.

2.3.4 Stochastic gradient descent

In practice, computing a gradient descent step based on the whole training dataset, for modern *large scale* machine learning systems, can be an incredibly difficult task.

Let $\{(x_1, y_1), \dots, (x_n, y_n)\} \in \mathbb{R}^{n \times d} \times \mathbb{R}^n$ be the whole training dataset, where n is the number of all the training data points and d is the dimension of each input sample: in large scale machine learning systems they can both be large. The number of training data points, these days, could be in the order of millions; moreover, the dimensionality of the feature vectors

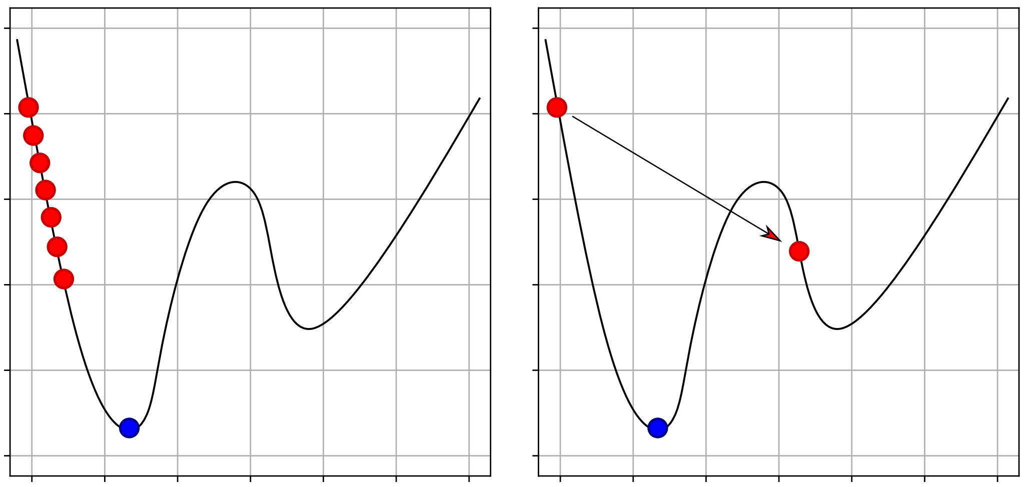

(a) *Very small learning rate* **(b)** *Large learning rate*

Figure 2.7: *Gradient descent. Effect of the learning rate value on the convergence to a local minimum.*

can also be pretty large, e.g., if x is a megapixel image then d is a million as well. This indicates that computing the gradient at a single point involves computing the gradient of the entire sum over the whole training samples in Eq. (2.8), and that sum is huge. As a result, the computation of a *single* gradient to do a *single* step of gradient descent could take hours or days: that is a major drawback so this is not what is commonly done.

As made clear in [36], one first technique that makes sense for optimizing this machine learning problem is to compute the gradient based on just a single randomly-chosen training sample, i.e. at iteration k an integer $i(k)$ is randomly picked:

$$i(k) \in \{1, 2, \dots, n\} \tag{2.10}$$

The randomness used is, as might be expected, the one *without* replacement: the algorithm takes the dataset, it runs a pre-shuffle operation and then just streams through the data. The update of the parameters set is performed using the loss produced by the i-th training sample alone:

$$\mathbf{W}_{k+1} \leftarrow \mathbf{W}_k - \eta_k \nabla \mathcal{L}_{i(k)}(\mathbf{W}_k) \tag{2.11}$$

The gradient iteration of one individual component is now n times faster. This technique is referred to as *Stochastic Gradient Descent* (SGD).

Instead of making a clear descent at every step like the gradient descent does—after all, the gradient descent is called gradient *descent*, at every step it descends, it decreases the loss function—the SGD doesn't do any descent at every step: sometimes it goes up, sometimes it goes down, still making progress towards the optimum.

Moreover, the SGD process is much more sensitive to the step size η_k than the method based on the full gradient is. This problem emerges when the process approaches the optimum: the greater the step size is, the wilder the fluctuations are. At the beginning of the SGD iterations, instead, high η_k values do not limit the stability of the process, that therefore can make quick initial progress. That is a very typical behaviour of SGD: the user may see the training loss decrease very fast in the beginning but this favorable performance declines afterwards, just as soon as the process gets closer to the optimum.

Because of the high variance that such process may have, then rather than pick one random training sample at the every given k-th iteration, the algorithm picks a mini-batch I_k of samples, so that:

$$\mathbf{W}_{k+1} \leftarrow \mathbf{W}_k - \frac{\eta_k}{|I_k|} \sum_{h \in I_k} \boldsymbol{\nabla} \mathcal{L}_h(\mathbf{W}_k) \tag{2.12}$$

This will average things and reduce the variance of the process. A mini-batch of size 1 is the SGD explained earlier, a mini-batch of size n is the pure gradient descent, something in between is what libraries actually use.

2.3.5 Backpropagation

The backpropagation is a very famous algorithm that computes every single gradient for deep network training. For the purpose of getting the idea of the calculus underlying this procedure, let $a_j^{(L)}$ be the activation of the j-th output layer's neuron, then let $a_k^{(L-1)}$ be the activation of the k-th last hidden layer's neuron, so that:

$$\begin{aligned} z_j^{(L)} &= w_{j,k}^{(L)} a_k^{(L-1)} + b_j^{(L)} \\ a_j^{(L)} &= \sigma(z_j^{(L)}) \end{aligned} \tag{2.13}$$

The desired value for $a_j^{(L)}$, for a given training example, is y_j. The loss for a single non-specific training example is $\mathcal{L}_h$, defined as:

$$\mathcal{L}_h = \sum_{j=1}^{n_L} (a_j^{(L)} - y_j)^2 \tag{2.14}$$

where n_L is the number of neurons in the output layer. The components of the vector $\nabla \mathcal{L}_h$, i.e. the partial derivatives of $\mathcal{L}_h$ with respect to all the weight and biases in the network are in the form:

$$\begin{aligned} \frac{\partial \mathcal{L}_h}{\partial w_{j,k}^{(L)}} &= \frac{\partial \mathcal{L}_h}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{j,k}^{(L)}} \\ &= 2 \sum_{j=1}^{n_L} (a_j^{(L)} - y_j)\, \sigma'(z_j^{(L)})\, a_k^{(L-1)} \end{aligned} \tag{2.15}$$

$$\begin{aligned} \frac{\partial \mathcal{L}_h}{\partial b_j^{(L)}} &= \frac{\partial \mathcal{L}_h}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} \\ &= 2 \sum_{j=1}^{n_L} (a_j^{(L)} - y_j)\, \sigma'(z_j^{(L)})\, 1 \end{aligned} \tag{2.16}$$

with $a_k^{(L-1)}$ that depends in turn by weights an biases going back up to the input layer; the algorithm is not able to directly influence the neurons' activations.

Taking SGD into consideration, the loss function $\mathcal{L}_{SGD}$ generated by one iteration is the average over all the losses $\mathcal{L}_h$ across the samples belonging to each mini-batch, then its derivative requires the same procedure, e.g., for the Eq. (2.15) it becomes:

$$\frac{\partial \mathcal{L}_{SGD}}{\partial w_{j,k}^{(L)}} = \frac{1}{|I_k|} \sum_{h \in I_k} \frac{\partial \mathcal{L}_h}{\partial w_{j,k}^{(L)}} \tag{2.17}$$

where I_k is the k-th mini-batch. This same chain rule idea can be iterated backwards to compute the partial derivatives of the loss function with respect to all the previous weights and biases.

2.4 Automatic differentiation

Differentiation shows up everywhere in science and engineering, as well as in this thesis: from the gradient-based optimization algorithm, the backpropagation, to the the technology that will be introduced in the next chapter, the *physics-informed neural networks*, where the computation of the derivatives of the networks' outputs with respect to the inputs is performed. The possible methods for computing the derivatives are four:

1. analytical *hand-coded* derivative;
2. finite difference or other numerical approximations;
3. symbolic differentiation;
4. automatic differentiation.

Following, all the possible approaches to the evaluation of a function derivative are discussed, with a strong attention on the most advanced technique available today for neural network computing—the Automatic Differentiation (AD). An overview scheme of all these techniques is shown in Fig. 2.8 on page 38.

2.4.1 Analytical derivative

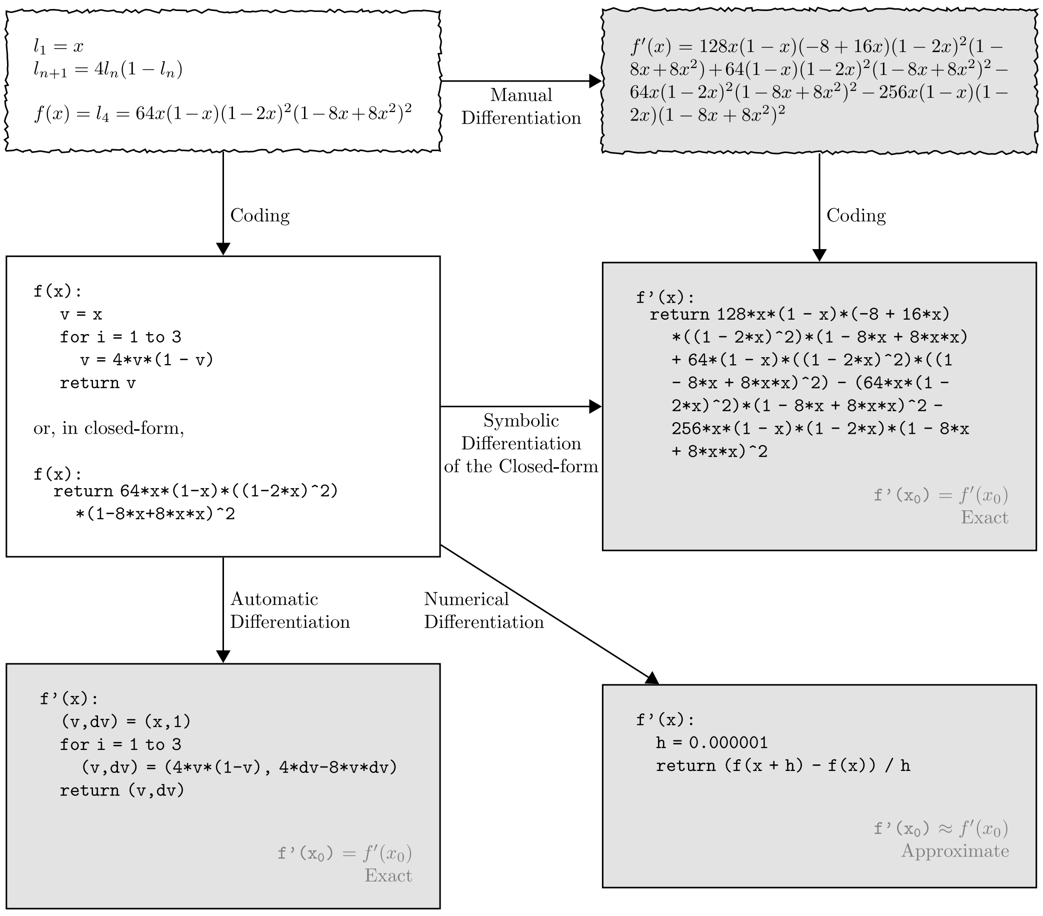
If the function to be derived depends on a small set of variables, if it's not made up of a series of highly nonlinear functions and if it's not highly composed, then the *manual* derivative can be performed using the basic derivative rules. Here is a short list of well-known derivative rules:

Power rule

$$\frac{d}{dx}x^n = nx^{n-1} \tag{2.18}$$

Sum and difference rule

$$\frac{d}{dx}\Big(f(x) \pm g(x)\Big) = \frac{d}{dx}f(x) \pm \frac{d}{dx}g(x) \tag{2.19}$$



$l_1 = x$
$l_{n+1} = 4l_n(1-l_n)$

$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$

$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$

```
f(x):
   v = x
   for i = 1 to 3
     v = 4*v*(1 - v)
   return v
```

or, in closed-form,

```
f(x):
  return 64*x*(1-x)*((1-2*x)^2)
    *(1-8*x+8*x*x)^2
```

```
f'(x):
  return 128*x*(1 - x)*(-8 + 16*x)
    *((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
    + 64*(1 - x)*((1 - 2*x)^2)*((1
    - 8*x + 8*x*x)^2) - (64*x*(1 -
    2*x)^2)*(1 - 8*x + 8*x*x)^2 -
    256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
    + 8*x*x)^2
```

f'(x_0) $= f'(x_0)$
Exact

```
f'(x):
  (v,dv) = (x,1)
  for i = 1 to 3
    (v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
  return (v,dv)
```

f'(x_0) $= f'(x_0)$
Exact

```
f'(x):
  h = 0.000001
  return (f(x + h) - f(x)) / h
```

f'(x_0) $\approx f'(x_0)$
Approximate

Figure 2.8: *Differentiation. Possible approaches (from Ref. [1]).*

Product rule

$$\frac{d}{dx}\Big(f(x)\cdot g(x)\Big) = \frac{d}{dx}f(x)\cdot g(x) + f(x)\cdot\frac{d}{dx}g(x) \tag{2.20}$$

This method returns the exact derivative, however, the procedure may become almost impossible, and subjected to errors hard to debug, for complicated functions.

2.4.2 Numerical differentiation

The second approach is a *numerical* differentiation method like the finite difference. The method follows from the limit definition of the derivative: it applies a small perturbation to a given function, then computes an approximation of its derivative whose accuracy depends on the magnitude of the perturbation.

Let $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ be a continuous scalar function defined in the n-dimensional space, its partial derivative with respect to x_i, evaluated in the point $\mathbf{x}_0$, can be approximated by a forward finite difference scheme as:

$$\left.\frac{\partial f(\mathbf{x})}{\partial x_i}\right|_{\mathbf{x}_0} \approx \frac{f(\mathbf{x}_0 + h\mathbf{e}_i) - f(\mathbf{x}_0)}{h} \tag{2.21}$$

where $\mathbf{e}_i$ is the i-th unit vector along the i-th axis, and h is the step size. A higher accuracy can be obtained by running a center finite difference scheme:

$$\left.\frac{\partial f(\mathbf{x})}{\partial x_i}\right|_{\mathbf{x}_0} \approx \frac{f(\mathbf{x}_0 + h\mathbf{e}_i) - f(\mathbf{x}_0 - h\mathbf{e}_i)}{2h} \tag{2.22}$$

While these scheme can be quite simple to implement, some issues may come up with accuracy and numerical stability. As the step size h get closer to zero, the truncation error decreases but the computer will face floating point errors and returns an incorrect result. Then again, if h is too large the rounding error will be small but accuracy of the approximation will be sub-standard. An instance of trade-off between truncation rounding is illustrated in Fig. 2.9.

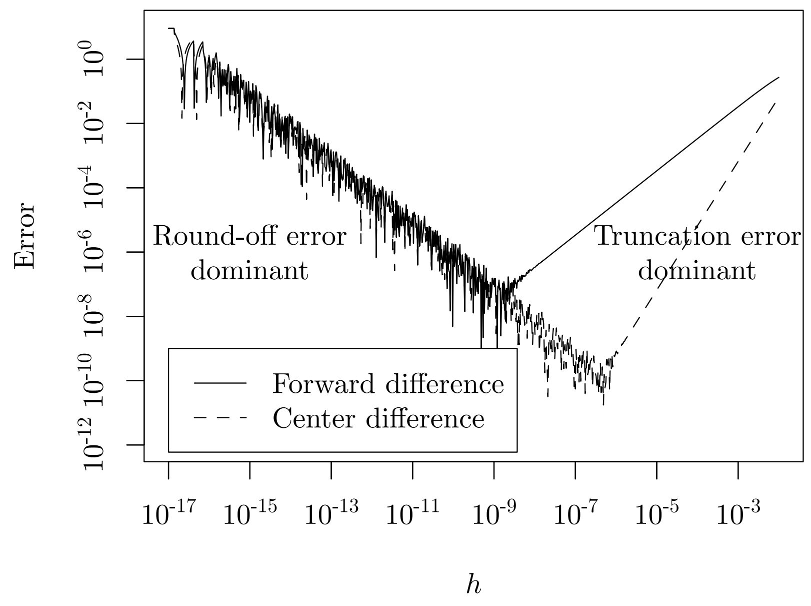


Figure 2.9: *Numerical differentiation. Error trend for the derivative of the logistic map given in Eq. (2.26), for $n = 4$, at $x_0 = 0.2$ (from Ref. [1]).*

2.4.3 Symbolic differentiation

The third strategy is the *symbolic* differentiation, that is basically an automated version of the manual differentiation and is used in software programs such as Mathematica, Maxima, and Maple. Indeed, just like the the manual differentiation, it works by breaking apart a complex expression into a bunch of simpler expressions by using the standard derivative rules.

This method provides the exact computation of the derivatives (up to numerical precision) since it bypasses the sources of error present in numerical differentiation, still it comes with a major issue: even though the method works towards a simplification of the expressions, the length of the resulting function (the derivative) can be exponentially larger than the original one and the evaluation of considerable large expression can be prohibitively slow. This problem is known as *expression swell*. It happens because some derivative rule, like the product rule, intrinsically lead to duplicated computation:

$$\begin{aligned} h(x) &= f(x)\, g(x) \\ h'(x) &= f'(x)\, g(x) + f(x)\, g'(x) \end{aligned} \tag{2.23}$$

If $f(x)$ in turn involves a product of function

$$f(x) = u(x)\ v(x) \tag{2.24}$$

then the $h'(x)$ expression's length would escalate

$$h'(x) = \Big(u'(x)\ v(x) + u(x)\ v'(x)\Big)\ g(x) + u(x)\ v(x)\ g'(x) \tag{2.25}$$

The measure of this problem becomes remarkably apparent by applying the symbolic differentiation to a recurrence relation known as *logistic map*:

$$l_{n+1} = 4l_n(1 - l_n),\ l_1 = x \tag{2.26}$$

For values of n like 1 and 2 the derivative expression is essentially as simple as the original, but as the n increases the derivative expression's length quickly gets beyond control, as made clear in Table 2.1. Sometimes the expression can be simplified in a shorter polynomial form, but this isn't always possible.

Table 2.1: *Symbolic differentiation. Expression swell, illustrated for the derivatives of the logistic map.*

n	l_n	$\frac{d}{dx}l_n$
1	x	1
2	$4x(1-x)$	$4(1-x) - 4x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$
4	$64x(1-x)(1-2x)^2 (1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2) + 64(1-x)(1-2x)^2(1-8x+8x^2)^2 - 64x(1-2x)^2(1-8x+8x^2)^2 - 256x(1-x)(1-2x)(1-8x+8x^2)^2$

2.4.4 Automatic differentiation

The automatic differentiation is one of the most useful and efficient techniques in scientific computing. This method has been developed as a way to compute derivatives with the same accuracy as symbolic differentiation, that is the *exact* answer up to the numerical precision, but rather than producing an expression for a derivative, the only output of the process is the *numerical value*. The two main algorithms for AD are the *forward mode* and the *reverse mode*.

Like finite differences, AD involves only the evaluation of the function, moreover, it bypasses the inefficiency of the symbolic differentiation by taking advantage of the intermediate variables that make up the original function's expression. These intermediate variables build the global function by way of primitive operations whose corresponding derivative rules are known.

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a differentiable function with n input variables x_i and m output variables y_j:

- **forward mode AD** takes one single pass to define all the intermediate variables v_i and the corresponding partial derivatives with respect to one input variable, e.g., $\frac{\partial v_i}{\partial x_1}$. It takes n forward AD evaluations to compute the full Jacobian of a function, for this reason, forward mode AD is recommended for functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ where $n \ll m$.

- **reverse mode AD** takes one single pass to compute all the partial derivatives of one output variable with respect to all the input variables, e.g., $\frac{\partial y_1}{\partial x_i}$. It takes m reverse AD evaluations to compute the full Jacobian of a function, for this reason, reverse mode AD is recommended for functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ where $n \gg m$. That it the case for the ANN, where the input are the weights whose amount is considerable large: the backpropagation algorithm takes one single reverse AD pass to compute all the gradients for every output.

As a way to make the concepts clear, let

$$y = f(x_1, x_2) = x_1 + e^{x_2} + \sin(x_1 x_2) \tag{2.27}$$

be a function that will serve as an example, the two partial derivatives of y are evaluated at $(x_1, x_2) = (4, 1)$. The full elaboration of the two forward iterations is given in Tables 2.2 and 2.3.

Table 2.2: *AD. Forward mode iteration for the Eq. (2.27) at $(x_1, x_2) = (4, 1)$. Left, intermediate variables. Right, partial derivatives with respect to the input x_1.*

v_1	$= x_1$	$= 4$	$\dot{v}_1$	$= \dot{x}_1$	$= 1$
v_2	$= x_2$	$= 1$	$\dot{v}_2$	$= \dot{x}_2$	$= 0$
v_3	$= e^{v_2}$	$= 2.718$	$\dot{v}_3$	$= \dot{v}_2 \, e^{v_2}$	$= 0$
v_4	$= v_1 v_2$	$= 4$	$\dot{v}_4$	$= \dot{v}_1 v_2 + v_1 \dot{v}_2$	$= 1$
v_5	$= \sin v_4$	$= -0.757$	$\dot{v}_5$	$= \dot{v}_4 \cos v_4$	$= -0.654$
v_6	$= v_1 + v_3 + v_5$	$= 5.961$	$\dot{v}_6$	$= \dot{v}_1 + \dot{v}_3 + \dot{v}_5$	$= 0.346$
y	$= v_6$	$= \mathbf{5.961}$	$\dot{y}$	$= \dot{v}_6$	$= \mathbf{0.346}$

Table 2.3: *AD. Forward mode iteration for the Eq. (2.27) at $(x_1, x_2) = (4, 1)$. Left, intermediate variables. Right, partial derivatives with respect to the input x_2.*

v_1	$= x_1$	$= 4$	$\dot{v}_1$	$= \dot{x}_1$	$= 0$
v_2	$= x_2$	$= 1$	$\dot{v}_2$	$= \dot{x}_2$	$= 1$
v_3	$= e^{v_2}$	$= 2.718$	$\dot{v}_3$	$= \dot{v}_2 \, e^{v_2}$	$= 2.718$
v_4	$= v_1 v_2$	$= 4$	$\dot{v}_4$	$= \dot{v}_1 v_2 + v_1 \dot{v}_2$	$= 4$
v_5	$= \sin v_4$	$= -0.757$	$\dot{v}_5$	$= \dot{v}_4 \cos v_4$	$= -2.615$
v_6	$= v_1 + v_3 + v_5$	$= 5.961$	$\dot{v}_6$	$= \dot{v}_1 + \dot{v}_3 + \dot{v}_5$	$= 0.104$
y	$= v_6$	$= \mathbf{5.961}$	$\dot{y}$	$= \dot{v}_6$	$= \mathbf{0.104}$

In order to compute the gradients with the reverse AD algorithm, still a forward AD pass is required. It's a two-part process: first, one forward pass computes all the outputs, then, one reverse pass computes all the gradients. While the forward AD pass complements every intermediate variable with the corresponding partial derivative, the reverse AD pass complements every intermediate variable with the corresponding *adjoint*:

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i} \tag{2.28}$$

The derivatives will be propagates backwards, according to the dependencies, by means of the adjoints. The full elaboration of this two-steps process is given in Table 2.4.

The models—introduced in the next chapter—that execute the computation of the derivatives of the networks' outputs can take advantage of both the forward and reverse AD algorithms: only *one* forward pass and *one* reverse pass are required to compute the derivatives with respect to *all* the network's weights.

Table 2.4: *AD. Reverse mode iteration for the Eq. (2.27) at $(x_1, x_2) = (4, 1)$. Left, one forward pass computes the intermediate variables. Right, one reverse pass computes the partial derivatives.*

v_1	$= x_1$	$= 4$
v_2	$= x_2$	$= 1$
v_3	$= e^{v_2}$	$= 2.718$
v_4	$= v_1 v_2$	$= 4$
v_5	$= \sin v_4$	$= -0.757$
v_6	$= v_1 + v_3 + v_5$	$= 5.961$
$\boldsymbol{y}$	$= \boldsymbol{v_6}$	$= \mathbf{5.961}$

$\boldsymbol{\bar{v}_1}$	$= \boldsymbol{\frac{\partial y}{\partial x_1}}$	$= \mathbf{0.346}$
$\boldsymbol{\bar{v}_2}$	$= \boldsymbol{\frac{\partial y}{\partial x_2}}$	$= \mathbf{0.104}$
$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1} + \bar{v}_6 \frac{\partial v_6}{\partial v_1}$	$= 0.346$
$\bar{v}_2$	$= \bar{v}_3 \frac{\partial v_3}{\partial v_2} + \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= 0.104$
$\bar{v}_3$	$= \bar{v}_6 \frac{\partial v_6}{\partial v_3}$	$= 1$
$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= -0.654$
$\bar{v}_5$	$= \bar{v}_6 \frac{\partial v_6}{\partial v_5}$	$= 1$
$\bar{v}_6$	$= \bar{y}$	$= 1$

2.5 Available libraries

Several open-source libraries are available and still in active development in the field of deep learning, helping in simplifying the overall programming experience. Three of them can be identified as the most used by data scientists as well as by beginners: TensorFlow, Keras and PyTorch.

- **Keras** is an high-level neural networks library that can run on top of TensorFlow as well as of several other libraries, this is why Keras can be regarded as a complementary option to TensorFlow rather than a rival library. It is written in Python code and best runs on GPUs and TPUs.
- **TensorFlow** is a symbolic math machine learning library developed by the Google Brain team. The term *tensor* refers to the representation of data as multi-dimensional array while the term *flow* refers to the series of operations that the user can performs on tensors. Tensorflow can be used with CPUs, GPUs and TPUs as well.
- **PyTorch** is a machine learning library used for applications such as natural language processing and computer vision, it's largely developed by the Facebook's AI research group. At its core it provides n-dimensional tensors similar to the NumPy ones; it can run on GPUs.

While Keras and Tensorflow are intrinsically interconnected—and PyTorch is an entirely separate alternative—the three libraries have got some differences that distinguish one from another.

Keras is a high-level Python *application programming interface* (API), recognized for its ease of use and syntactic simplicity. Tensorflow provides both high and low level APIs, widely used in Python, which are completely under stable releases. Other language APIs are under development and not in stable releases. Pytorch is a lower-level API, mainly used in Python, focused on direct work with array expressions.

Keras is commonly used for small datasets and its *performance* is relatively slower; on the other hand, Tensorflow and PyTorch provide a similar fast pace which is suitable for high performance models and large datasets that require fast execution.

One major competitive advantage of TensorFlow over the other deep learning libraries lies in its greater *popularity*, that results in a larger community where the users' issues can be addressed quickly. As a matter of fact, all the program codes written within the scope of this thesis use TensorFlow [41].

Chapter 3

Physics-Informed Neural Networks

3.1 Introduction

The Physics-Informed Neural Networks (PINNs) are a specific type of neural networks (NNs) trained to approximate the solution to any given law of physics described, in general, by a partial differential equation (PDE) or by a system of PDEs. This kind of approach allows to use the PDEs in *strong form* directly [8].

There are mainly two challenges, in the contemporary scientific areas of expertise, that can be effectively worked out by the PINN technology. The *first* challenge is about the actual data availability on physical, biological or engineering systems: while other neural network applications like computer vision and natural language processing can rely on a giant amount of data ready for use, the same advantage is not within easy reach for physical systems since the cost of data acquisition for this field of research can be exorbitant. The *second* obstacle concerns the dimensionality, the truncation errors and the numerical quadrature errors brought in by the conventional solving techniques: the traditional mesh-based methods, such as the finite element method, can obtain the approximate solution of a PDE through a mesh built in the computational domain and, since these methods requires the PDE to be written in its integral form, through the approximated solution of integrals.

This chapter is devoted to the description of the differential problems, and to the way they can be solved by NNs. First of all, the partial differential equations, their classification and solution possibilities are presented in Section 3.2. Section 3.3 summarizes the algorithm that makes the NNs capable of effectively solve the problems described by partial differential equations. Section 3.4 provides the analysis of the sources of error caused by the use of NNs in solving differential equations. Finally, the core of the Python code implemented throughout this work of thesis is tested and validated, on three different classes of problem, in Section 3.5.

3.2 Partial differential equations

The differential equations are a fundamental element for many mathematical models of physical phenomena, many everyday devices are designed on the basis of these equations. The partial differential equations are differential equations whose unknown function is derived with respect to multiple variables of the time or the space.

Let $u(\mathbf{x}, t)$ be an unknown function defined in $\Omega \subset \mathbb{R}^d \times (0, T)$, let g be the set of data on which the PDE depends. In this case, the generic PDE is expressed as:

$$\mathcal{P}(u, g) = f\left(\mathbf{x}, t, u, \frac{\partial u}{\partial t}, \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_d}, \ldots, \frac{\partial^{p_1+\cdots+p_d+p_t} u}{\partial x_1^{p_1} \ldots \partial x_d^{p_d} \partial t^{p_t}}, g\right) = 0 \quad (3.1)$$

where $\mathbf{x} = (x_1, \ldots, x_d)^T$, and $p_1, \ldots, p_d, p_t \in \mathbb{N}$. The maximum value of the sum $p_1 + \cdots + p_d + p_t$ is called the *order* of the PDE, it is an integer number and equals the maximum order of the partial derivatives present in the Eq. (3.1). If this equation depends linearly on the unknown u it is called *linear*, otherwise it is called *nonlinear*.

3.2.1 PDE families

The partial differential equations can be organized into three different families: *elliptic*, *parabolic* and *hyperbolic* equations.

This classification is accomplished examining the coefficients of the PDE. For the sake of simplicity, an example of this analysis is carried out for a linear second-order PDE with constant coefficients, written in the form $Lu = G$:

$$Lu = A\frac{\partial^2 u}{\partial x_1^2} + B\frac{\partial^2 u}{\partial x_1 \partial x_2} + C\frac{\partial^2 u}{\partial x_2^2} + D\frac{\partial u}{\partial x_1} + E\frac{\partial u}{\partial x_2} + Fu \tag{3.2}$$

where G is an assigned function, while $A, B, C, D, E, F \in \mathbb{R}$.

Let $\Delta = B^2 - 4AC$ be the *discriminant* of the PDE, the classification is accomplished basing on the sign of the discriminant so that:

if $\Delta < 0$	the PDE is *elliptic*
if $\Delta = 0$	the PDE is *parabolic*
if $\Delta > 0$	the PDE is *hyperbolic*

The numerical validation of the Python code for implementing the PINNs, for each of the three PDE categories, is carried out in Section 3.5.

3.2.2 PDE solution

A given PDE does not completely define a differential problem, since a *well-posed* problem needs the PDE to be supplied with suitable boundary conditions (BC)—that could be Dirichlet, Neumann or Robin boundary conditions—and an initial condition (IC) on the unknown function u. A differential problem is defined well-posed if:

- the solution exists;
- the solution is unique;
- the solution depends *continuously* on the data, i.e. if the data change very little, the solution changes very little.

Anyway, in most cases is not possible to obtain the solution of a PDE in closed form. For this reason, it is very relevant the use of numerical methods that allow to build an approximation u_N of the exact solution u. The PINNs can be successfully applied to problems belonging to all the three classes of PDEs. In Section 3.4 will be showed that a feedforward neural network with *enough* neurons can simultaneously and uniformly approximate any function and its partial derivatives. Despite this, the error estimation for PINNs—and for supervised learning overall—is currently still an open research problem. For further details about the PDEs and the numerical methods for handling them, see Ref. [31].

3.3 The PINN algorithm

Let $u(\mathbf{x})$ be an unknown function defined in the spatio-temporal domain $\Omega \subset \mathbb{R}^d$, let $\boldsymbol{\lambda}$ be the set of data on which the PDE depends. For the sake of simplicity, a second-order differential equation is considered. In this case, the generic PDE is expressed as:

$$\mathcal{P}(u, \boldsymbol{\lambda}) = f\left(\mathbf{x}, \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_d}, \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 u}{\partial x_1 \partial x_d}, \ldots, \boldsymbol{\lambda}\right) = 0 \tag{3.3}$$

where $\mathbf{x} = (x_1, \ldots, x_d)$. Any of the variables x_i, for $i = 1, \ldots, d$, could represent the temporal variable, consequently, the IC can be managed as a Dirichlet BC set on the border $t = 0$. Suitable BC can be given in the following forms:

$$u(\mathbf{x}) - g_D(\mathbf{x}) = 0 \qquad \text{on } \Gamma_D \subset \partial\Omega \tag{3.4}$$

$$\mathbf{n} \cdot \boldsymbol{\nabla}\, u(\mathbf{x}) - g_N(\mathbf{x}) = 0 \qquad \text{on } \Gamma_N \subset \partial\Omega \tag{3.5}$$

$$\alpha u(\mathbf{x}) + \beta \mathbf{n} \cdot \boldsymbol{\nabla}\, u(\mathbf{x}) - g_R(\mathbf{x}) = 0 \qquad \text{on } \Gamma_R \subset \partial\Omega \tag{3.6}$$

The description of the PINN algorithm is now detailed:

Step 1: a neural network $\hat{u}(\mathbf{x}, \boldsymbol{\theta})$ is built as a surrogate of the unknown solution $u(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^d$. The NN's input neurons number matches d, while the NN's output neurons number matches the dimension of u. The set of all the NN's parameters is $\boldsymbol{\theta}$, so that: $\boldsymbol{\theta} = \{\mathbf{W}^\ell, \mathbf{b}^\ell\}_{1 \leq \ell \leq L}$, where L is the number of layers of the NN. The tuning of the *hyperparameters* of the NN (e.g. the network size, the learning rate, the number of training points, the value of the loss' weights) is up to the user, is significantly based on the experience and could strongly affect the performances.

Step 2: two sets of scattered training points $\mathcal{T}_f \subset \Omega$ and $\mathcal{T}_b \subset \partial\Omega$ are defined: these are the locations in the computational domain where $\hat{u}$ will be trained, i.e., $\hat{u}$ is constrained to satisfy the PDE in these points. These points can be either:

- randomly generated by the program
- defined by the user

The whole training points set is defined as $\mathcal{T} = \{\mathcal{T}_f, \mathcal{T}_b\}$.

Step 3: a suitable loss function is defined to make an estimation of the discrepancy between $\hat{u}$ and the constraints, the PDE and the BC/IC, whose residuals are respectively symbolized as $\mathcal{P}(\hat{u}, \boldsymbol{\lambda})$ and $\mathcal{B}(\hat{u}, \mathbf{x})$:

$$\mathcal{L}(\boldsymbol{\theta}, \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}, \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}, \mathcal{T}_b) \tag{3.7}$$

where w_f and w_b are the weights, and:

$$\mathcal{L}_f(\boldsymbol{\theta}, \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{x \in \mathcal{T}_f} \|\mathcal{P}(\hat{u}, \boldsymbol{\lambda})\|_2^2 \tag{3.8}$$

$$\mathcal{L}_b(\boldsymbol{\theta}, \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{x \in \mathcal{T}_b} \|\mathcal{B}(\hat{u}, \mathbf{x})\|_2^2 \tag{3.9}$$

All the derivatives within both the residuals $\mathcal{P}(\hat{u}, \boldsymbol{\lambda})$ and $\mathcal{B}(\hat{u}, \mathbf{x})$ are effectively handled via AD.

Step 4: the loss function $\mathcal{L}(\boldsymbol{\theta}, \mathcal{T})$ is minimized through the *training* of the neural network $\hat{u}$. This procedure will constantly update the parameters set $\boldsymbol{\theta}$ up to an optimal composition $\boldsymbol{\theta}^*$ that will get $\hat{u}$ to behave the desired way.

A diagram of the whole procedure—for the case of the 1D heat equation supplied with mixed BCs—is shown in Fig. 3.1 on page 52.

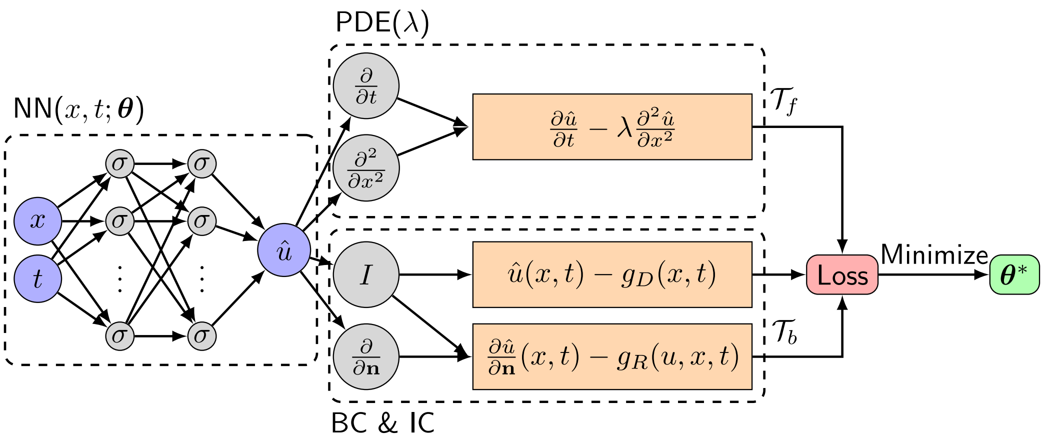

PDE(λ)
NN($x, t; \boldsymbol{\theta}$)
x
t
σ
$\hat{u}$
$\frac{\partial}{\partial t}$
$\frac{\partial^2}{\partial x^2}$
$\frac{\partial \hat{u}}{\partial t} - \lambda \frac{\partial^2 \hat{u}}{\partial x^2}$
$\mathcal{T}_f$
I
$\hat{u}(x,t) - g_D(x,t)$
$\frac{\partial}{\partial \mathbf{n}}$
$\frac{\partial \hat{u}}{\partial \mathbf{n}}(x,t) - g_R(u,x,t)$
$\mathcal{T}_b$
Loss
Minimize
$\boldsymbol{\theta}^*$
BC & IC

Figure 3.1: *PINN algorithm. 1D heat equation. (from Ref. [25]).*

The IC is dealt with as a special case of Dirichlet data on the spatio-temporal border $t = 0$, so that:

$$\begin{cases} \dfrac{\partial u}{\partial t} - \lambda \dfrac{\partial^2 u}{\partial x^2} = 0 & \text{in } \Omega \\ u(x,t) - g_D(x,t) = 0 & \text{on } \Gamma_D \subset \partial\Omega \\ \mathbf{n} \cdot \boldsymbol{\nabla}\, u(x,t) - g_R(u,x,t) = 0 & \text{on } \Gamma_R \subset \partial\Omega \end{cases} \tag{3.10}$$

All the codes written in this thesis run DeepXDE [25, 37], a deep learning Python library conceived to solve differential problems. Moreover, this library has the potential to:

- randomly generate different training points $\mathcal{T}$ in each optimization iteration, this is very advantageous in multiscale problems where the number of training points required is very large and would be computationally expensive to perform every iteration with the full batch of training points;
- enforce *hard constraints* of BC/IC, for simple cases: the loss $\mathcal{L}_b$ introduced by a hard-constrained BC/IC is *zero*.

Each of these options will be fully employed in every possible case. Based on the experience developed throughout this work, the hard constraint of the IC doesn't work for steep IC (e.g., in Riemann problems).

3.4 Error analysis

The loss $\mathcal{L}$ is a non-convex, highly nonlinear, function with respect to the parameters set $\boldsymbol{\theta}$. The optimization problem for a function like that doesn't have, in general, a unique solution. Consequently, there is no guarantee on unique solutions for PINNs. However, the question of whether there exists a NN satisfying both the PDE and the BC can be addressed. The theorem of derivative approximation due to Pinkus [25,28], expressed using a single hidden layer NN, shows that a feedforward NN with *enough* neurons can uniformly approximate any function and its partial derivatives. Yet, the intrinsic limits of the NN approach, explained in [3], are due to the fact that:

1. The NN have inevitably limited size. Let $\mathcal{F}$ be the family of all the functions representable by a given finite-size NN; the *best* function representable by the NN, i.e., the closest to u, is defined as: $u_{\mathcal{F}} = \text{argmin}_{f \in \mathcal{F}} \|f - u\|$. The **approximation error** $\mathcal{E}_{\mathbf{app}}$ is defined as: $\|u_{\mathcal{F}} - u\|$.

2. The NN are trained on a finite set of training points. The NN's representable function if the loss is at the global minimum is defined as: $u_{\mathcal{T}} = \text{argmin}_{f \in \mathcal{F}} \mathcal{L}(f, \mathcal{T})$. The **generalization error** $\mathcal{E}_{\mathbf{gen}}$, determined both by the density of the training points and by the NN's expressiveness, is defined as: $\|u_{\mathcal{T}} - u_{\mathcal{F}}\|$.

3. Minimizing the non-convex, highly nonlinear, loss function can be computationally unmanageable [2], hence, the computation of the global minimum of $\mathcal{L}$ is a task very unlikely to be performed. Let $\tilde{u}_{\mathcal{T}}$ be the actual function represented by the NN as a result of the training. Hence, the **optimization error** $\mathcal{E}_{\mathbf{opt}}$ is defined as: $\|\tilde{u}_{\mathcal{T}} - u_{\mathcal{T}}\|$.

Hence, the total error $\mathcal{E}$ is defined as:

$$\mathcal{E} \overset{\text{def}}{=\!=} \|\tilde{u}_{\mathcal{T}} - u\| \leq \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{gen}} + \mathcal{E}_{\text{opt}} \tag{3.11}$$

As previously stated, the estimation of this error is currently still an open research problem; the illustration of this error decomposition is shown in Fig. 3.2.

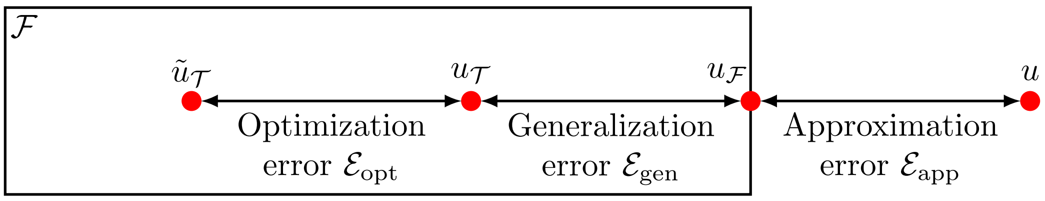

$\mathcal{F}$
$\tilde{u}_{\mathcal{T}}$
$u_{\mathcal{T}}$
$u_{\mathcal{F}}$
u
Optimization error $\mathcal{E}_{\text{opt}}$
Generalization error $\mathcal{E}_{\text{gen}}$
Approximation error $\mathcal{E}_{\text{app}}$

Figure 3.2: *Error analysis. Decomposition of the total error (from Ref. [25]).*

3.5 Code validation

In this section, the Python code implementing the PINNs is tested on three different classes of problem. In order to investigate the accuracy of the method, the relative error of the NN solutions with respect to the exact solutions is computed in norm $L^2(\Omega)$.

3.5.1 Elliptic problem, *parametric*

The code is tested on the *Poisson equation*. The problem is defined in the domain $\Omega = (x_{\min}, x_{\max}) \times (\kappa_{\min}, \kappa_{\max})$, for $x \in (0,1)$ and $\kappa \in (1,4)$.

Find $u(x,\kappa)\colon \mathbb{R}^2 \to \mathbb{R}$ so that:

$$\begin{cases} -\dfrac{\partial^2 u}{\partial x^2} = f(x,\kappa) & \text{in } \Omega \\ u(0,\kappa) = u(1,\kappa) = 0 \end{cases} \tag{3.12}$$

The selected forcing term, and the corresponding exact solution obtained by integration are:

$$f(x,\kappa) = \sin(2\pi\kappa x) \tag{3.13}$$

$$u_{\text{ex}}(x,\kappa) = \frac{1}{(2\pi\kappa)^2}\Big[\sin(2\pi\kappa x) - \sin(2\pi\kappa)x\Big] \tag{3.14}$$

In order to set the problem properly before the training, two observation have to be done:

1. The factor $\frac{1}{(2\pi\kappa)^2}$ implies small solution values whereas the NN performance are better around $\mathcal{O}(1)$ solution values, therefore the problem must be scaled so that $\tilde{u} = \mathbf{U}u$, where $\mathbf{U}$ is the trade-off value $\kappa_{\max}(2\pi)^2$.

2. The Dirichlet BCs in such a simple domain can be hard constrained, i.e., the loss contribution due to the BCs will be zero.

Hyperparameters. For the training is used an Adam optimizer [18] and a fixed learning rate of 10^{-4}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer. The training points in the domain and on the boundary are 10^4 and 5, respectively, both *resampled* every 5×10^4 iterations. The number of iterations is 6.5×10^5.

Even though both the BCs are hard constrained in the code, they still enter the loss function as penalty terms ($\mathcal{L}_{BC_0}$, $\mathcal{L}_{BC_1}$) with associated weights set to 1. The same weight value proved to work fine for the loss term generated by the PDE residual ($\mathcal{L}_{PDE}$). As a result, the suitable loss function for this problem is:

$$\mathcal{L} = \mathbf{1}\mathcal{L}_{PDE} + \mathbf{1}\mathcal{L}_{BC_0} + \mathbf{1}\mathcal{L}_{BC_1} \tag{3.15}$$

The results for a selected set of parameter values, provided with the relative errors of the PINN solutions computed in norm $L^2(\Omega)$, are shown in Fig. 3.3 on page 56.

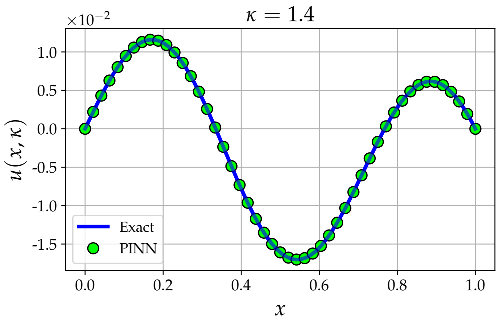
3.5.2 Parabolic problem, *parametric*

The code is tested on the *heat equation*. The problem is defined in the cubic domain $\Omega \times (0, T)$ with $\Omega = (x_{\text{min}}, x_{\text{max}}) \times (\kappa_{\text{min}}, \kappa_{\text{max}})$, considering $x \in (0, 1)$, $\kappa \in (1, 4)$ and $t \in (0, 1)$.

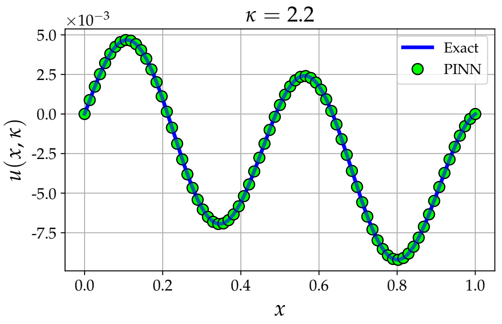
Find $u(x, \kappa, t) \colon \mathbb{R}^3 \to \mathbb{R}$ so that:

$$\begin{cases} \dfrac{\partial u}{\partial t} - \dfrac{\partial^2 u}{\partial x^2} = f(x, \kappa, t) & \text{in } \Omega \times (0, T) \\ u = 0 & \text{in } \Omega : t = 0 \\ u(0, \kappa, t) = u(1, \kappa, t) = 0 \end{cases} \tag{3.16}$$

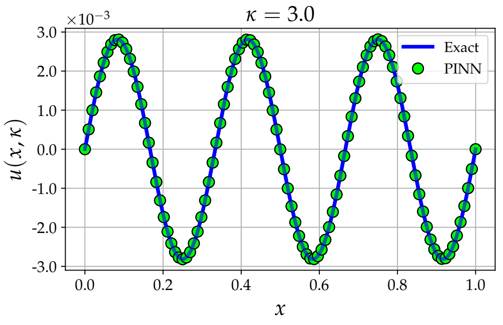
The forcing term, obtained by differentiation, and the the selected exact solution are:



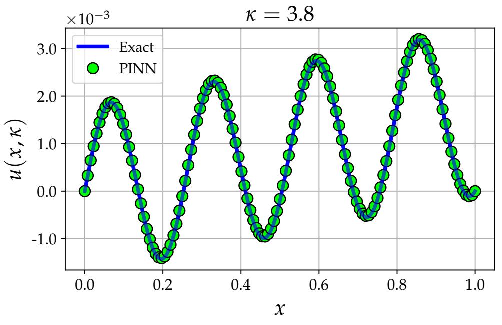
(a) $\frac{\|u_{ex}-u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 2.232 \times 10^{-7}$



(b) $\frac{\|u_{ex}-u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 5.668 \times 10^{-6}$



(c) $\frac{\|u_{ex}-u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 2.220 \times 10^{-6}$



(d) $\frac{\|u_{ex}-u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 2.044 \times 10^{-4}$

Figure 3.3: *Elliptic problem, parametric. Poisson equation. PINN solutions for selected values of the parameter κ. Errors computed in norm $L^2(\Omega)$.*

$$\begin{aligned} f(x,\kappa,t) = &\Big[\sin(2\pi\kappa x) - \sin(2\pi\kappa)x\Big]\left[\sin\left(\frac{\pi}{2}t\right) + \left(\frac{\pi}{2}t\right)\cos\left(\frac{\pi}{2}t\right)\right] \\ &+ \sin\left(\frac{\pi}{2}t\right)\sin(2\pi\kappa x)(2\pi\kappa)^2 t \end{aligned} \tag{3.17}$$

$$u_{\text{ex}}(x,\kappa,t) = \Big[\sin(2\pi\kappa x) - \sin(2\pi\kappa)x\Big]\sin\left(\frac{\pi}{2}t\right)t \tag{3.18}$$

Both the Dirichlet BCs, in such a simple domain, and the homogeneous IC can be hard constrained, i.e., the loss contribution due to the BCs and to the IC will be zero.

Hyperparameters. For the training is used an Adam optimizer and a fixed learning rate of 10^{-4}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer. The training points in the domain, on the boundary and for the IC are 1.5×10^4, 5 and 5, respectively, each one *resampled* every 5×10^4 iterations. The number of iterations is 5×10^5.

The IC and the two BCs, hard constrained in the code, still enter the loss function as penalty terms ($\mathcal{L}_{IC}$, $\mathcal{L}_{BC_0}$, $\mathcal{L}_{BC_1}$) with associated weights set to 1. The weight value that proved to work fine for the loss term generated by the PDE residual ($\mathcal{L}_{PDE}$) is 10^{-4}. Hence, the suitable loss function for this problem is:

$$\mathcal{L} = \mathbf{10^{-4}}\mathcal{L}_{PDE} + \mathbf{1}\mathcal{L}_{IC} + \mathbf{1}\mathcal{L}_{BC_0} + \mathbf{1}\mathcal{L}_{BC_1} \tag{3.19}$$

The results for picked out values of κ and t, provided with the relative errors of the PINN solutions computed in norm $L^2(\Omega)$, are shown in Fig. 3.4 on page 58.

3.5.3 Nonlinear hyperbolic problem

The code is tested on the *Burgers' equation* in the domain $\Omega \times (0, T)$, considering $\Omega = (-0.5, 1.5)$ and $t \in (0, 1)$.

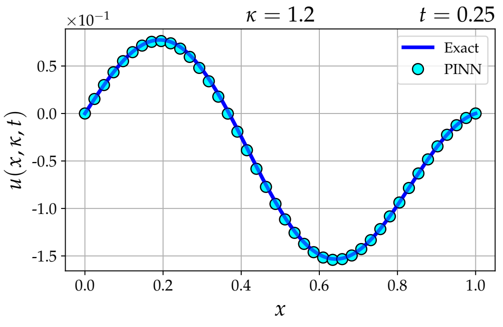
Find $u(x,t) \colon \mathbb{R}^2 \to \mathbb{R}$ so that:

$$\begin{cases} \dfrac{\partial u}{\partial t} + u\dfrac{\partial u}{\partial x} = 0 & \text{in } \Omega \times (0,T) \\ u = u_0 & \text{in } \Omega \, : \, t = 0 \\ u = 1 & \text{on } \partial\Omega_{in} \times (0,T) \end{cases} \tag{3.20}$$

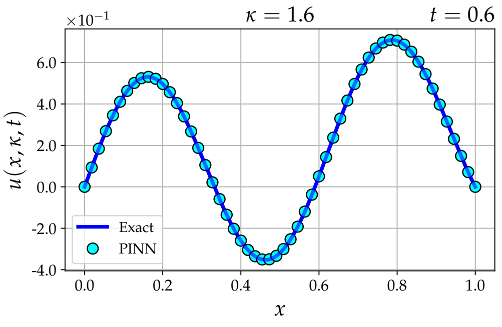
where $\partial\Omega_{in}$ denotes the inflow boundary, specifically the *left* boundary, since the IC is defined as:

$$u_0(x) = \begin{cases} 1 & x \leq 0 \\ 1 - x & 0 < x < 1 \\ 0 & x \geq 1 \end{cases} \tag{3.21}$$

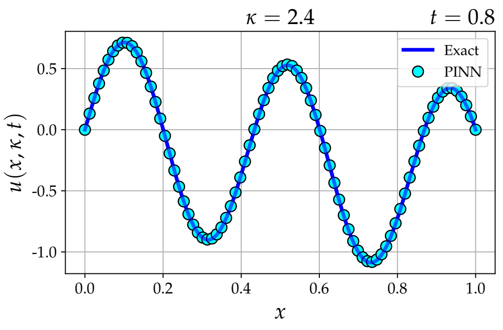
For the case of this selected IC, a shock wave is expected to originate at the spatio-temporal coordinates $(x = 1, t = 1)$.



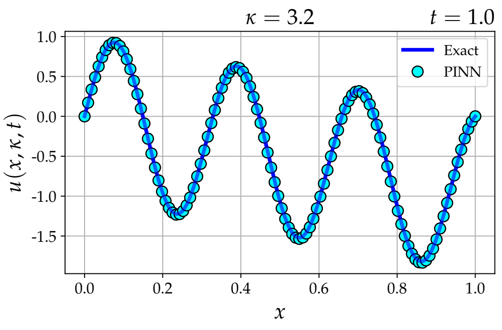
(a) $\frac{\|u_{ex}-u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 2.514 \times 10^{-4}$



(b) $\frac{\|u_{ex}-u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 1.876 \times 10^{-6}$



(c) $\frac{\|u_{ex}-u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 3.706 \times 10^{-5}$



(d) $\frac{\|u_{ex}-u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 1.947 \times 10^{-4}$

Figure 3.4: *Parabolic problem, parametric. Heat equation. PINN solutions at four temporal snapshots, for selected values of the parameter κ. Errors computed in norm $L^2(\Omega)$.*

The form of the exact solution, acceptable before the occurrence of the shock wave, is:

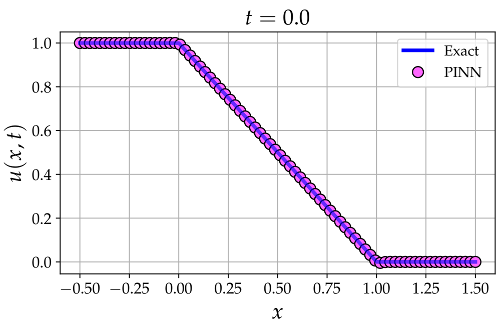
$$u_{ex}(x,t) = \begin{cases} 1 & x \leq t \\ \dfrac{x-1}{t-1} & t < x < 1 \\ 0 & x \geq 1 \end{cases} \tag{3.22}$$

Hyperparameters. For the training is used an Adam optimizer and a fixed learning rate of 10^{-4}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer. The training points in the domain, on the boundary and for the IC are 10^4, 10 and 50, respectively, each one *resampled* every

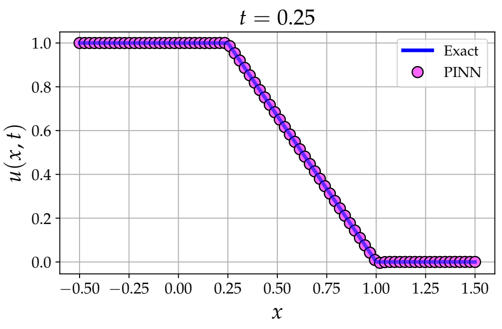
5×10^4 iterations. The number of iterations is 7.5×10^5. The loss function is made up of three terms ($\mathcal{L}_{PDE}$, $\mathcal{L}_{IC}$, $\mathcal{L}_{BC_{in}}$), respectively generated by the PDE, IC and BC residuals. Every single term enters the loss function with an associated weight set to 1:

$$\mathcal{L} = \mathbf{1}\mathcal{L}_{PDE} + \mathbf{1}\mathcal{L}_{IC} + \mathbf{1}\mathcal{L}_{BC_{in}} \tag{3.23}$$

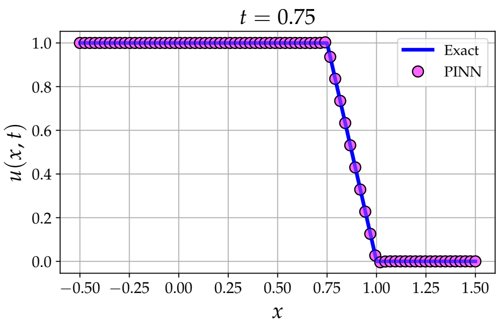
The results, provided with the relative errors of the PINN solutions computed in norm $L^2(\Omega)$, are shown in Fig. 3.5. Another approach to this problem can be found in [32], where Raissi et al. applied the PINN method to the *viscous* Burgers' equation, investigating the accuracy of the results for different network architectures and for different number of training points.



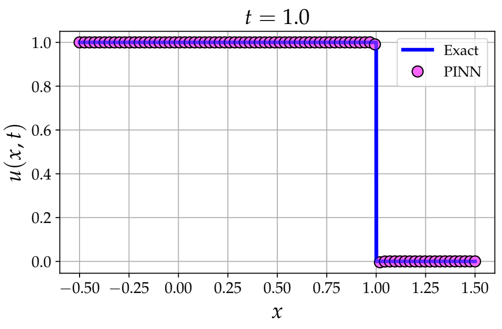
(a) $\frac{\|u_{ex} - u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 7.765 \times 10^{-7}$



(b) $\frac{\|u_{ex} - u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 6.760 \times 10^{-7}$



(c) $\frac{\|u_{ex} - u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 6.311 \times 10^{-7}$



(d) $\frac{\|u_{ex} - u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 3.358e \times 10^{-4}$

Figure 3.5: *Nonlinear hyperbolic problem. Burgers' equation. PINN solutions at four temporal snapshots. Errors computed in norm $L^2(\Omega)$.*

Chapter 4

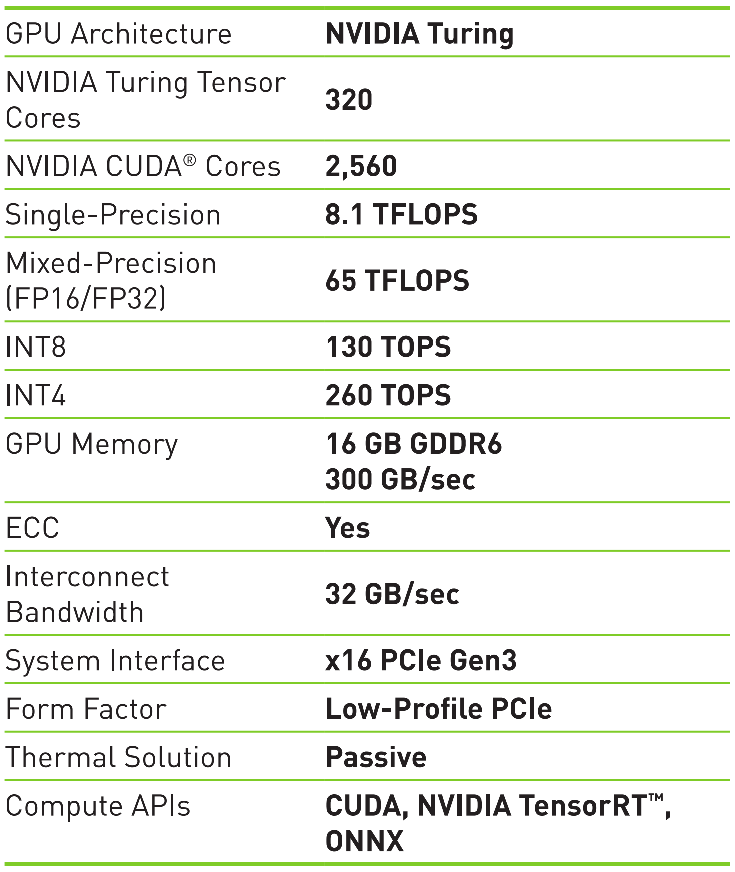
Results

4.1 Introduction

This chapter presents the numerical solutions produced, by the PINN tool, to both steady state and transient problems. Many of these problems have got an exact solution the PINN output will be compared to, whereas for the other cases the comparison solutions will be the numerical results provided by two other methods: for the one-dimensional cases it will be FullSWOF_1D, a C++ code based on the one-dimensional shallow water equations [6]; for the two-dimensional case it will be the Liou–Steffen splitting (LSS), a finite-volume component-wise Total Variation Diminishing (TVD) scheme [23].

The first two cases, examined in Sections 4.2 and 4.3, present hydrostatic equilibrium problems with an increased level of difficulty [5]. In Section 4.4, while there is still a steady state solution, *impulsive* boundary terms are introduced [13].

The next five cases are devoted to a thorough analysis of the one-dimensional dam break problem on a flat topography, once again with an increased level of difficulty, basing on the cased proposed in [7]. First, the problem on a wet domain is investigated in Section 4.5. Then, the additional difficulty of the dry domain is considered in Section 4.7. Even more, a friction term is added to the dry domain case in Section 4.8. Two *parametric* cases are built upon both the wet and the dry cases, they are described in Sections 4.6 and 4.9. No analytic solutions are available for the cases that involve friction terms.



GPU Architecture	**NVIDIA Turing**
NVIDIA Turing Tensor Cores	**320**
NVIDIA CUDA® Cores	**2,560**
Single-Precision	**8.1 TFLOPS**
Mixed-Precision (FP16/FP32)	**65 TFLOPS**
INT8	**130 TOPS**
INT4	**260 TOPS**
GPU Memory	**16 GB GDDR6 300 GB/sec**
ECC	**Yes**
Interconnect Bandwidth	**32 GB/sec**
System Interface	**x16 PCIe Gen3**
Form Factor	**Low-Profile PCIe**
Thermal Solution	**Passive**
Compute APIs	**CUDA, NVIDIA TensorRT™, ONNX**

Figure 4.1: *NVIDIA T4 Specifications (from Ref. [39]).*

Finally, the two-dimensional shallow water equations are examined. The circular dam break [11, 23] is examined in Section 4.10. The problem is expanded in Section 4.11 by making the gravitational acceleration a wide-ranging *parameter* of the problem.

All the tests are conducted on a NVIDIA® T4 GPU, a cloud computing resource freely accessible in Google Colaboratory [38]. The details of this hardware are given in Fig. 4.1.

4.2 Lake at rest with an immersed bump

This case tests the ability of the PINN to *preserve* steady states. The problem is defined in the domain $\Omega \times (0, T)$ with $\Omega = (x_{\min}, x_{\max})$, considering $x \in (0, 25)$ and $t \in (0, 100)$. The topography, given by the Eq. (4.1), is completely immersed and is flat at the boundaries.

$$z(x) = \begin{cases} 0.2 - 0.05(x-10)^2 & \text{if } 8 \text{ m} < x < 12 \text{ m} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

The initial conditions are:

$$\begin{cases} h = 0.5 - z & \text{m} \\ q = 0 & \text{m}^2/\text{s} \end{cases} \tag{4.2}$$

and the boundary conditions are chosen as:

$$\begin{cases} h = 0.5 & \text{m} \\ q = 0 & \text{m}^2/\text{s} \end{cases} \tag{4.3}$$

The problem involves ICs and BCs for h and q, while the SWE are implemented holding as unknowns h and u, as explained in Section 1.2 on page 13. This obstacle is solved adding the definition of the flow as a third, algebraic equation, to the PDE system. The complete system is given in Eq. (4.4).

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial (hu)}{\partial x} = 0 \\ \dfrac{\partial (hu)}{\partial t} + \dfrac{\partial (hu^2 + \frac{1}{2} g h^2)}{\partial x} = ghS_{0_x} \\ q - hu = 0 \end{cases} \tag{4.4}$$

The representation of the initial condition, that the numerical scheme is tested to keep unchanged for a large time, is shown in Fig. 4.2.

Model setup. In order to make the learning as fast as possible, the model is suitably scaled and hard constraints are imposed, where possible. Specifically, the reference values for the scaling are set as:

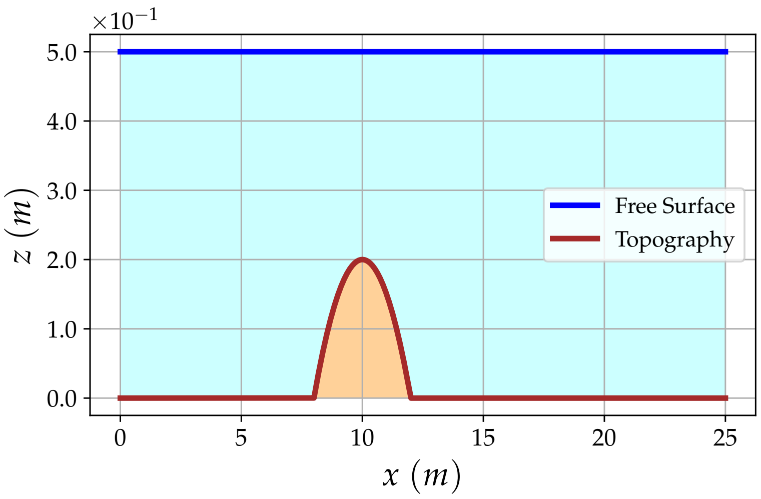


Figure 4.2: *Immersed bump. Steady state condition.*

- $\mathbf{L} = 1$
- $\mathbf{H} = 1$
- $\mathbf{T} = 100$
- $\mathbf{U} = 1$

Both the ICs for the water height and for the flow have been hard constrained. Considering that the problem is *stationary*, and that the flow must be *null* over the whole spatio-temporal domain, the relation between the water velocity and the flow is used to build both IC and BC for the velocity without alter the setup of the original problem.

Hyperparameters. For the training is used an Adam optimizer and a fixed learning rate of 10^{-3}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer.

The training points in the domain, on the boundary and for the IC are 4×10^3, 2×10^3 and 10^3, respectively, each one *resampled* every 100 iterations. The number of iterations is 4×10^5.

The loss function is made up of nine terms: ($\mathcal{L}^h_{PDE}$, $\mathcal{L}^{hu}_{PDE}$, $\mathcal{L}^q_{equat}$, $\mathcal{L}^h_{BC}$, $\mathcal{L}^u_{BC}$, $\mathcal{L}^q_{BC}$ $\mathcal{L}^h_{IC}$, $\mathcal{L}^u_{IC}$, $\mathcal{L}^q_{IC}$), respectively generated by the residuals of the mass conservation and of the momentum balance PDEs, the algebraic equation for the flow, the BCs and the ICs for the variables h, u and q. Every term enters the loss function with an associated weight set to 1:

$$\begin{aligned}\mathcal{L} = &\ \mathbf{1}\mathcal{L}^{h}_{PDE} + \mathbf{1}\mathcal{L}^{hu}_{PDE} + \mathbf{1}\mathcal{L}^{q}_{equat} \\ &+ \mathbf{1}\mathcal{L}^{h}_{BC} + \mathbf{1}\mathcal{L}^{u}_{BC} + \mathbf{1}\mathcal{L}^{q}_{BC} + \mathbf{1}\mathcal{L}^{h}_{IC} + \mathbf{1}\mathcal{L}^{u}_{IC} + \mathbf{1}\mathcal{L}^{q}_{IC}\end{aligned} \tag{4.5}$$

The results of the learning of *all* the variables, h, u and q, provided with the errors of the PINN's approximations computed in norm $L^2(\Omega)$, are shown in Fig. 4.3 on page 65. The integral errors account for a very good numerical approximation. Moreover, examining the plots, the characteristics of the problem seem to be replicated pretty well.

4.3 Lake at rest with an emerged bump

This case tests the ability of the PINN to preserve steady states, furthermore, the *wet/dry transition* treatment is analyzed. The problem is defined in the domain $\Omega \times (0, T)$ with $\Omega = (x_{\text{min}}, x_{\text{max}})$, considering $x \in (0, 25)$ and $t \in (0, 100)$. The topography, given by the Eq. (4.1) on page 62, is the same one of the previous case except that in this instance the water height is set to a value that makes some parts of the topography emerge.

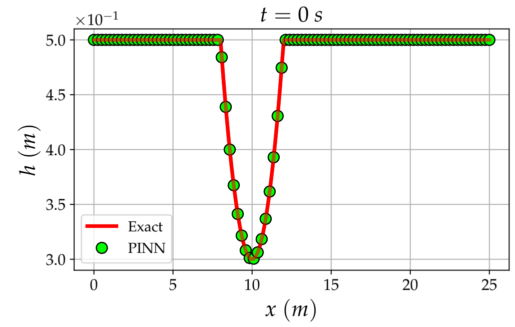
Indeed, the initial conditions are:

$$\begin{cases} h = \max(0.1, z) - z & \text{m} \\ q = 0 & \text{m}^2/\text{s} \end{cases} \tag{4.6}$$

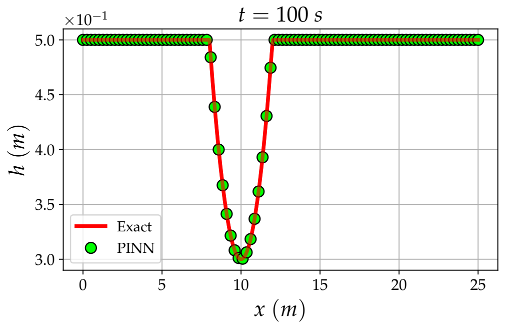
and the boundary conditions are chosen as:

$$\begin{cases} h = 0.1 & \text{m} \\ q = 0 & \text{m}^2/\text{s} \end{cases} \tag{4.7}$$

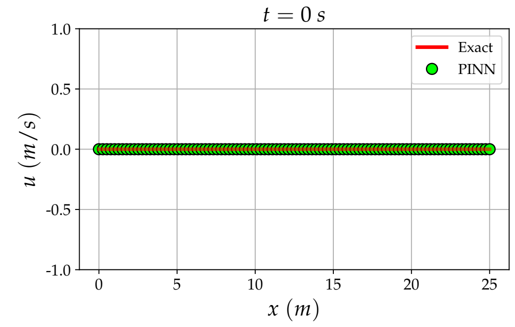
As in the previous case, the problem involves ICs and BCs for h and q, while the SWE are implemented holding as unknowns h and u. Once again, the complication is worked out adding the definition of the flow as a third, algebraic equation, to the PDE system (see Eq. (4.4) on page 62).



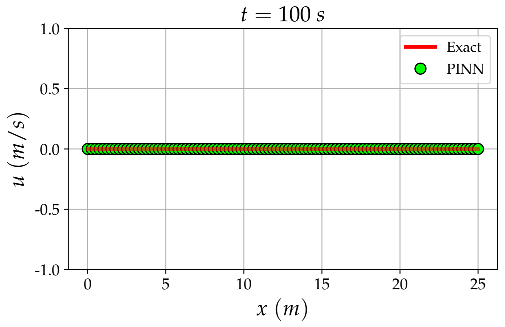
(a) $\frac{\|h_{ex}-h_{PINN}\|_{L^2}}{\|h_{ex}\|_{L^2}} = 0$



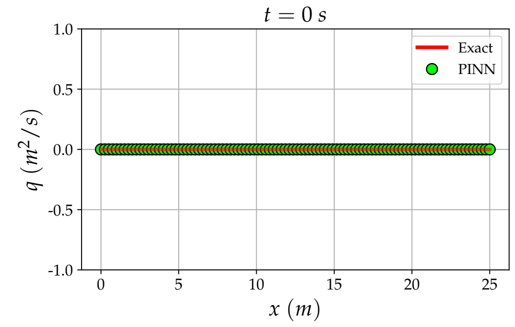
(b) $\frac{\|h_{ex}-h_{PINN}\|_{L^2}}{\|h_{ex}\|_{L^2}} = 2.431 \times 10^{-15}$



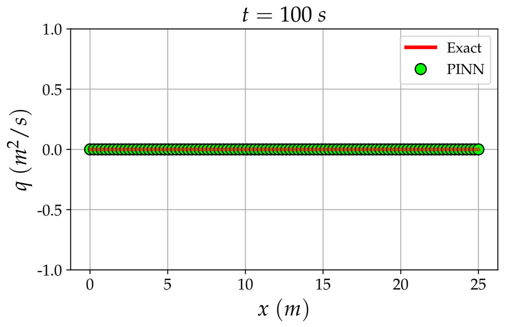
(c) $\|u_{ex} - u_{PINN}\|_{L^2} = 2.415 \times 10^{-11}$



(d) $\|u_{ex} - u_{PINN}\|_{L^2} = 1.483 \times 10^{-10}$



(e) $\|q_{ex} - q_{PINN}\|_{L^2} = 0$



(f) $\|q_{ex} - q_{PINN}\|_{L^2} = 3.322 \times 10^{-10}$

Figure 4.3: *Immersed bump. PINN solutions for **(a)**-**(b)** the water height, **(c)**-**(d)** the velocity, **(e)**-**(f)** the flow. Errors computed in norm $L^2(\Omega)$.*

The representation of the initial condition, that the numerical scheme is tested to keep unchanged for a large time, is shown in Fig. 4.4.

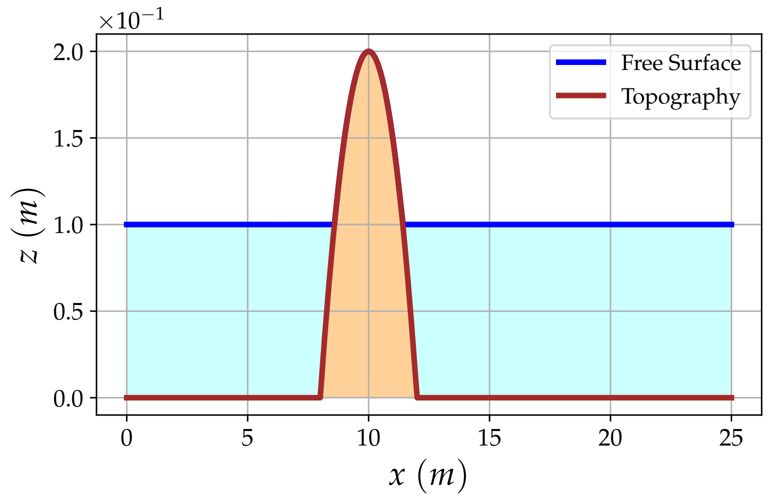


Figure 4.4: *Emerged bump. Steady state condition.*

Model setup. In order to make the learning as fast as possible, the model is suitably scaled and hard constraints are imposed, where possible. Specifically, the reference values for the scaling are set as:

- $\mathbf{L} = 1$
- $\mathbf{H} = 1$
- $\mathbf{T} = 100$
- $\mathbf{U} = 1$

Both the ICs for the water height and for the flow have been hard constrained. Considering that the problem is *stationary*, and that the flow must be *null* over the whole spatio-temporal domain, the relation between the water velocity and the flow is used to build both IC and BC for the velocity without alter the setup of the original problem.

Hyperparameters. For the training is used an Adam optimizer and a fixed learning rate of 10^{-3}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer.

The training points in the domain, on the boundary and for the IC are 4×10^3, 2×10^3 and 10^3, respectively, each one *resampled* every 100 iterations. The number of iterations is 6.5×10^5.

The loss function is made up of nine terms: ($\mathcal{L}^h_{PDE}$, $\mathcal{L}^{hu}_{PDE}$, $\mathcal{L}^q_{equat}$, $\mathcal{L}^h_{BC}$, $\mathcal{L}^u_{BC}$, $\mathcal{L}^q_{BC}$ $\mathcal{L}^h_{IC}$, $\mathcal{L}^u_{IC}$, $\mathcal{L}^q_{IC}$), respectively generated by the residuals of the mass conservation and of the momentum balance PDEs, the algebraic equation for the flow, the BCs and the ICs for the variables h, u and q. Every single term enters the loss function with an associated weight set to 1:

$$\begin{aligned}\mathcal{L} = \mathbf{1}\mathcal{L}^h_{PDE} + \mathbf{1}\mathcal{L}^{hu}_{PDE} + \mathbf{1}\mathcal{L}^q_{equat} \\ + \mathbf{1}\mathcal{L}^h_{BC} + \mathbf{1}\mathcal{L}^u_{BC} + \mathbf{1}\mathcal{L}^q_{BC} + \mathbf{1}\mathcal{L}^h_{IC} + \mathbf{1}\mathcal{L}^u_{IC} + \mathbf{1}\mathcal{L}^q_{IC}\end{aligned} \tag{4.8}$$

The results of the learning of *all* the variables, h, u and q, provided with the errors of the PINN's approximations computed in norm $L^2(\Omega)$, are shown in Fig. 4.5 on page 68. The integral errors account for an excellent numerical approximation, moreover, the qualitative analysis of the the plots confirms that the characteristics of the problem are replicated pretty well.

4.4 Subcritical flow

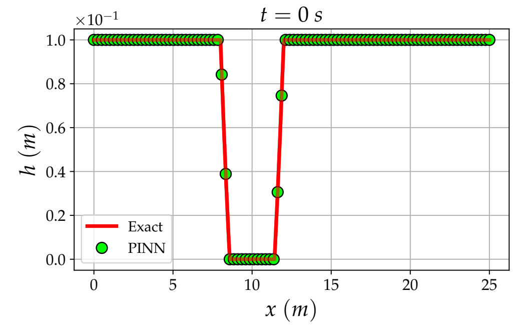
This case tests the ability of the PINN to *catch* steady states. Furthermore, the *impulsive* boundary terms treatment is analyzed. The problem is defined in the domain $\Omega \times (0, T)$ with $\Omega = (x_{\min}, x_{\max})$, considering $x \in (0, 25)$ and $t \in (0, 100)$. The topography, given by the Eq. (4.1) on page 62, is the same one of the previous two cases. The water height is set to a value that keeps every part of the topography completely immersed.

The initial conditions are:

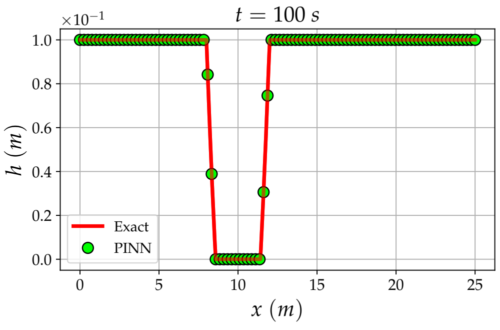
$$\begin{cases} h = 2 - z & \text{m} \\ q = 0 & \text{m}^2/\text{s} \end{cases} \tag{4.9}$$

and the boundary conditions are chosen as:

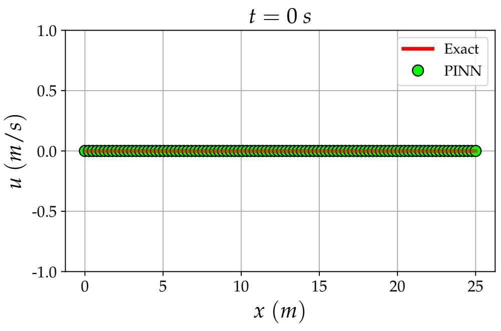
$$\begin{cases} \text{upstream:} & q = 4.42 \cdot \text{step}(t) & \text{m}^2/\text{s} \\ \text{downstream:} & h = 2 & \text{m} \end{cases} \tag{4.10}$$



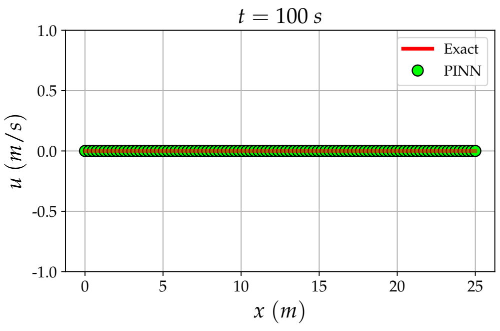
(a) $\frac{\|h_{ex} - h_{PINN}\|_{L2}}{\|h_{ex}\|_{L2}} = 0$



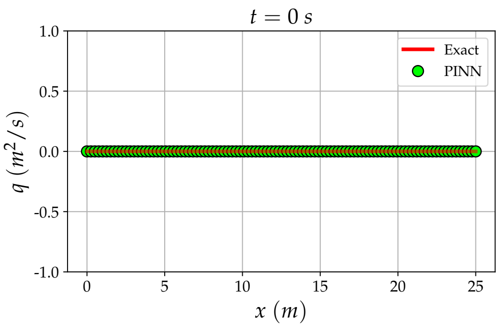
(b) $\frac{\|h_{ex} - h_{PINN}\|_{L2}}{\|h_{ex}\|_{L2}} = 3.919 \times 10^{-12}$



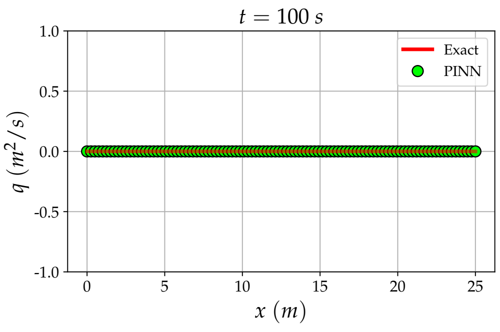
(c) $\|u_{ex} - u_{PINN}\|_{L^2} = 8.887 \times 10^{-12}$



(d) $\|u_{ex} - u_{PINN}\|_{L^2} = 7.958 \times 10^{-11}$



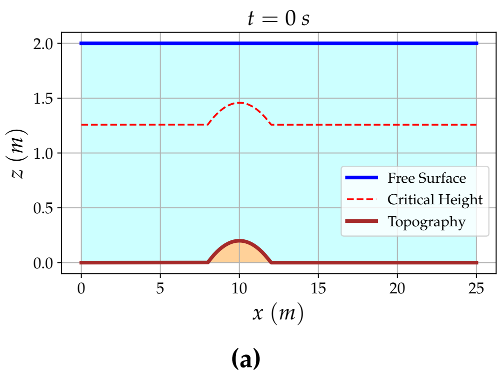
(e) $\|q_{ex} - q_{PINN}\|_{L^2} = 0$



(f) $\|q_{ex} - q_{PINN}\|_{L^2} = 2.702 \times 10^{-13}$

Figure 4.5: *Emerged bump. PINN solutions for **(a)**-**(b)** the water height, **(c)**-**(d)** the velocity, **(e)**-**(f)** the flow. Errors computed in norm $L^2(\Omega)$.*

The representation of the initial condition, and of the solution at the steady state that the numerical scheme is tested to catch, are shown in Fig. 4.6.



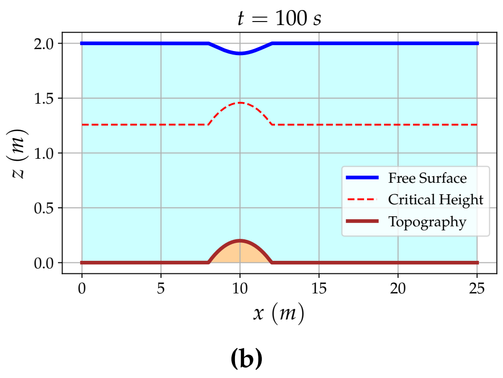


Figure 4.6: *Subcritical flow: **(a)** initial condition, **(b)** steady state solution.*

The problem still involves ICs and BCs for h and q. Conversely to the two previous problems, the SWE are here implemented holding as unknowns h and q:

$$
\begin{cases}
\dfrac{\partial h}{\partial t} + \dfrac{\partial q}{\partial x} = 0 \\
\dfrac{\partial q}{\partial t} + \dfrac{\partial}{\partial x}\left(\dfrac{q^2}{h} + \dfrac{1}{2}gh^2\right) = ghS_{0_x}
\end{cases}
\tag{4.11}
$$

Model setup. In order to make the learning as fast as possible, the model is suitably scaled and hard constraints are imposed, where possible. Specifically, the reference values for the scaling are set as:

- $\mathbf{L} = 1$
- $\mathbf{H} = 1$
- $\mathbf{T} = 100$
- $\mathbf{U} = 1$

Both the ICs for the water height and for the flow, and the upstream BC for the flow as well, have been hard constrained.

Hyperparameters. For the training is used an Adam optimizer and a fixed learning rate of 10^{-3}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer. The training points in the domain, on the boundary and for the IC are 4×10^3, 2×10^3 and 10^3, respectively, each one *resampled* every 100 iterations. The number of iterations is 5×10^5.

The loss function is made up of six terms: ($\mathcal{L}^h_{PDE}$, $\mathcal{L}^q_{PDE}$, $\mathcal{L}^h_{BC}$, $\mathcal{L}^q_{BC}$ $\mathcal{L}^h_{IC}$, $\mathcal{L}^q_{IC}$), respectively generated by the residuals of the mass conservation and of the momentum balance equations, the BCs and the ICs for the variables h and q. Every single term enters the loss function with an associated weight set to 1:

$$\begin{aligned}\mathcal{L} = &\mathbf{1}\mathcal{L}^h_{PDE} + \mathbf{1}\mathcal{L}^q_{PDE} \\ &+ \mathbf{1}\mathcal{L}^h_{BC} + \mathbf{1}\mathcal{L}^q_{BC} + \mathbf{1}\mathcal{L}^h_{IC} + \mathbf{1}\mathcal{L}^q_{IC}\end{aligned} \tag{4.12}$$

The results of the learning of *all* the variables, h, u and q, provided with the errors of the PINN's approximations computed in norm $L^2(\Omega)$, are shown in Fig. 4.7 on page 71. Specifically, the results for the velocity are built after the learning, applying the relation: $u = q/h$. Once again, in spite of the complexity introduced, the PINN provides a very good numerical approximation.

4.5 Dam break on a wet domain

This case is known as the Stoker's solution [35]. It is a classical Riemann problem. The problem is defined in the domain $\Omega \times (0, T)$ with $\Omega = (x_{\min}, x_{\max})$, considering $x \in (0, 10)$ and $t \in (0, 6)$. The dam break is instantaneous, the bottom is flat and there is no friction.

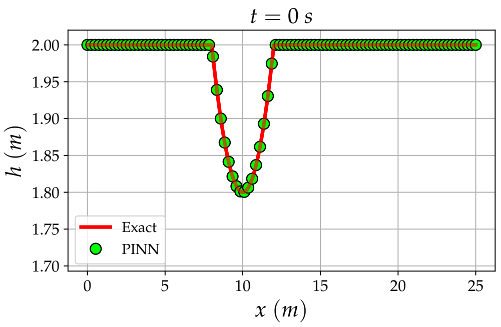
The initial conditions are:

$$h(x,0) = \begin{cases} 5 \times 10^{-3} \text{ m} & x \leq 5 \text{ m} \\ 1 \times 10^{-3} \text{ m} & x > 5 \text{ m} \end{cases} \tag{4.13}$$

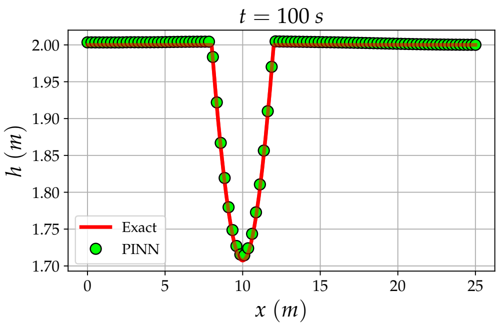
and

$$u = 0 \text{ m/s} \tag{4.14}$$

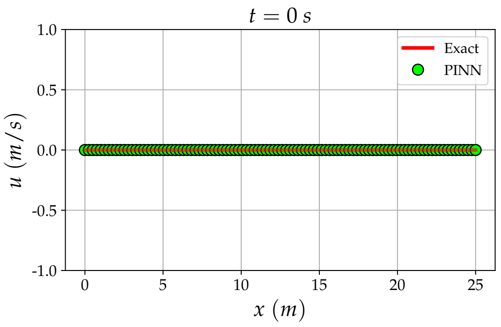
whereas no boundary condition is required to be set.



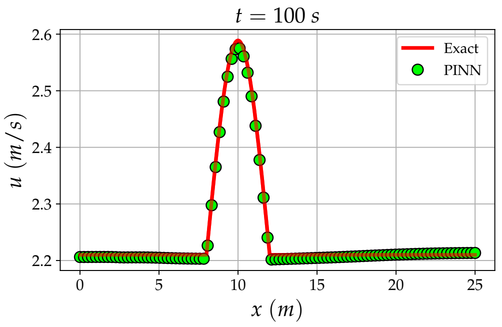
(a) $\frac{\|h_{ex} - h_{PINN}\|_{L^2}}{\|h_{ex}\|_{L^2}} = 0$



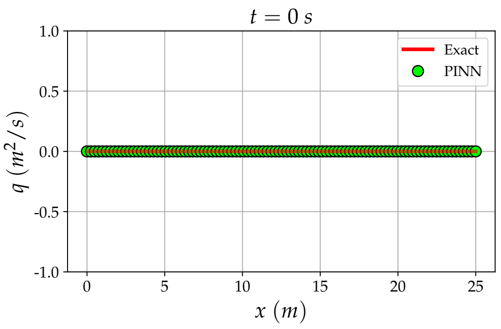
(b) $\frac{\|h_{ex} - h_{PINN}\|_{L^2}}{\|h_{ex}\|_{L^2}} = 3.519 \times 10^{-6}$



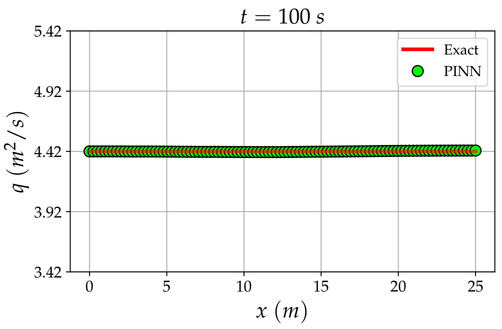
(c) $\|u_{ex} - u_{PINN}\|_{L^2} = 0$



(d) $\frac{\|u_{ex} - u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 6.927 \times 10^{-6}$



(e) $\|q_{ex} - q_{PINN}\|_{L^2} = 0$



(f) $\frac{\|q_{ex} - q_{PINN}\|_{L^2}}{\|q_{ex}\|_{L^2}} = 8.242 \times 10^{-7}$

Figure 4.7: *Subcritical flow. PINN solutions for* ***(a)-(b)*** *the water height,* ***(c)-(d)*** *the velocity,* ***(e)-(f)*** *the flow. Errors computed in norm $L^2(\Omega)$.*

The SWE are implemented holding as unknowns h and u, plus, two supplementary inequalities are included in the system for the purpose of making the learning faster. The complete system is given in Eq. (4.15).

$$\begin{cases} \frac{\partial h}{\partial t} + \frac{\partial (hu)}{\partial x} = 0 \\ \frac{\partial (hu)}{\partial t} + \frac{\partial (hu^2 + \frac{1}{2} g h^2)}{\partial x} = 0 \\ h \geq 1 \times 10^{-3} \\ u \geq 0 \end{cases} \tag{4.15}$$

Model setup. Considering that the initial data for water height belong to the $\mathcal{O}(10^{-3})$ scale, the model *has* to be scaled at least for the water height. Specifically, the reference values for the scaling are set as:

- $\mathbf{L} = 10$
- $\mathbf{H} = 5 \times 10^{-3}$
- $\mathbf{T} = 6$
- $\mathbf{U} = u_{\max}$

where $u_{\max}$ is the maximum value of the analytic solution for the water velocity, provided by FullSWOF_1D.

Hyperparameters. For the training is used an Adam optimizer and a fixed learning rate of 2×10^{-4}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer.

The training points in the domain and for the IC are 2×10^4 and 500, respectively. No training point is set on the boundary. The number of iterations is 9×10^5.

The loss function is made up of six terms: ($\mathcal{L}^h_{PDE}$, $\mathcal{L}^{hu}_{PDE}$, $\mathcal{L}^h_{ineq}$, $\mathcal{L}^u_{ineq}$ $\mathcal{L}^h_{IC}$, $\mathcal{L}^u_{IC}$), respectively generated by the residuals of the mass conservation and of the momentum balance equations, the inequalities and the ICs for the variables h and u. Every single term enters the loss function with an associated weight that proved to work fine after a trial-and-error effort:

$$\begin{aligned}\mathcal{L} = & \mathbf{10^2}\mathcal{L}^{h}_{PDE} + \mathbf{10^2}\mathcal{L}^{hu}_{PDE} \\ & + \mathbf{10^5}\mathcal{L}^{h}_{ineq} + \mathbf{10^5}\mathcal{L}^{u}_{ineq} \\ & + \mathbf{10^5}\mathcal{L}^{h}_{IC} + \mathbf{10^5}\mathcal{L}^{u}_{IC}\end{aligned} \tag{4.16}$$

The results of the learning of *all* the variables, h, u and q, provided with the errors of the PINN's approximations computed in norm $L^2(\Omega)$, are shown in Fig. 4.8 on page 74. Specifically, the results for the flow are built after the learning, applying the relation: $q = hu$. The integral errors account for a good numerical approximation. Examining the plots, even though the PINN's predictions are a little bit smoother than the exact solutions, the position of the shock seems to be identified pretty well.

4.6 Dam break on a wet domain with friction, *parametric*

This case is built upon the previous one, analyzed in Section 4.5. The problem is defined in the domain $\Omega \times (0, T)$, this time with:

$$\Omega = (x_{\min}, x_{\max}) \times (y_{\min}, y_{\max}) \tag{4.17}$$

considering $x \in (0, 10)$, $y \in (0, 1)$ and $t \in (0, 6)$. The additional variable y accounts for the parameter's space. Still, the dam break is instantaneous and the bottom is flat.

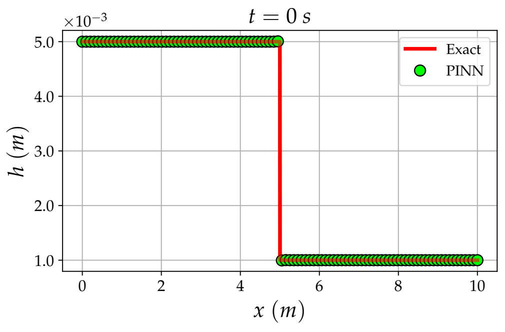
The initial conditions are:

$$h(x, y, 0) = \begin{cases} 5 \times 10^{-3} \text{ m} & x \leq 5 \text{ m} \\ 1 \times 10^{-3} \text{ m} & x > 5 \text{ m} \end{cases} \tag{4.18}$$

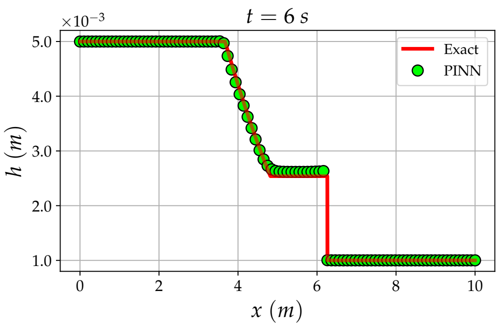
and

$$u = 0 \text{ m/s} \tag{4.19}$$

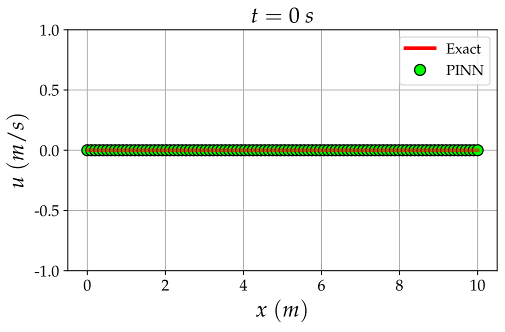
whereas no boundary condition is required to be set.



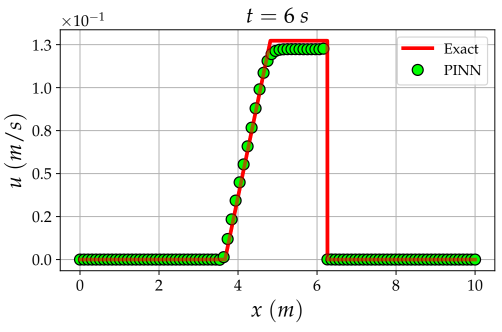
(a) $\frac{\|h_{ex}-h_{PINN}\|_{L^2}}{\|h_{ex}\|_{L^2}} = 3.054 \times 10^{-4}$



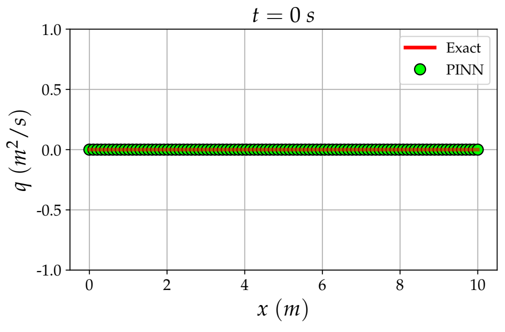
(b) $\frac{\|h_{ex}-h_{PINN}\|_{L^2}}{\|h_{ex}\|_{L^2}} = 1.079 \times 10^{-3}$



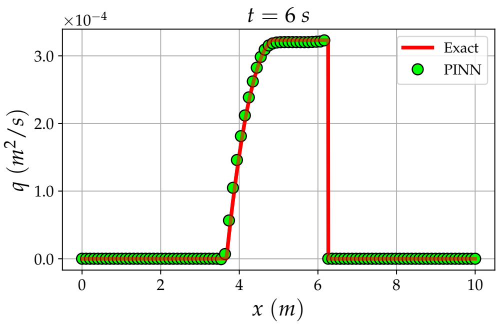
(c) $\|u_{ex} - u_{PINN}\|_{L^2} = 4.528 \times 10^{-2}$



(d) $\frac{\|u_{ex}-u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 2.873 \times 10^{-2}$



(e) $\|q_{ex} - q_{PINN}\|_{L^2} = 2.804 \times 10^{-7}$



(f) $\frac{\|q_{ex}-q_{PINN}\|_{L^2}}{\|q_{ex}\|_{L^2}} = 2.625 \times 10^{-2}$

Figure 4.8: *Dam break on a wet domain. PINN solutions for* ***(a)****-****(b)*** *the water height,* ***(c)****-****(d)*** *the velocity,* ***(e)****-****(f)*** *the flow. Errors computed in norm* $L^2(\Omega)$.

The additional parameter's space is integrated in the mathematical formulation by way of a linear transformation between the variable y and the Chézy's friction coefficient, C, actually entering the PDE:

$$C = 100y + 20 \tag{4.20}$$

This technique is used with the purpose of keeping the parameter's space dimension unitary, while obtaining a Chézy's friction coefficient ranging extensively from the values 20 and 120.

The SWE are implemented holding as unknowns h and u, plus, two supplementary inequalities are included in the system for the purpose of making the learning faster. The complete system is given in Eq. (4.21).

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial (hu)}{\partial x} = 0 \\ \dfrac{\partial (hu)}{\partial t} + \dfrac{\partial (hu^2 + \frac{1}{2}gh^2)}{\partial x} = -g\dfrac{u|\mathbf{u}|}{C^2} \\ h \geq 1 \times 10^{-3} \\ u \geq 0 \end{cases} \tag{4.21}$$

Model setup. Just as occurred in the case analyzed in Section 4.5, the initial data for water height belonging to the $\mathcal{O}(10^{-3})$ scale entails the need of a suitable model scaling, at least for the water height. Specifically, the reference values for the scaling are set as:

- $\mathbf{L} = 10$
- $\mathbf{H} = 5 \times 10^{-3}$
- $\mathbf{T} = 6$
- $\mathbf{U} = u_{\max}^{120}$

where $u_{\max}^{120}$ is the maximum value of the numerical solution for the water velocity, considering $C = 120$, computed by FullSWOF_1D.

Hyperparameters. For the training is used an Adam optimizer and a fixed learning rate of 10^{-4}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer. The training points in the domain and for the IC are 4×10^3 and 500, respectively. No training point is set on the boundary. The number of iterations is 8.5×10^5.

The loss function is made up of six terms: ($\mathcal{L}^h_{PDE}$, $\mathcal{L}^{hu}_{PDE}$, $\mathcal{L}^h_{ineq}$, $\mathcal{L}^u_{ineq}$ $\mathcal{L}^h_{IC}$, $\mathcal{L}^u_{IC}$), respectively generated by the residuals of the mass conservation and of the momentum balance equations, the inequalities and the ICs for the variables h and u. Every single term enters the loss function with an associated weight that proved to work fine after a trial-and-error effort:

$$\begin{aligned}\mathcal{L} = {} & \mathbf{10^2}\mathcal{L}^h_{PDE} + \mathbf{10^2}\mathcal{L}^{hu}_{PDE} \\ & + \mathbf{10^5}\mathcal{L}^h_{ineq} + \mathbf{10^5}\mathcal{L}^u_{ineq} \\ & + \mathbf{10^5}\mathcal{L}^h_{IC} + \mathbf{10^5}\mathcal{L}^u_{IC}\end{aligned} \tag{4.22}$$

The results of the learning of *all* the variables, h, u and q, are shown in Fig. 4.9, Fig. 4.10 and Fig. 4.11, respectively. Specifically, the results for the flow are built after the learning, applying the relation: $q = hu$. No error can be computed since no analytic solution is available. Examining the plots, the PINN's approximations are a little bit smoother than the FullSWOF_1D ones. However, the features of this case study—particularly, the positions of the shocks—appear to be identified pretty well over the whole wide-ranging parameter's space.

4.7 Dam break on a dry domain

This case is known as the Ritter's solution [33]. It is a classical Riemann problem. The problem is defined in the domain $\Omega \times (0, T)$ with $\Omega = (x_{\min}, x_{\max})$, considering $x \in (0, 10)$ and $t \in (0, 6)$. The dam break is instantaneous, the bottom is flat and there is no friction.

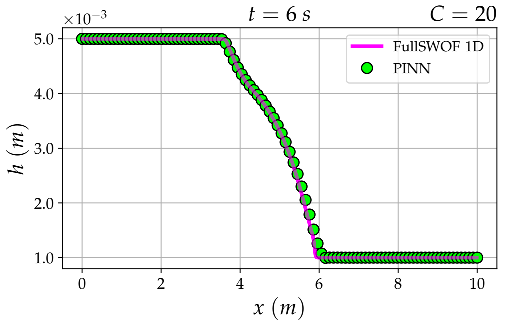
The initial condition for the water height is :

$$h(x,0) = \begin{cases} 5 \times 10^{-3} \ \text{m} & x \leq 5 \text{ m} \\ 0 \ \text{m} & x > 5 \text{ m} \end{cases} \tag{4.23}$$

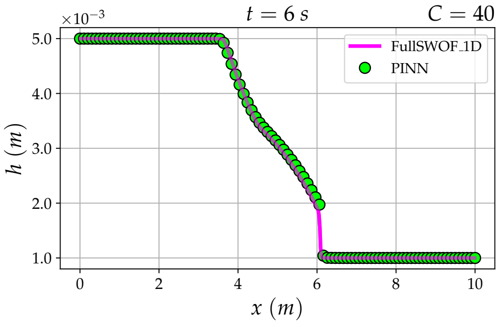
while both the IC and the BC for the water velocity are:

$$u = 0 \text{ m/s} \tag{4.24}$$

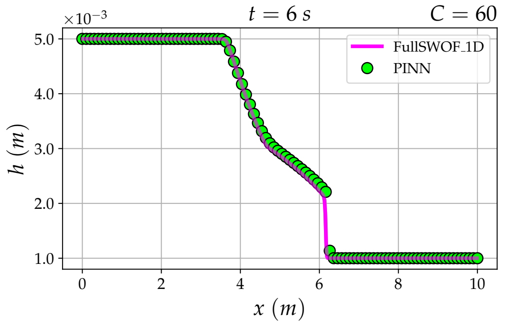
No boundary condition is set for the water height.



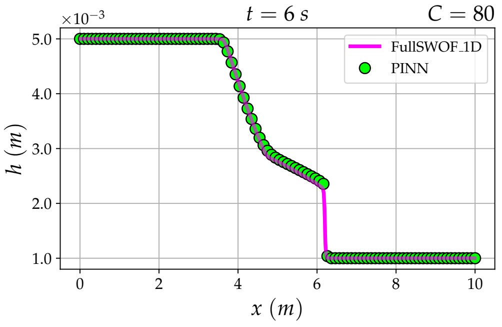
(a)



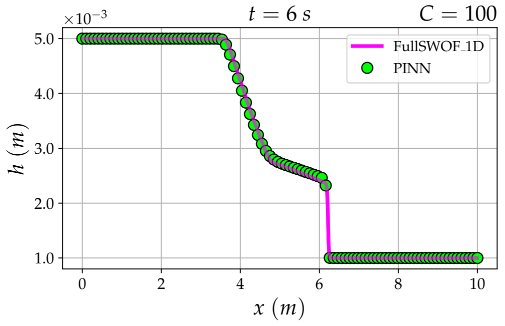
(b)



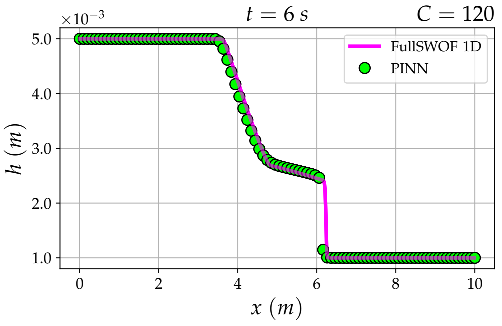
(c)



(d)

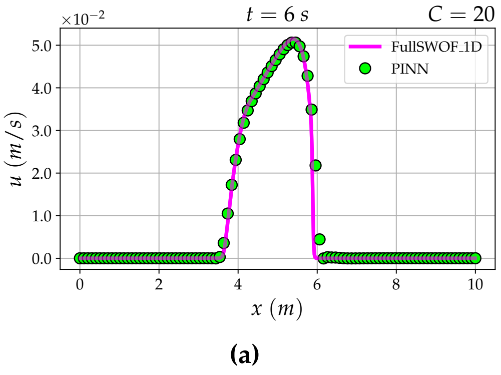


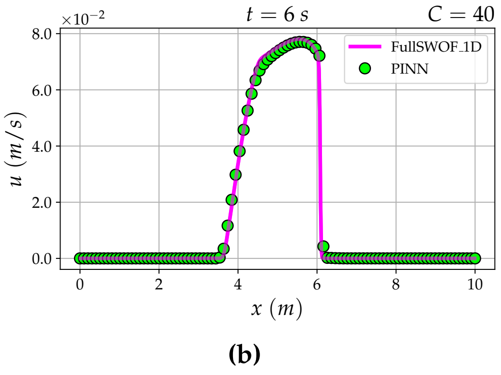
(e)

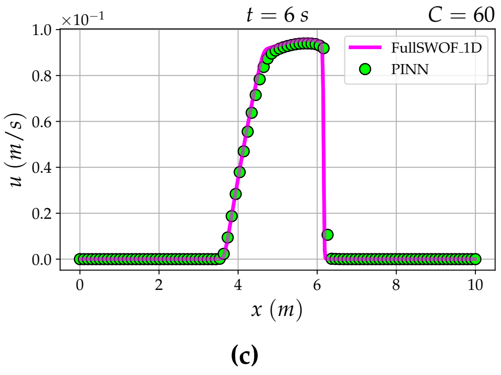


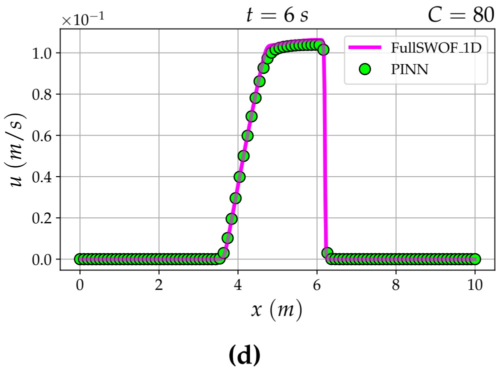
(f)

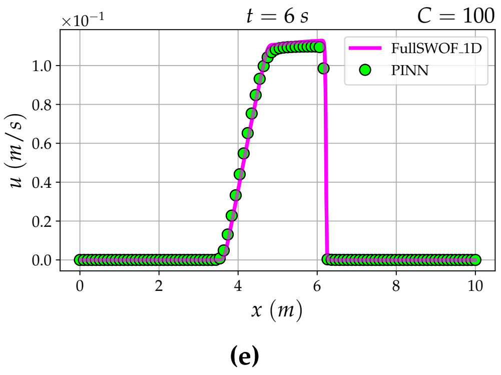
Figure 4.9: *Dam break on a wet domain with friction, parametric. PINN solutions for the water height. The parameter, C, ranges over the entire space. Comparison with the FullSWOF_1D results.*











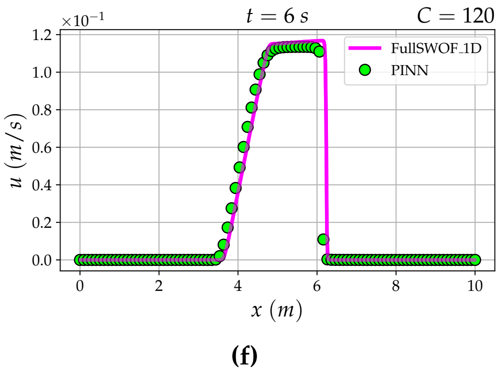
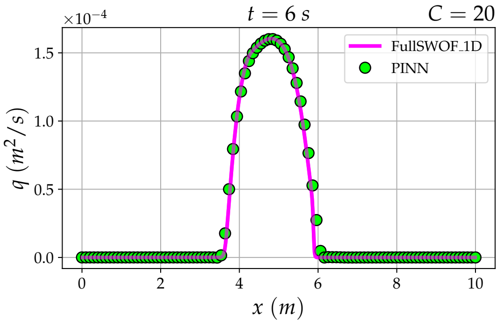
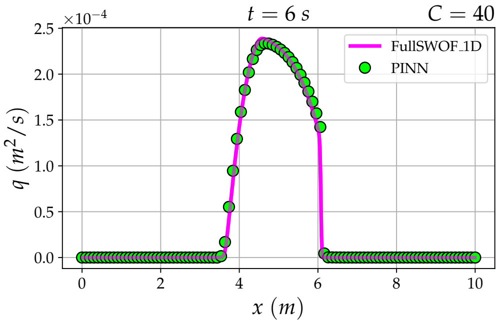


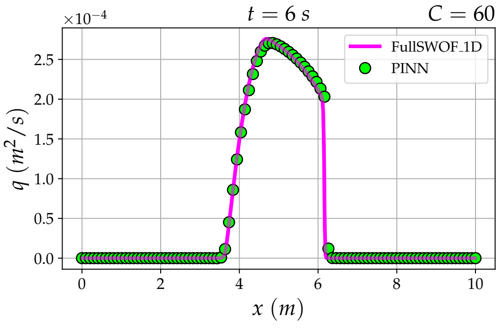
Figure 4.10: *Dam break on a wet domain with friction, parametric. PINN solutions for the water velocity. The parameter, C, ranges over the entire space. Comparison with the FullSWOF_1D results.*



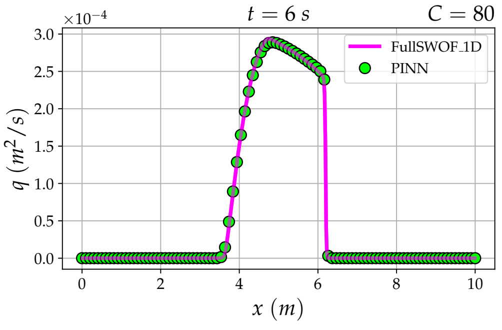
(a)



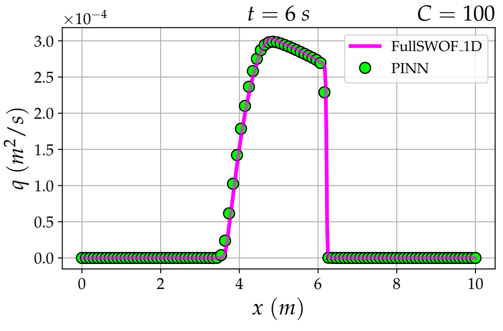
(b)



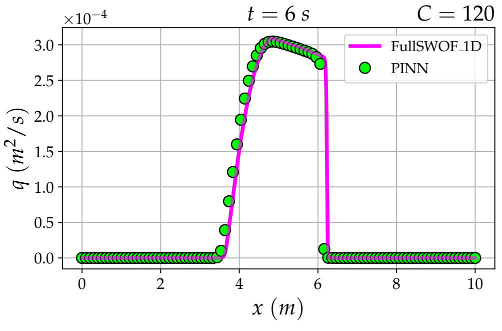
(c)



(d)



(e)



(f)

Figure 4.11: *Dam break on a wet domain with friction, parametric. PINN solutions for the water flow. The parameter, C, ranges over the entire space. Comparison with the FullSWOF_1D results.*

The SWE are implemented holding as unknowns h and u, plus, two supplementary inequalities are included in the system for the purpose of making the learning faster. The complete system is given in Eq. (4.25).

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial (hu)}{\partial x} = 0 \\ \dfrac{\partial (hu)}{\partial t} + \dfrac{\partial (hu^2 + \frac{1}{2} g h^2)}{\partial x} = 0 \\ h \geq 0 \\ u \geq 0 \end{cases} \tag{4.25}$$

Model setup. Just as occurred in the previous two cases, analyzed in the Sections 4.5 and 4.6, the initial data for water height belonging to the $\mathcal{O}(10^{-3})$ scale entails the need of a suitable model scaling, at least for the water height. Specifically, the reference values for the scaling are set as:

- $\mathbf{L} = 10$
- $\mathbf{H} = 5 \times 10^{-3}$
- $\mathbf{T} = 6$
- $\mathbf{U} = u_{\max}$

where $u_{\max}$ is the maximum value of the analytic solution for the water velocity, provided by FullSWOF_1D.

Hyperparameters. For the training is used an Adam optimizer and a fixed learning rate of 2×10^{-4}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer.

The training points in the domain, on the boundary and for the IC are 10^4, 500 and 500, respectively. The number of iterations is 5×10^5.

The loss function is made up of seven terms: ($\mathcal{L}^h_{PDE}$, $\mathcal{L}^{hu}_{PDE}$, $\mathcal{L}^h_{ineq}$, $\mathcal{L}^u_{ineq}$ $\mathcal{L}^h_{IC}$, $\mathcal{L}^u_{IC}$, $\mathcal{L}^u_{BC}$), respectively generated by the residuals of the mass conservation and of the momentum balance equations, the inequalities and the ICs for the variables h and u, and the BC for the variable u. Every single term enters the loss function with an associated weight that proved to work fine after a trial-and-error effort:

$$\begin{aligned}\mathcal{L} = \mathbf{10^2}\mathcal{L}^{h}_{PDE} + \mathbf{10^2}\mathcal{L}^{hu}_{PDE} \\ + \mathbf{10^5}\mathcal{L}^{h}_{ineq} + \mathbf{10^5}\mathcal{L}^{u}_{ineq} \\ + \mathbf{10^5}\mathcal{L}^{h}_{IC} + \mathbf{10^5}\mathcal{L}^{u}_{IC} + \mathbf{10^5}\mathcal{L}^{u}_{BC}\end{aligned} \tag{4.26}$$

The results of the learning of *all* the variables, h, u and q, provided with the errors of the PINN's approximations computed in norm $L^2(\Omega)$, are shown in Fig. 4.12 on page 82. Specifically, the results for the flow are built after the learning, applying the relation: $q = hu$.

The integral errors account for a good numerical approximation. Examining the plots, the PINN's prediction looks to be comparable to the exact—challenging—solution more than the FullSWOF_1D's approximation does. Specifically, the position of the shock seems to be predicted considerably better.

4.8 Dam break on a dry domain with friction

Several methods investigating this case may be found in the literature, though each one comes with a limitation. For instance, the Dressler's approach [9] provides no information concerning the shape of the wave tip. The problem is defined in the domain $\Omega \times (0, T)$ with $\Omega = (x_{\min}, x_{\max})$, considering $x \in (0, 2000)$ and $t \in (0, 40)$. The dam break is instantaneous, the bottom is flat and the Chézy's friction coefficient is set to 40.

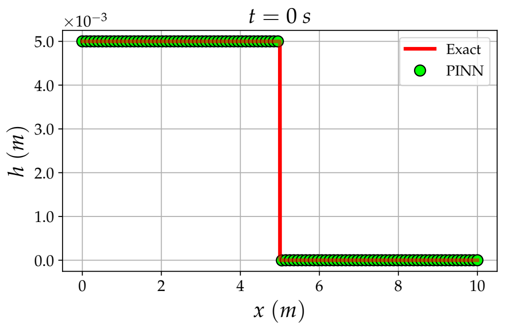
The initial conditions are:

$$h(x,0) = \begin{cases} 6 \text{ m} & x \leq 1000 \text{ m} \\ 0 \text{ m} & x > 1000 \text{ m} \end{cases} \tag{4.27}$$

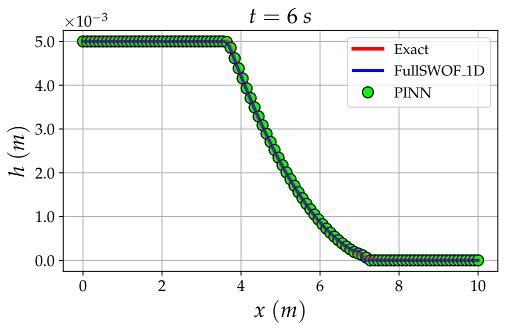
and

$$u = 0 \text{ m/s} \tag{4.28}$$

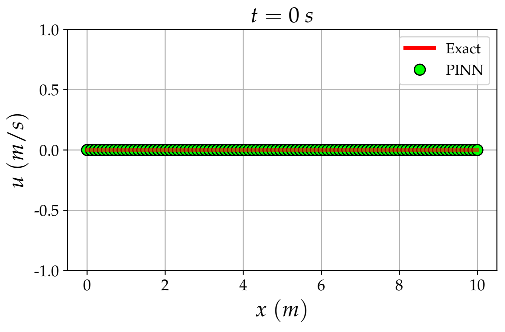
whereas no boundary condition is required to be set.



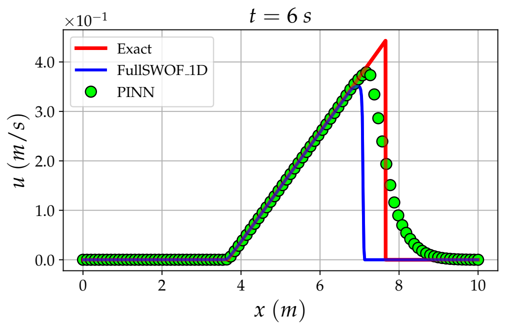
(a) $\frac{\|h_{ex}-h_{PINN}\|_{L^2}}{\|h_{ex}\|_{L^2}} = 2.181 \times 10^{-3}$



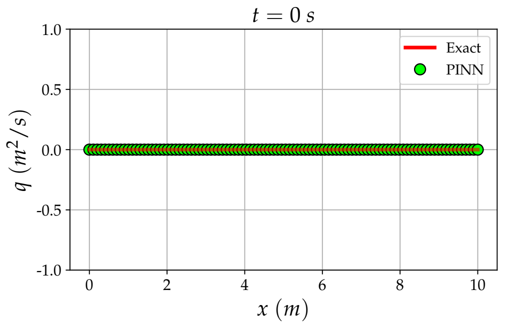
(b) $\frac{\|h_{ex}-h_{PINN}\|_{L^2}}{\|h_{ex}\|_{L^2}} = 2.869 \times 10^{-5}$



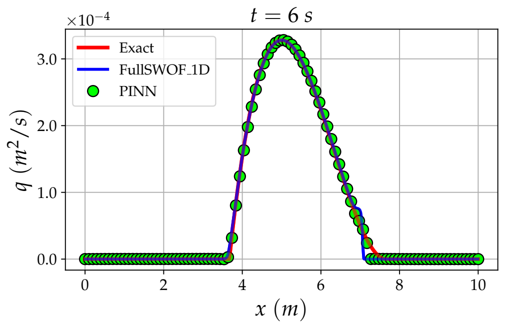
(c) $\|u_{ex} - u_{PINN}\|_{L^2} = 1.948 \times 10^{-4}$



(d) $\frac{\|u_{ex}-u_{PINN}\|_{L^2}}{\|u_{ex}\|_{L^2}} = 6.279 \times 10^{-2}$



(e) $\|q_{ex} - q_{PINN}\|_{L^2} = 1.804 \times 10^{-9}$



(f) $\frac{\|q_{ex}-q_{PINN}\|_{L^2}}{\|q_{ex}\|_{L^2}} = 4.887 \times 10^{-4}$

Figure 4.12: *Dam break on a dry domain. PINN solutions for **(a)**-**(b)** the water height, **(c)**-**(d)** the velocity, **(e)**-**(f)** the flow. Comparison with the FullSWOF_1D results. Errors computed in norm $L^2(\Omega)$.*

The SWE are implemented holding as unknowns h and u, plus, two supplementary inequalities are included in the system for the purpose of making the learning faster. The complete system is given in Eq. (4.29).

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial (hu)}{\partial x} = 0 \\ \dfrac{\partial (hu)}{\partial t} + \dfrac{\partial (hu^2 + \frac{1}{2}gh^2)}{\partial x} = -g\dfrac{u|\mathbf{u}|}{40^2} \\ h \geq 0 \\ u \geq 0 \end{cases} \tag{4.29}$$

Model setup. Considering the large spatio-temporal domain, the model *has* to be scaled. Specifically, the reference values for the scaling are set as:

- $\mathbf{L} = 2000$
- $\mathbf{H} = 6$
- $\mathbf{T} = 40$
- $\mathbf{U} = u_{\max}$

where $u_{\max}$ is the maximum value of the analytic solution for the water velocity, provided by FullSWOF_1D.

Hyperparameters. For the training is used an Adam optimizer and a fixed learning rate of 2×10^{-5}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer.

The training points in the domain and for the IC are 5×10^3 and 500, respectively. No training point is set on the boundary. The number of iterations is 1.2×10^6.

The loss function is made up of six terms: ($\mathcal{L}^h_{PDE}$, $\mathcal{L}^{hu}_{PDE}$, $\mathcal{L}^h_{ineq}$, $\mathcal{L}^u_{ineq}$ $\mathcal{L}^h_{IC}$, $\mathcal{L}^u_{IC}$), respectively generated by the residuals of the mass conservation and of the momentum balance equations, the inequalities and the ICs for the variables h and u. Every single term enters the loss function with an associated weight that proved to work fine after a trial-and-error effort:

$$\begin{aligned}\mathcal{L} = {} & \mathbf{10^2}\mathcal{L}^h_{PDE} + \mathbf{10^2}\mathcal{L}^{hu}_{PDE} \\ & + \mathbf{10^5}\mathcal{L}^h_{ineq} + \mathbf{10^5}\mathcal{L}^u_{ineq} \\ & + \mathbf{10^5}\mathcal{L}^h_{IC} + \mathbf{10^5}\mathcal{L}^u_{IC}\end{aligned} \tag{4.30}$$

The results of the learning of *all* the variables, h, u and q, provided with the errors of the PINN's approximations computed in norm $L^2(\Omega)$, are shown in Fig. 4.13 on page 85. Specifically, the results for the flow are built after the learning, applying the relation: $q = hu$.

The integral errors—obtainable only for the IC—account for a good numerical approximation. Examining the plots, the PINN's predictions mainly fit the the FullSWOF_1D's approximation. Moreover, the position of the shock seems to be identified just about perfectly.

4.9 Dam break on a dry domain with friction, *parametric*

This case is built upon the previous one, analyzed in Section 4.8. The problem is defined in the domain $\Omega \times (0, T)$, this time with:

$$\Omega = (x_{\min}, x_{\max}) \times (y_{\min}, y_{\max}) \tag{4.31}$$

considering $x \in (0, 2000)$, $y \in (0, 1)$ and $t \in (0, 40)$. The additional variable y accounts for the parameter's space. Still, the dam break is instantaneous and the bottom is flat.

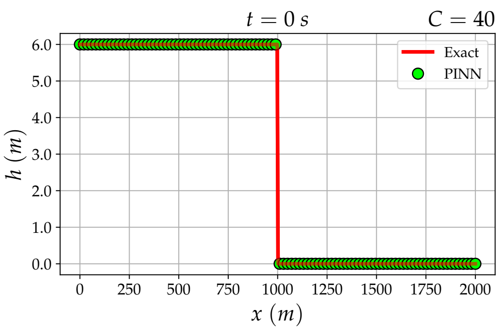
The initial conditions are:

$$h(x, y, 0) = \begin{cases} 6 \text{ m} & x \leq 1000 \text{ m} \\ 0 \text{ m} & x > 1000 \text{ m} \end{cases} \tag{4.32}$$

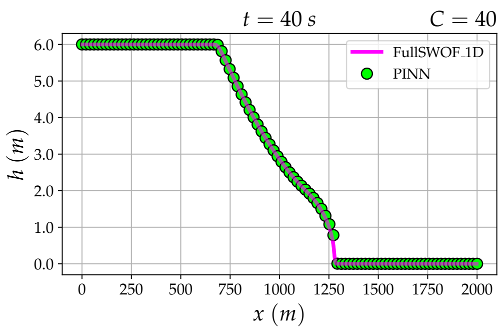
and

$$u = 0 \text{ m/s} \tag{4.33}$$

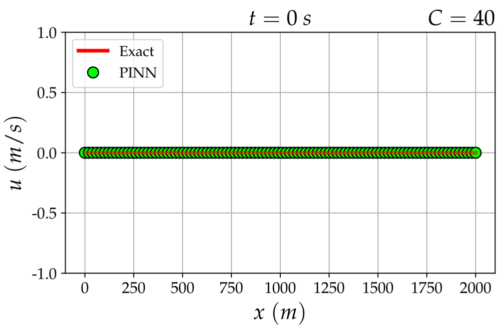
whereas no boundary condition is required to be set.



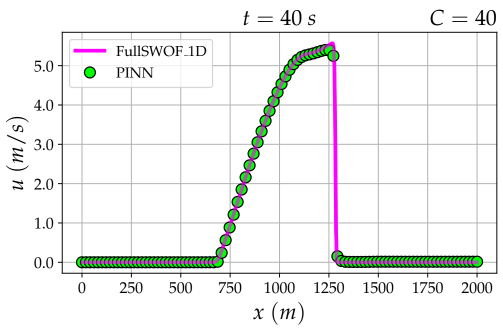
(a) $\frac{\|h_{ex} - h_{PINN}\|_{L^2}}{\|h_{ex}\|_{L^2}} = 9.404 \times 10^{-6}$



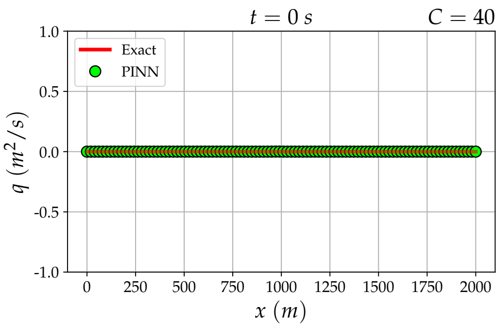
(b)



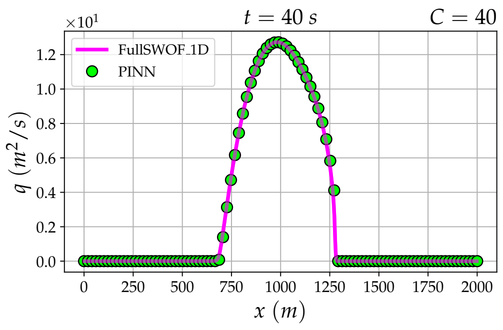
(c) $\|u_{ex} - u_{PINN}\|_{L^2} = 2.218 \times 10^{-3}$



(d)



(e) $\|q_{ex} - q_{PINN}\|_{L^2} = 5.638 \times 10^{-3}$



(f)

Figure 4.13: *Dam break on a dry domain with friction. PINN solutions for **(a)**-**(b)** the water height, **(c)**-**(d)** the velocity, **(e)**-**(f)** the flow. Comparison with the FullSWOF_1D results. Errors computed in norm $L^2(\Omega)$.*

The additional parameter's space is integrated in the mathematical formulation by way of a linear transformation between the variable y and the Chézy's friction coefficient, C, actually entering the PDE:

$$C = 100y + 20 \tag{4.34}$$

This technique, already used in Section 4.6, is used with the purpose of keeping the parameter's space dimension unitary, while obtaining a Chézy's friction coefficient ranging extensively from the values 20 and 120.

The SWE are implemented holding as unknowns h and u, plus, two supplementary inequalities are included in the system for the purpose of making the learning faster. The complete system is given in Eq. (4.35).

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial (hu)}{\partial x} = 0 \\ \dfrac{\partial (hu)}{\partial t} + \dfrac{\partial (hu^2 + \frac{1}{2}gh^2)}{\partial x} = -g\dfrac{u|\mathbf{u}|}{C^2} \\ h \geq 0 \\ u \geq 0 \end{cases} \tag{4.35}$$

Model setup. Considering the large spatio-temporal domain, the model *has* to be scaled. Specifically, the reference values for the scaling are set as:

- $\mathbf{L} = 2000$
- $\mathbf{T} = 40$
- $\mathbf{H} = 6$
- $\mathbf{U} = u_{\mathrm{max}}^{120}$

where u_{max}^{120} is the maximum value of the numerical solution for the water velocity, considering $C = 120$, computed by FullSWOF_1D.

Hyperparameters. For the training is used an Adam optimizer and a fixed learning rate of 10^{-4}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 60 neurons per layer.

The training points in the domain and for the IC are 2×10^4 and 10^3, respectively. No training point is set on the boundary. The number of iterations is 5×10^5.

The loss function is made up of six terms: ($\mathcal{L}^h_{PDE}$, $\mathcal{L}^{hu}_{PDE}$, $\mathcal{L}^h_{ineq}$, $\mathcal{L}^u_{ineq}$ $\mathcal{L}^h_{IC}$, $\mathcal{L}^u_{IC}$), respectively generated by the residuals of the mass conservation and of the momentum balance equations, the inequalities and the ICs for the variables h and u. Every single term enters the loss function with an associated weight that proved to work fine after a trial-and-error effort:

$$\begin{aligned} \mathcal{L} = \mathbf{10^2}\mathcal{L}^h_{PDE} + \mathbf{10^2}\mathcal{L}^{hu}_{PDE} + \mathbf{10^5}\mathcal{L}^h_{ineq} \\ + \mathbf{10^5}\mathcal{L}^u_{ineq} + \mathbf{10^5}\mathcal{L}^h_{IC} + \mathbf{10^5}\mathcal{L}^u_{IC} \end{aligned} \tag{4.36}$$

The results of the learning of *all* the variables, h, u and q, are shown in Figures. 4.14, 4.15 and 4.16, respectively. Specifically, the results for the flow are built after the learning, applying the relation: $q = hu$. No error can be computed since no analytic solution is available.

Examining the plots, the PINN's predictions are a little bit smoother than the FullSWOF_1D approximation. However, the features of this case study—particularly, the positions of the shocks—appear to be identified almost perfectly over the whole wide-ranging parameter's space.

4.10 Circular dam break

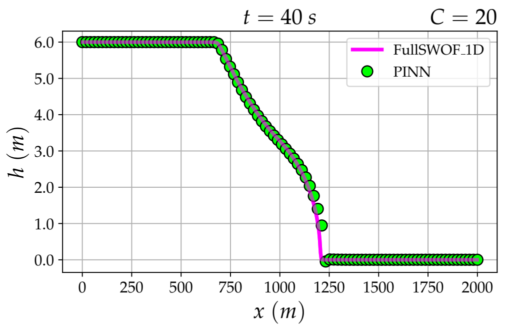
The problem is defined in the domain $\Omega \times (0, T)$ with:

$$\Omega = (x_{\min}, x_{\max}) \times (y_{\min}, y_{\max}) \tag{4.37}$$

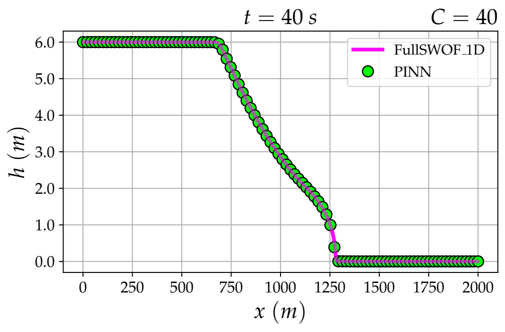
considering $x \in (-25, 25)$, $y \in (-25, 25)$ and $t \in (0, 0.69)$. The dam break is instantaneous and the bottom is flat.

The initial conditions are:

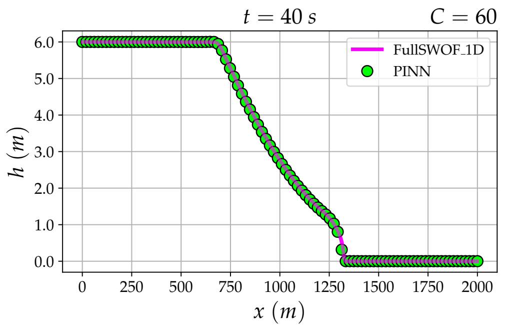
$$h(x, y, 0) = \begin{cases} 10 \ \ \mathrm{m} & |x| \leq \sqrt{100 - y^2}, |y| \leq \sqrt{100 - x^2} \\ 1 \ \ \mathrm{m} & \text{otherwise} \end{cases} \tag{4.38}$$



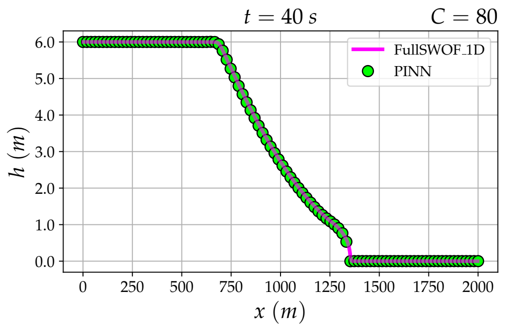
(a)



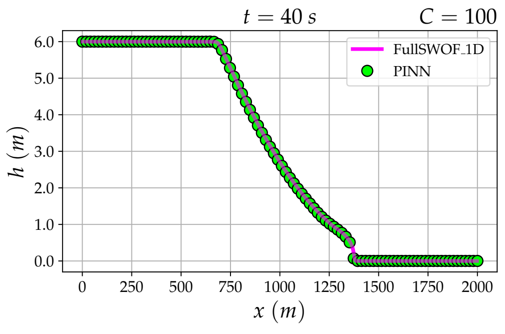
(b)



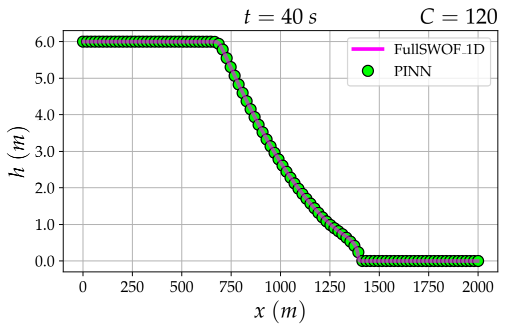
(c)



(d)

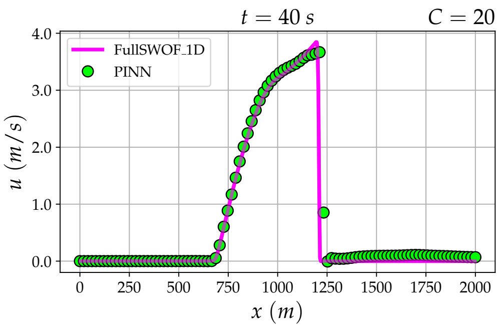


(e)

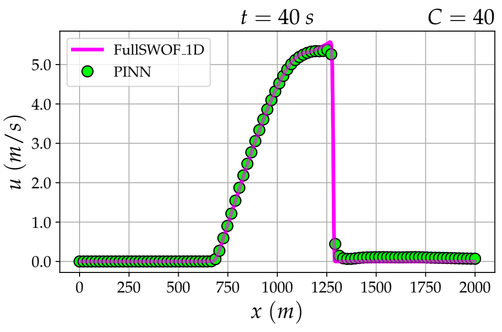


(f)

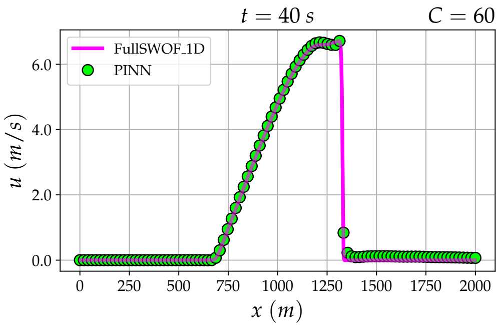
Figure 4.14: *Dam break on a dry domain with friction, parametric. PINN solutions for the water height. The parameter, C, ranges over the entire space. Comparison with the FullSWOF_1D results.*



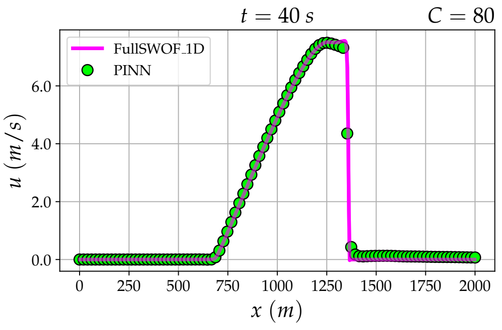
(a)



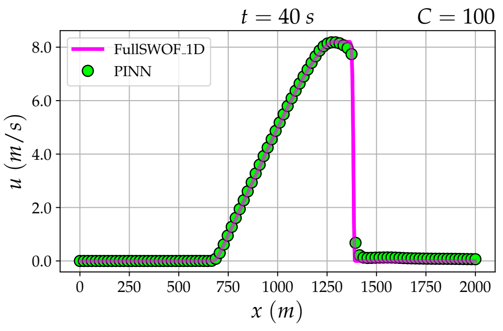
(b)



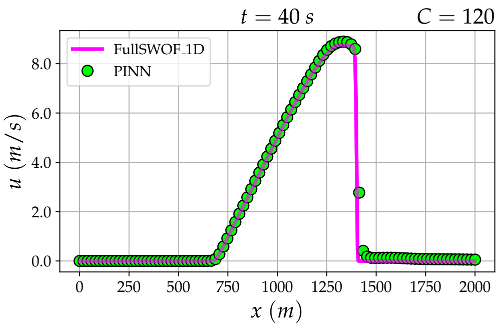
(c)



(d)

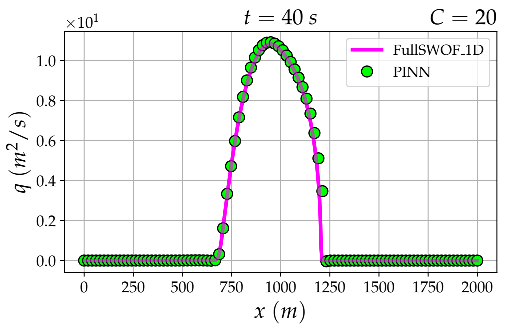


(e)

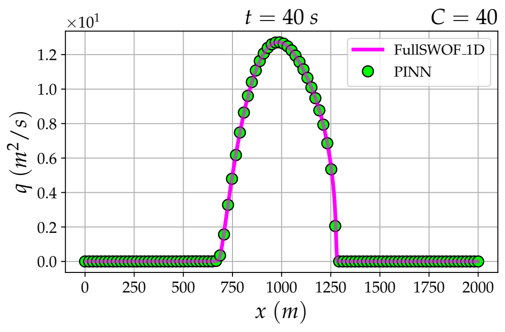


(f)

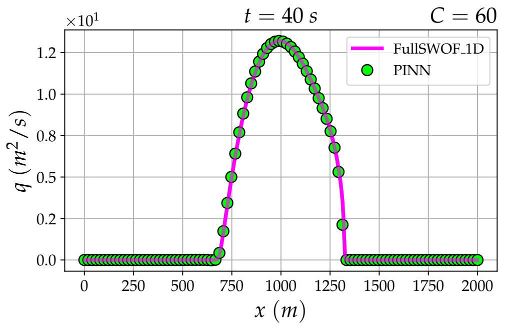
Figure 4.15: *Dam break on a dry domain with friction, parametric. PINN solutions for the water velocity. The parameter, C, ranges over the entire space. Comparison with the FullSWOF_1D results.*



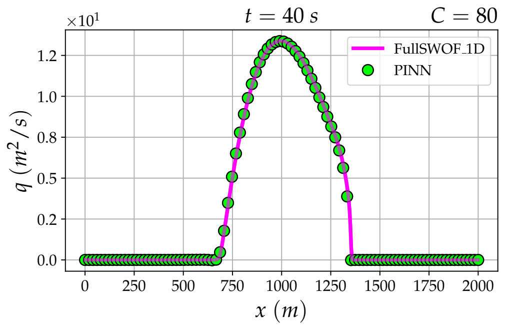
(a)



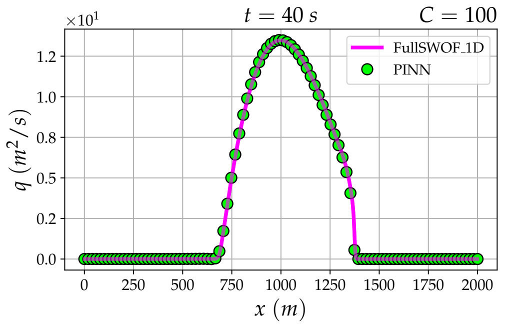
(b)



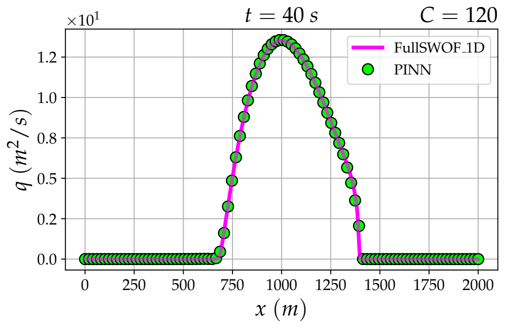
(c)



(d)



(e)



(f)

Figure 4.16: *Dam break on a dry domain with friction, parametric. PINN solutions for the water flow. The parameter, C, ranges over the entire space. Comparison with the FullSWOF_1D results.*

and

$$u(x,y,0) = 0 \text{ m/s} \tag{4.39}$$
$$v(x,y,0) = 0 \text{ m/s} \tag{4.40}$$

while the boundary conditions are:

$$u(x_{\text{min}},y,t) = 0 \text{ m/s} \tag{4.41}$$
$$u(x_{\text{max}},y,t) = 0 \text{ m/s} \tag{4.42}$$

and

$$v(x,y_{\text{min}},t) = 0 \text{ m/s} \tag{4.43}$$
$$v(x,y_{\text{max}},t) = 0 \text{ m/s} \tag{4.44}$$

The representation of the initial condition is shown in Fig. 4.17.

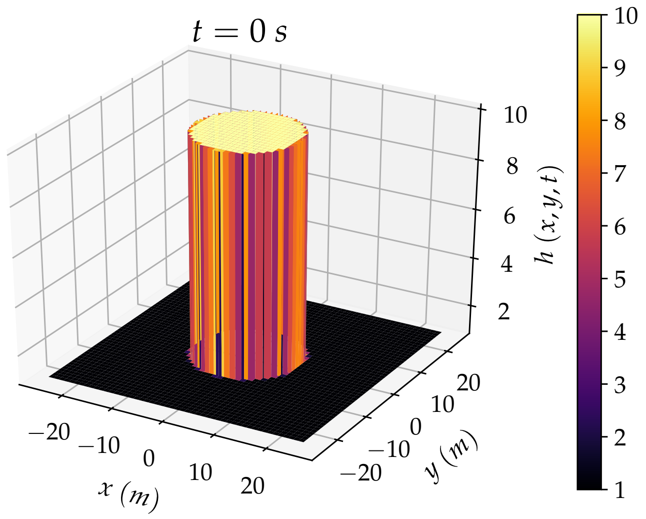


Figure 4.17: *Circular dam break. Initial condition.*

The SWE are implemented holding as unknowns h, u and v, plus, one supplementary inequality is included in the system for the purpose of making the learning faster. The complete system is given in Eq. (4.45).

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial (hu)}{\partial x} + \dfrac{\partial (hv)}{\partial y} = 0 \\ \dfrac{\partial (hu)}{\partial t} + \dfrac{\partial (hu^2 + \frac{1}{2}gh^2)}{\partial x} + \dfrac{\partial (huv)}{\partial y} = 0 \\ \dfrac{\partial (hv)}{\partial t} + \dfrac{\partial (huv)}{\partial x} + \dfrac{\partial (hv^2 + \frac{1}{2}gh^2)}{\partial y} = 0 \\ h \geq 1 \end{cases} \tag{4.45}$$

Remark. In spite of the large spatial domain, the model is not scaled.

Hyperparameters. For the training are used, sequentially, Adam and L-BFGS [24] optimizers. The learning rate is fixed at 10^{-4}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 50 neurons per layer.

The training points in the domain, on the boundary and for the IC are 2.5×10^4, 10^3 and 10^4, respectively. The number of iterations is 2×10^5 with the Adam optimizer and 8×10^4 with the L-BFGS.

The loss function is made up of eleven terms: ($\mathcal{L}^h_{PDE}$, $\mathcal{L}^{hu}_{PDE}$, $\mathcal{L}^{hv}_{PDE}$, $\mathcal{L}^h_{ineq}$, $\mathcal{L}^h_{IC}$, $\mathcal{L}^u_{IC}$, $\mathcal{L}^v_{IC}$, $\mathcal{L}^u_{BC_{x_{\min}}}$, $\mathcal{L}^u_{BC_{x_{\max}}}$, $\mathcal{L}^v_{BC_{y_{\min}}}$, $\mathcal{L}^v_{BC_{y_{\max}}}$), respectively generated by the residuals of the mass conservation and of two momentum balance equations, the inequality, the three ICs and the four BCs. Every term enters the loss function with an associated weight set to 1:

$$\begin{aligned} \mathcal{L} = &\mathbf{1}\mathcal{L}^h_{PDE} + \mathbf{1}\mathcal{L}^{hu}_{PDE} + \mathbf{1}\mathcal{L}^{hv}_{PDE} + \mathbf{1}\mathcal{L}^h_{ineq} \\ &+ \mathbf{1}\mathcal{L}^h_{IC} + \mathbf{1}\mathcal{L}^u_{IC} + \mathbf{1}\mathcal{L}^v_{IC} \\ &+ \mathbf{1}\mathcal{L}^u_{BC_{x_{\min}}} + \mathbf{1}\mathcal{L}^u_{BC_{x_{\max}}} + \mathbf{1}\mathcal{L}^v_{BC_{y_{\min}}} + \mathbf{1}\mathcal{L}^v_{BC_{y_{\max}}} \end{aligned} \tag{4.46}$$

The results of the learning are shown in Figures. 4.18 and 4.19, on page 93. Examining the plots, the features of this case study seem to be replicated pretty well. Specifically, the symmetry of the problem is almost totally preserved.

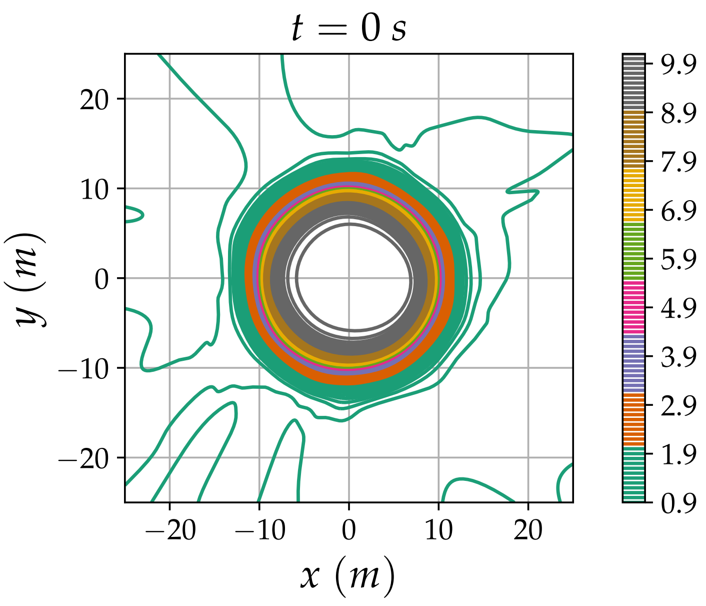


Figure 4.18: *Circular dam break. PINN solution for the water height at time $t = 0$ s.*

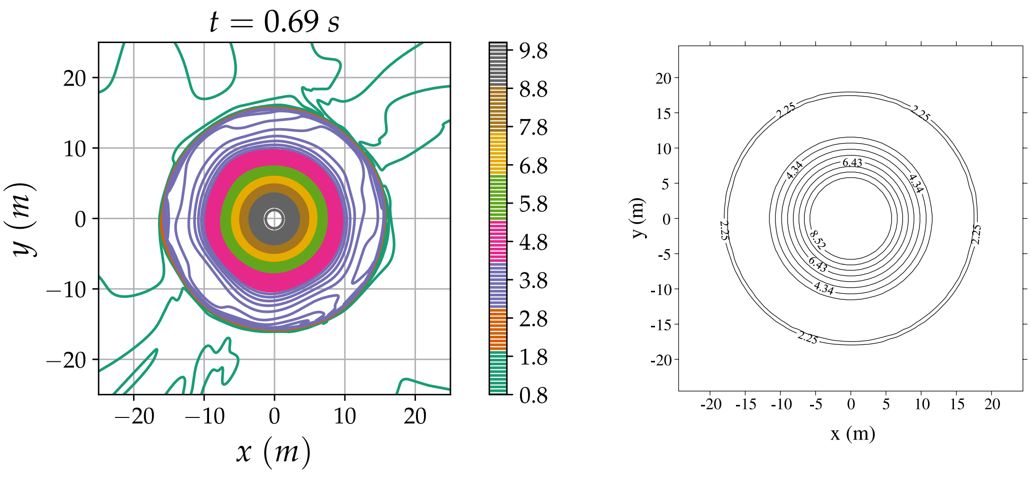


Figure 4.19: *Circular dam break. Solution for the water height at time $t = 0.69$ s: comparison between the PINN solution* ***(left)****, and the LSS scheme's [23] one* ***(right)****.*

4.11 Circular dam break, *parametric*

This case is built upon the previous one, analyzed in Section 4.10. The problem is defined in the domain $\Omega \times (0, T)$, this time with:

$$\Omega = (x_{\text{min}}, x_{\text{max}}) \times (y_{\text{min}}, y_{\text{max}}) \times (z_{\text{min}}, z_{\text{max}}) \tag{4.47}$$

considering $x \in (-25, 25)$, $y \in (-25, 25)$, $z \in (1, 20)$ and $t \in (0, 1)$. The additional variable z accounts for the parameter's space. Still, the dam break is instantaneous and the bottom is flat.

The initial conditions are:

$$h(x,y,z,0) = \begin{cases} 10 \ \text{ m} & |x| \le \sqrt{100 - y^2}, |y| \le \sqrt{100 - x^2} \\ 1 \ \text{ m} & \text{otherwise} \end{cases} \tag{4.48}$$

and

$$u(x,y,z,0) = 0 \text{ m/s} \tag{4.49}$$
$$v(x,y,z,0) = 0 \text{ m/s} \tag{4.50}$$

while the boundary conditions are:

$$u(x_{\text{min}}, y, z, t) = 0 \text{ m/s} \tag{4.51}$$
$$u(x_{\text{max}}, y, z, t) = 0 \text{ m/s} \tag{4.52}$$

and

$$v(x, y_{\text{min}}, z, t) = 0 \text{ m/s} \tag{4.53}$$
$$v(x, y_{\text{max}}, z, t) = 0 \text{ m/s} \tag{4.54}$$

The representation of the initial condition is shown in Fig. 4.17 on page 91.

The SWE are implemented holding as unknowns h, u and v. The gravitational acceleration, g, enters the PDE as a wide-ranging parameter belonging to the third spatial axis of the problem, i.e., z. The complete system is given in Eq. (4.55).

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial (hu)}{\partial x} + \dfrac{\partial (hv)}{\partial y} = 0 \\ \dfrac{\partial (hu)}{\partial t} + \dfrac{\partial (hu^2 + \frac{1}{2} z h^2)}{\partial x} + \dfrac{\partial (huv)}{\partial y} = 0 \\ \dfrac{\partial (hv)}{\partial t} + \dfrac{\partial (huv)}{\partial x} + \dfrac{\partial (hv^2 + \frac{1}{2} z h^2)}{\partial y} = 0 \end{cases} \tag{4.55}$$

Remark. In spite of the large domain, the model is not scaled.

Hyperparameters. For the training are used, sequentially, Adam and L-BFGS optimizers. The learning rate is fixed at 10^{-3}, the hyperbolic tangent is used as activation function, and the network architecture is set to 4 hidden layers with 30 neurons per layer.

The training points in the domain, on the boundary and for the IC are 1.2×10^4, 5×10^3 and 1.2×10^4, respectively. The number of iterations is 3×10^5 with the Adam optimizer and 2×10^5 with the L-BFGS.

The loss function is made up of ten terms: ($\mathcal{L}^h_{PDE}$, $\mathcal{L}^{hu}_{PDE}$, $\mathcal{L}^{hv}_{PDE}$, $\mathcal{L}^h_{IC}$, $\mathcal{L}^u_{IC}$, $\mathcal{L}^v_{IC}$, $\mathcal{L}^u_{BC_{x_{\min}}}$, $\mathcal{L}^u_{BC_{x_{\max}}}$, $\mathcal{L}^v_{BC_{y_{\min}}}$, $\mathcal{L}^v_{BC_{y_{\max}}}$), respectively generated by the residuals of the mass conservation and of two momentum balance equations, the three ICs and the four BCs. Every term enters the loss function with an associated weight set to 1:

$$\begin{aligned} \mathcal{L} = \mathbf{1}\mathcal{L}^h_{PDE} + \mathbf{1}\mathcal{L}^{hu}_{PDE} + \mathbf{1}\mathcal{L}^{hv}_{PDE} \\ + \mathbf{1}\mathcal{L}^h_{IC} + \mathbf{1}\mathcal{L}^u_{IC} + \mathbf{1}\mathcal{L}^v_{IC} \\ + \mathbf{1}\mathcal{L}^u_{BC_{x_{\min}}} + \mathbf{1}\mathcal{L}^u_{BC_{x_{\max}}} + \mathbf{1}\mathcal{L}^v_{BC_{y_{\min}}} + \mathbf{1}\mathcal{L}^v_{BC_{y_{\max}}} \end{aligned} \tag{4.56}$$

The results of the learning are shown in Fig. 4.20 on page 96. Examining the plots, the features of this case study seem to be replicated pretty well. Specifically, the PINN predictions preserve the symmetry of the problem over most of the wide-ranging parameter's space.

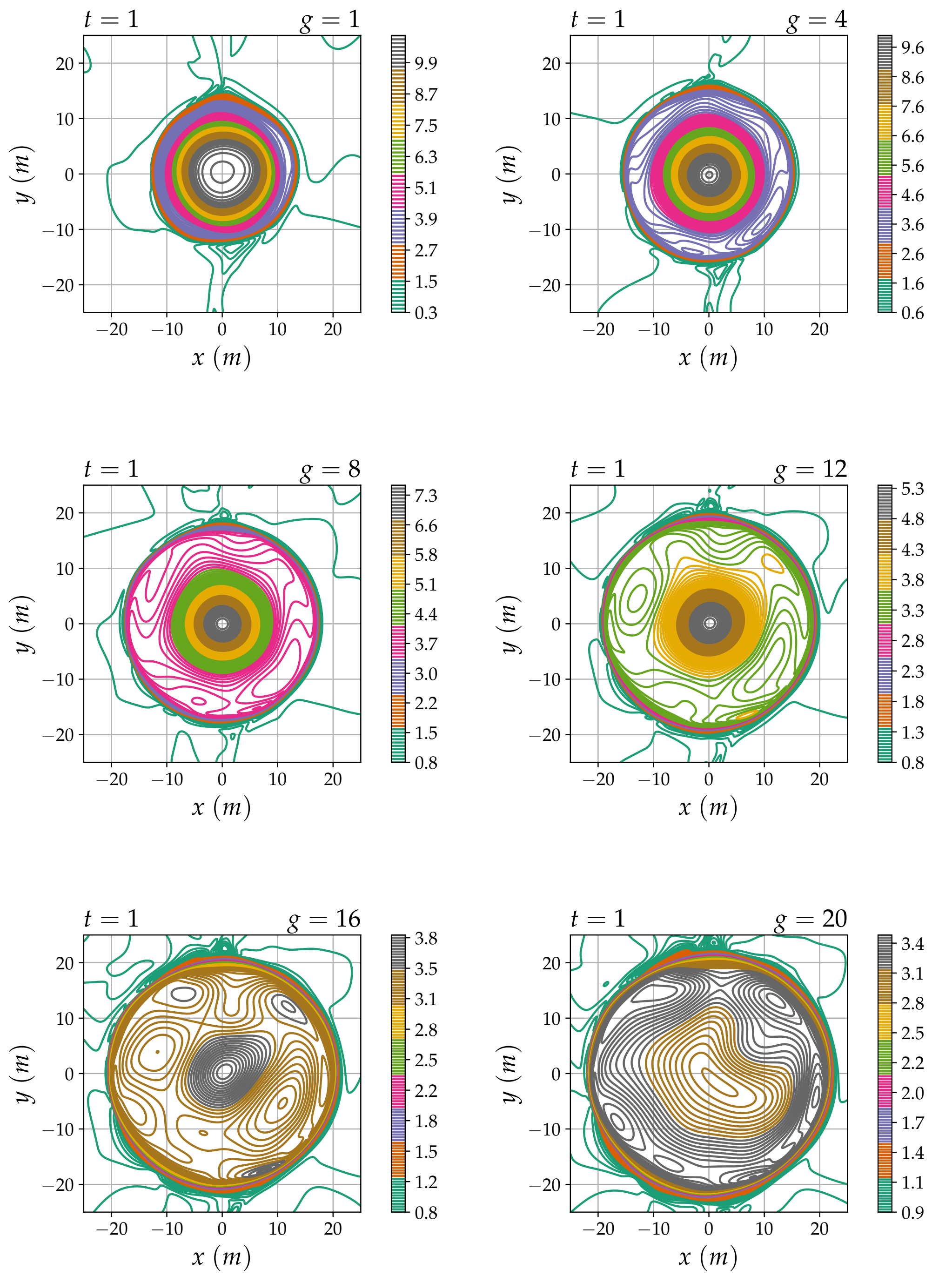


Figure 4.20: *Circular dam break, parametric. PINN solutions for the water height. The parameter, g, ranges over the entire space.*

Conclusions

In this work the PINN technology was successfully applied to some preliminary elliptic, parabolic and hyperbolic models, then, to a set of hyperbolic partial differential equations: the inviscid Shallow Water Equations (SWE). The purpose of this thesis was to examine the degree to which the PINN is actually a convenient tool to be used. Particularly, the effectiveness of this technology when dealing with parametric problems has been investigated.

Multiple points *in favor* of the PINN application have been identified. First of all, the accuracy of the numerical solutions produced is solid: the integral errors, as well as the qualitative inspection of the results, gave evidence of an extremely accurate tool. The PINN is able to learn, with considerable precision, analytic solutions known in the literature. Moreover, the method is totally agnostic to the presence of viscosity or shock waves.

Dealing with *parametric* problems, the PINNs shown their suitability: once that the network was trained over the needed parameter space, every single recall of the model for the requested parameter value was immediate. This happened in light of the fact that these recalls were only evaluations of a trained network.

PINNs proved to work well when the dimensions of the problems were increased. There is no mesh to be built. One more dimension for the problem domain translates into one more input-layer neuron, one more dimension for the problem solution translates into one more output-layer neuron. *Prospectively*, this characteristic of the PINN could bypass the limit of dimensionality of the traditional numerical discretization methods [29].

As would be expected, also some factors that could be counted *against* the use of PINNs have been experienced. Unquestionably, the computational cost of the PINN's training is considerably high: the average time required for the training of every single model examined has been of about four hours, running a NVIDIA® T4 GPU [39]. Plus, it's hard to overlook the fact that the onerous training phase comes only after a first—even more demanding—step, where the choices of the hyperparameters' values, and of the proper scaling of the model, have to be made. It's a tough trial-and-error task, regularly way longer than the following training phase.

Even taking into consideration only the simpler elliptic and parabolic models, PINNs are currently not a believable alternative to Finite Element Method (FEM), Finite Volume Method (FVM) or Finite Difference Method (FDM): the computational cost—let alone the drawn out troubles to set the fitting model's hyperparameters values—of a PINN implementation is so much higher than the three traditional methods' one, that daring a comparison today is not even fair.

But those traditional methods take advantage of a 50-years-long process of development. Putting things in perspective, since research in the neural network field is very active, great improvements are expected. In order to reduce the computational cost of the training, the clustered residual points distribution [26]—a smaller set of points placed were they matter the most, without *a priori* knowledge of the solution—even for the parametric cases, would definitely help. Concerning the design of the most effective neural network architecture for the problem at hand, there's still experience to be accumulated before it ceases to be done empirically by the user.

Bibliography

[1] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, J.M. Siskind, *Automatic Differentiation in Machine Learning: a Survey*, Journal of Machine Learning Research, 18(153):1–43, 2018.

[2] A.L. Blum, R.L. Rivest, *Training a 3-node neural network is NP-complete*, Neural Networks, 5(1):117–127, 1992.

[3] L. Bottou, O. Bousquet, *The Tradeoffs of Large Scale Learning*, Advances in Neural Information Processing Systems, 161–168, 2008.

[4] A. Burkov, *The Hundred-Page Machine Learning Book*, Andriy Burkov, 2019.

[5] O. Delestre, *Simulation du ruissellement d'eau de pluie sur des surfaces agricoles*, PhD thesis, Université d'Orléans, 2010.

[6] O. Delestre, F. Darboux, F. James, C. Lucas, C. Laguerre, S. Cordier, *FullSWOF: Full Shallow-Water equations for Overland Flow*, Journal of Open Source Software, 2(20):448, 2017.

[7] O. Delestre, C. Lucas, P.A. Ksinant, F. Darboux, C. Laguerre, T.N.T. Vo, F. James, S. Cordier, *SWASHES: a compilation of Shallow Water Analytic Solutions for Hydraulic and Environmental Studies*, International Journal for Numerical Methods in Fluids, 72(3):269–300, 2013.

[8] M. W. M. G. Dissanayake, N. Phan-Thien, *Neural-network-based approximations for solving partial differential equations*, Communications in Numerical Methods in Engineering, 10(3):195–201, 1994.

[9] R.F. DRESSLER, *Hydraulic resistance effect upon the dam-break functions*, Journal of Research of the National Bureau of Standards, 49(3):217–225, 1952.

[10] S. DUTTA, *Reinforcement Learning with TensorFlow*, Packt, 2018.

[11] D. FERRARESE, *Metodi a volumi finiti centrati well balanced per la soluzione delle equazioni delle Shallow Water*, MSc thesis, Politecnico di Milano, 2010.

[12] Y. GOLDBERG, *A Primer on Neural Network Models for Natural Language Processing*, Journal of Artificial Intelligence Research, 57:345–420, 2016.

[13] N. GOUTAL, F. MAUREL, *Proceedings of the 2^{nd} workshop on dam-break wave simulation*, Electricité de France, Direction des études et recherches, Technical Report HE-43/97/016/B, 1997.

[14] W.H. GREEN, G.A. AMPT, *Studies on soil physics*, The Journal of Agricultural Science, 4(1):1–24 , 1911.

[15] E. GRIMSON, J. GUTTAG, A. BELL, *6.0002 Introduction to Computational Thinking and Data Science*, MIT OpenCourseWare, `https://ocw.mit.edu`, 2016.

[16] K. HORNIK, M. STINCHCOMBE, H. WHITE, *Multilayer feedforward networks are universal approximators*, Neural Networks, 2(5):359–366, 1989.

[17] A. KARPATNE, G. ATLURI, J.H. FAGHMOUS, M. STEINBACH, A. BANERJEE, A. GANGULY, S. SHEKHAR, N. SAMATOVA, V. KUMAR, *Theory-Guided Data Science: A New Paradigm for Scientific Discovery from Data*, Institute of Electrical and Electronics Engineers, 29:2318–2331, 2017.

[18] D.P. KINGMA, J. BA, *Adam: A Method for Stochastic Optimization*, International Conference for Learning Representations, 2015.

[19] A. KRIZHEVSKY, I. SUTSKEVER, G.E. HINTON, *Imagenet classification with deep convolutional neural networks*, Advances in Neural Information Processing Systems, 25:1097–1105, 2012.

[20] B.M. LAKE, R. SALAKHUTDINOV, J.B. TENENBAUM, *Human-level concept learning through probabilistic program induction*, Science, 350:1332–1338, 2015.

[21] D. LAZER, R. KENNEDY, G. KING, A. VESPIGNANI, *The Parable of Google Flu: Traps in Big Data Analysis*, Science, 343:1203–1205, 2014.

[22] Y. LECUN, Y. BENGIO, G. HINTON, *Deep learning*, Nature, 521:436–444, 2015.

[23] G.F. LIN, J.S. LAI, W.D. GUO, *Finite-volume component-wise TVD schemes for 2D shallow water equations*, Advances in Water Resources, 26(8):861–873, 2003.

[24] D.C. LIU, J. NOCEDAL, *On the limited memory BFGS method for large scale optimization*, Mathematical Programming, 45(1):503–528, 1989.

[25] L. LU, X. MENG, Z. MAO, G.E. KARNIADAKIS, *DeepXDE: A deep learning library for solving differential equations*, arXiv:1907.04502v2, 2020.

[26] Z. MAO, A.D. JAGTAP, G.E. KARNIADAKIS, *Physics-informed neural networks for high-speed flows*, Computer Methods in Applied Mechanics and Engineering, 360:112789, 2020.

[27] G. MARCUS, E. DAVIS, *Eight (No, Nine!) Problems With Big Data*, The New York Times, sect. A, p. 23, Apr. 7, 2014.

[28] A. PINKUS, *Approximation theory of the MLP model in neural networks*, Acta Numerica, 8:143–195, 1999.

[29] T. POGGIO, H. MHASKAR, L. ROSASCO, B. MIRANDA, Q. LIAO, *Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review*, International Journal of Automation and Computing, 14:503–519, 2017.

[30] K. QIAN, A. MOHAMED, C. CLAUDEL, *Physics Informed Data Driven model for Flood Prediction: Application of Deep Learning in prediction of urban flood development*, arXiv, 2019.

[31] A. QUARTERONI, *Modellistica Numerica per Problemi Differenziali*, Springer, 2012.

[32] M. RAISSI, P. PERDIKARIS, G.E. KARNIADAKIS, *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*, Journal of Computational Physics, 378:686–707, 2019.

[33] A. RITTER, *Die Fortpflanzung der Wasserwellen*, Zeitschrift des Vereines Deuscher Ingenieure, 36(33):947–954, 1892.

[34] S. SKANSI, *Introduction to Deep Learning*, Springer International Publishing, 2018.

[35] J.J. STOKER, *Water Waves: The Mathematical Theory with Applications*, Pure and Applied Mathematics, Volume 4, 1957.

[36] G. STRANG, *18.065 Matrix Methods in Data Analysis, Signal Processing, and Machine Learning*, MIT OpenCourseWare, `https://ocw.mit.edu`, 2018.

[37] *DeepXDE*, Documentation, `https://github.com/lululxvi/deepxde`, 2022.

[38] *Google Colab*, Documentation, `https://colab.research.google.com`, 2022.

[39] *NVIDIA® T4*, Documentation, `https://www.nvidia.com/T4`, 2022.

[40] *Project Jupyter*, Documentation, `https://docs.jupyter.org`, 2022.

[41] *TensorFlow*, Documentation, `https://www.tensorflow.org`, 2022.

Appendix A

Source code

Every test case dealt with in this thesis is identified by its own Python code, written in a Jupyter [40] notebook available in Google Colaboratory. However, the main structure of all these codes is basically equivalent and is applied, *mutatis mutandis*, to each test case.

In this appendix, the sections that make up these codes are presented in sequential order. For the sake of clarity, the code is organized in cells, just as it is in a Jupyter environment.

A.1 Preamble

This section of code is shared by every test case. The needed libraries are made available to the Jupyter session, the double precision format is set, the access to the free-of-charge Google's computing resources that run in the cloud is guaranteed.

```
pip install deepxde -q
```

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

import matplotlib.pyplot as plt
import numpy             as np
import deepxde           as dde
import time              as time
```

```python
import matplotlib
import matplotlib.ticker as mtick

from matplotlib         import cm
from matplotlib.ticker import ScalarFormatter
from deepxde.backend   import tf
from google.colab      import output
from scipy             import interpolate
from scipy.interpolate import griddata
from numpy             import savetxt
from deepxde.callbacks import EarlyStopping

from google.colab      import files
from google.colab      import drive
```

```python
dde.config.real.set_float64()
```

```python
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```python
drive.mount('/content/drive')
```

A.2 Data import from FullSWOF_1D

Whether it is an analytic solution or a numerical one, the import procedure of the FullSWOF_1D's output is the same: the FullSWOF_1D's output—that comes in the form of a text file organized in columns—is read and suitably stored in the Jupyter session. The code that follows is taken from the case presented in Section 4.5.

```python
with open('/content/drive/My Drive/Colab_IN/SWASHES_411','r') as fh:
    read_data = fh.read().split()

num_colonne = 8
```

```
num_punti   = 10000
num_total   = np.int(num_colonne * num_punti)

swashes_x   = np.zeros([num_punti,1],dtype=np.float64)
swashes_h   = np.zeros([num_punti,1],dtype=np.float64)
swashes_u   = np.zeros([num_punti,1],dtype=np.float64)
swashes_z   = np.zeros([num_punti,1],dtype=np.float64)
swashes_q   = np.zeros([num_punti,1],dtype=np.float64)
swashes_zh  = np.zeros([num_punti,1],dtype=np.float64)
swashes_Fr  = np.zeros([num_punti,1],dtype=np.float64)
swashes_zhc = np.zeros([num_punti,1],dtype=np.float64)

i = 1
j = 1

while i < num_total:
    swashes_zhc[-j] = read_data[-i]
    i = i + 1
    swashes_Fr [-j] = read_data[-i]
    i = i + 1
    swashes_zh [-j] = read_data[-i]
    i = i + 1
    swashes_q  [-j] = read_data[-i]
    i = i + 1
    swashes_z  [-j] = read_data[-i]
    i = i + 1
    swashes_u  [-j] = read_data[-i]
    i = i + 1
    swashes_h  [-j] = read_data[-i]
    i = i + 1
    swashes_x  [-j] = read_data[-i]
    i = i + 1
    j = j + 1
```

A.3 Parameters definition and scaling

This section of code is devoted to the test case's parameters definition and their, potential, scaling. The code that follows is taken from the case presented in Section 4.6.

```
dim_input  = 3
dim_output = 2
```

```
Time  = 6

X_min = 0
X_max = 10
X_dam = 5

h_L = 5e-3
h_R = 1e-3

friction_MIN = 0
friction_MAX = 1

g_0 = 9.81
```

```
scale_L = 10
scale_T = 6
scale_U = np.max(swashes_u_120)
scale_H = 5e-3
```

```
Time  = Time  / scale_T

X_min = X_min / scale_L
X_max = X_max / scale_L
X_dam = X_dam / scale_L

h_L = h_L / scale_H
h_R = h_R / scale_H

g = g_0 * scale_H / (scale_U ** 2.0)
```

A.4 Functions definition and scaling

This section of code is devoted to the test case's functions definition and their, potential, scaling. The code that follows is taken from the case presented in Section 4.11.

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

```
def boundary_x1 (x, on_boundary):
    return on_boundary and np.isclose(x[0], X_min)

def boundary_x2 (x, on_boundary):
    return on_boundary and np.isclose(x[0], X_max)

def boundary_y1 (x, on_boundary):
    return on_boundary and np.isclose(x[1], Y_min)

def boundary_y2 (x, on_boundary):
    return on_boundary and np.isclose(x[1], Y_max)
```

```
def func_IC_h(x):
    return 10.0 * ((x[:, 0:1] - center_x) * (x[:, 0:1] - center_x) +  \
                   (x[:, 1:2] - center_y) * (x[:, 1:2] - center_y) <= \
                   (radius*radius)) / scale_H                      +  \
            1.0 * ((x[:, 0:1] - center_x) * (x[:, 0:1] - center_x) +  \
                   (x[:, 1:2] - center_y) * (x[:, 1:2] - center_y) >  \
                   (radius*radius)) / scale_H

def func_IC_u(x):
    return np.zeros([len(x), 1])

def func_IC_v(x):
    return np.zeros([len(x), 1])

def func_BC_all(x):
    return np.zeros([len(x), 1])
```

```python
def pde(x, y):

    g = x[:, 2:3] * scale_H / (scale_U ** 2.0)

    h = y[:, 0:1]
    u = y[:, 1:2]
    v = y[:, 2:3]

    U1  = h
    U2  = h * u
    U3  = h * v

    E1  = h * u
    E2  = h * u * u + 0.5 * h*h * g
    E3  = h * u * v

    G1  = h * v
    G2  = h * v * u
    G3  = h * v * v + 0.5 * h*h * g

    E1_x = tf.gradients(E1, x)[0][:, 0:1]
    E2_x = tf.gradients(E2, x)[0][:, 0:1]
    E3_x = tf.gradients(E3, x)[0][:, 0:1]

    G1_y = tf.gradients(G1, x)[0][:, 1:2]
    G2_y = tf.gradients(G2, x)[0][:, 1:2]
    G3_y = tf.gradients(G3, x)[0][:, 1:2]

    U1_t = tf.gradients(U1, x)[0][:, 3:4] * scale_L/(scale_T*scale_U)
    U2_t = tf.gradients(U2, x)[0][:, 3:4] * scale_L/(scale_T*scale_U)
    U3_t = tf.gradients(U3, x)[0][:, 3:4] * scale_L/(scale_T*scale_U)

    equaz_1 = U1_t + E1_x + G1_y
    equaz_2 = U2_t + E2_x + G2_y
    equaz_3 = U3_t + E3_x + G3_y

    return [equaz_1, equaz_2, equaz_3]
```

```python
geom       = dde.geometry.Cuboid([X_min, Y_min, G_min],
                                 [X_max, Y_max, G_max])
timedomain = dde.geometry.TimeDomain(0.0, Time)
geomtime   = dde.geometry.GeometryXTime(geom, timedomain)
```

```python
IC_h = dde.IC(geomtime, func_IC_h, on_initial, component = 0)
IC_u = dde.IC(geomtime, func_IC_u, on_initial, component = 1)
IC_v = dde.IC(geomtime, func_IC_v, on_initial, component = 2)

BC_u1 = dde.DirichletBC(geomtime, func_BC_all,
                        boundary_x1, component = 1)
BC_u2 = dde.DirichletBC(geomtime, func_BC_all,
                        boundary_x2, component = 1)
BC_v1 = dde.DirichletBC(geomtime, func_BC_all,
                        boundary_y1, component = 2)
BC_v2 = dde.DirichletBC(geomtime, func_BC_all,
                        boundary_y2, component = 2)

IC_BC = [IC_h, IC_u, IC_v, BC_u1, BC_u2, BC_v1, BC_v2]
```

A.5 Hard constraints

Whenever possible, it is massively convenient to enforce hard constraints for both the ICs and the BCs. The code that follows is taken from the case presented in Section 4.4: the hard constraints for the two ICs, and for one *impulsive* upstream BC, are all simultaneously defined. This function will be used, later in the code, to modify the NN's output.

```python
def Strong_Conditions(X,Y):

    x = X[:, 0:1]
    t = X[:, 1:2]

    h = Y[:, 0:1]
    q = Y[:, 1:2]

    c1 = tf.math.greater(x,  8.0)
    c2 = tf.math.less   (x, 12.0)
    c3 = tf.math.logical_and(c1,c2)

    f1 = 0.2 - 0.05 * (x - 10.0) ** 2.0
    f2 = tf.zeros_like(f1)
    f3 = tf.where(c3,f1,f2)

    G = tf.math.greater(t, 0)
    A = tf.zeros_like(t)
    B = tf.ones_like(t)
    C = tf.ones_like(t) * 4.42

    D = tf.where(G, B, A)
    E = tf.where(G, C, A)

    h_new = h * t + 2 - f3
    q_new = (q * x + E) * D

    return tf.concat((h_new, q_new), axis=1)
```

A.6 NN building and training

After all the physics is defined, it's time to define the training point, to build the NN and train it. The code that follows is taken from the case presented in Section 4.4. It includes both the resampling of the training points and the hard constraints—obtained modifying the NN's output.

```
data = dde.data.TimePDE(
    geomtime, pde, IC_BC,
    num_domain   = 4000,
    num_boundary = 2000,
    num_initial  = 1000)

net = dde.maps.FNN(
    layer_sizes        = [dim_input] + [60]*4 + [dim_output],
    activation         = "tanh",
    kernel_initializer = "Glorot uniform")

net.apply_output_transform(lambda x, y: Strong_Conditions(x,y))

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

```
my_path = "/content/drive/MyDrive/Colab_OUT/03_Bump_3/model.ckpt"
model.compile('adam', lr=0.001)
model.restore(my_path, verbose=1)

resampler = dde.callbacks.PDEResidualResampler(period=100)
```

```
checker   = dde.callbacks.ModelCheckpoint(
            "/content/drive/My Drive/Colab_OUT/03_Bump_3/model.ckpt",
            verbose = 1,
            save_better_only = True,
            period = 1000)

model.train(epochs=1000000, callbacks=[resampler,checker]),
            model_restore_path=my_path)
```

A.7 Output visualization

The final step is the visualization, and the save, of the results. In order to better harmonize the figures with this document, LaTeX and its fonts are made available to the Jupyter session. The code that follows is taken from the case presented in Section 4.6.

```
!apt install texlive-fonts-recommended texlive-fonts-extra cm-super
dvipng
```

```
matplotlib.rcParams['text.usetex'] = True
matplotlib.rcParams['font.family'] = "serif"
matplotlib.rcParams['font.serif']  = "Palatino"
matplotlib.rcParams['font.size']   = 13
```

```
tratto = '#ff00ff'
punto  = '#00FF00'
```

```
NN_Time  = Time
NN_Param = 120
NN_Param = (NN_Param - 20) / 100

x_sw = swashes_x_120
h_sw = swashes_h_120
u_sw = swashes_u_120
q_sw = swashes_q_120

```

```
N1  = 10000
N2  =   100

lw = 3
```

```
nPoints = N1

X_nn = swashes_x / scale_L
X_nn = np.reshape(X_nn, (len(X_nn), 1))
P_nn = np.ones_like(X_nn) * NN_Param
T_nn = np.ones_like(X_nn) * NN_Time
X    = np.hstack((X_nn, P_nn, T_nn))

U_nn = model.predict(X)
h_nn = U_nn[:,0].reshape(nPoints,1) * scale_H
u_nn = U_nn[:,1].reshape(nPoints,1) * scale_U
```

```
class ScalarFormatterForceFormat(ScalarFormatter):
    def _set_format(self):
        self.format = "%1.1f"

yfmt_0 = ScalarFormatterForceFormat()
yfmt_1 = ScalarFormatterForceFormat()
yfmt_2 = ScalarFormatterForceFormat()
yfmt_3 = ScalarFormatterForceFormat()

fig_0 = plt.figure(0)
ax_0  = fig_0.add_subplot()
plt.title(r'$C = {:}$'.format(np.int(NN_Param * 100 + 20)), fontsize=18,
loc='right')
plt.title(r'$t = {:}$'.format(np.int(NN_Time*scale_T)) + ' ' +
r'$s$',fontsize=18, loc='center')
plt.xlabel(r'$x$' + ' ' + r'$(m)$',fontsize=18)
plt.ylabel(r'$h$' + ' ' + r'$(m)$',fontsize=18)
ax_0.yaxis.set_major_formatter(yfmt_0)
ax_0.ticklabel_format(axis='y', style='sci', scilimits=(0,0))
```

```
fig_1 = plt.figure(1)
ax_1  = fig_1.add_subplot()
plt.title(r'$C = {:}$'.format(np.int(NN_Param * 100 + 20)), fontsize=18,
loc='right')
plt.title(r'$t = {:}$'.format(np.int(NN_Time*scale_T)) + ' ' +
r'$s$',fontsize=18, loc='center')
plt.xlabel(r'$x$' + ' ' + r'$(m)$',fontsize=18)
plt.ylabel(r'$u$' + ' ' + r'$(m/s)$',fontsize=18)
ax_1.yaxis.set_major_formatter(yfmt_1)
ax_1.ticklabel_format(axis='y', style='sci', scilimits=(0,0))

fig_2 = plt.figure(2)
ax_2  = fig_2.add_subplot()
plt.title(r'$C = {:}$'.format(np.int(NN_Param * 100 + 20)), fontsize=18,
loc='right')
plt.title(r'$t = {:}$'.format(np.int(NN_Time*scale_T)) + ' ' +
r'$s$',fontsize=18, loc='center')
plt.xlabel(r'$x$' + ' ' + r'$(m)$',fontsize=18)
plt.ylabel(r'$q$' + ' ' + r'$(m^2/s)$',fontsize=18)
ax_2.yaxis.set_major_formatter(yfmt_2)
ax_2.ticklabel_format(axis='y', style='sci', scilimits=(0,0))

X_nn = swashes_x

plt.figure(0)
plt.plot(x_sw, h_sw,  linewidth=lw, color=tratto)

plt.figure(1)
plt.plot(x_sw, u_sw,  linewidth=lw, color=tratto)

plt.figure(2)
plt.plot(x_sw, q_sw,  linewidth=lw, color=tratto)
```

```
nPoints = N2

lw = 3.

X_nn = np.linspace(X_min, X_max, nPoints)
X_nn = np.reshape(X_nn, (len(X_nn), 1))
P_nn = np.ones_like(X_nn) * NN_Param
T_nn = np.ones_like(X_nn) * NN_Time

```

```
X     = np.hstack((X_nn, P_nn, T_nn))

U_nn = model.predict(X)
h_nn = U_nn[:,0].reshape(nPoints,1) * scale_H
u_nn = U_nn[:,1].reshape(nPoints,1) * scale_U
q_nn = h_nn * u_nn

X_nn = X_nn * scale_L


plt.figure(0)
plt.plot(X_nn, h_nn, linewidth  = lw
                   , linestyle  = ''
                   , marker     = 'o'
                   , markersize = 9
                   , mfc = punto
                   , mec = 'k')

plt.figure(1)
plt.plot(X_nn, u_nn, linewidth  = lw
                   , linestyle  = ''
                   , marker     = 'o'
                   , markersize = 9
                   , mfc = punto
                   , mec = 'k')

plt.figure(2)
plt.plot(X_nn, q_nn, linewidth  = lw
                   , linestyle  = ''
                   , marker     = 'o'
                   , markersize = 9
                   , mfc = punto
                   , mec = 'k')
```

```
nPoints = N1

X_nn = swashes_x
X_nn = np.reshape(X_nn, (len(X_nn), 1))
P_nn = np.ones_like(X_nn) * NN_Param
T_nn = np.ones_like(X_nn) * NN_Time
X     = np.hstack((X_nn, P_nn, T_nn))

U_nn = model.predict(X)
```

```python
h_nn = U_nn[:,0].reshape(nPoints,1) * scale_H
u_nn = U_nn[:,1].reshape(nPoints,1) * scale_U
q_nn = h_nn * u_nn

plt.figure(0)
plt.plot(x_sw, h_sw, linewidth=lw, color=tratto, alpha=.6)
plt.legend(['FullSWOF\_1D', 'PINN'], loc='best', fontsize=12)
plt.tight_layout()
plt.grid(True)
plt.savefig('/content/drive/MyDrive/Colab_OUT/05_Dam_2/05_Dam_2_h_' +
str(np.int(NN_Time*scale_T)) + '_' + str(np.int(NN_Param * 100 + 20)) +
'.pdf', bbox_inches='tight')

plt.figure(1)
plt.plot(x_sw, u_sw, linewidth=lw, color=tratto , alpha=.6)
plt.legend(['FullSWOF\_1D', 'PINN'], loc='best', fontsize=12)
plt.tight_layout()
plt.grid(True)
plt.savefig('/content/drive/MyDrive/Colab_OUT/05_Dam_2/05_Dam_2_u_' +
str(np.int(NN_Time*scale_T)) + '_' + str(np.int(NN_Param * 100 + 20)) +
'.pdf', bbox_inches='tight')

plt.figure(2)
plt.plot(x_sw, q_sw, linewidth=lw, color=tratto, alpha=.6)
plt.legend(['FullSWOF\_1D', 'PINN'], loc='best', fontsize=12)
plt.tight_layout()
plt.grid(True)
plt.savefig('/content/drive/MyDrive/Colab_OUT/05_Dam_2/05_Dam_2_q_' +
str(np.int(NN_Time*scale_T)) + '_' + str(np.int(NN_Param * 100 + 20)) +
'.pdf', bbox_inches='tight')
```

List of Figures

List of Tables