TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF PRODUCTION ENGINEERING AND MANAGEMENT

INTERDEPARTMENTAL POSTGRADUATE PROGRAM IN APPLIED MATHEMATICS

MASTER'S THESIS

DEEP LEARNING IN SHALLOW WATERS: SOLUTION OF SHALLOW WATER EQUATIONS USING PHYSICS-INFORMED NEURAL NETWORKS

ELIAS MALAMAS

EXAMINATION BOARD:

Prof. Anargiros Delis (Supervisor)

Prof. Papadopoulou Elena

Prof. Nikolos Ioannis

Chania, August 2024

ABSTRACT

In this thesis we use Physics-informed Neural Networks (PINNs) to solve the Shallow Water Equations (SWE). We provide some insight into the novel idea of PINNs, which constitute a deviation from the rigorous context of the supervised learning paradigm, in the sense that no experimental or simulation data are necessary to train the neural network to solve the SWE, making them the solver of choice in cases where the production of labelled data is costly, time-consuming, or even impossible. We first provide an outline of the system of Partial Differential Equations (PDEs), that describe the SWEs and include some principal properties. We then introduce the idea of using the PINNs as an "unconventional" solver to those PDEs. In order to validate the solver, we engage the PINNs in several benchmark problems of increasing numerical difficulty, in order to prove the adequacy of the PINN idea as a SWE solver. In the sequel, we focus on the effect of the sampling strategy of the training points (domain and boundary) that are used to train the PINN, on the performance of the PINN, in an effort to shed some light on this aspect of the PINN training, when they are used to solve the SWE, applied on a demanding, Riemann problem.

ΠΕΡΙΛΗΨΗ

Σε αυτήν την εργασία χρησιμοποιούμε τα Φυσιογνωστικά Νευρωνικά Δίκτυα (ΦυΓΝεΔ) για την επίλυση των Εξισώσεων Αβαθών Νερών (ΕΑΝ). Παρέχουμε μια επιγραμματική παρουσίαση της καινοτόμου ιδέας των ΦυΓΝεΔ, τα οποία συνιστούν περίπτωση που αποκλίνει από το αυστηρό πλαίσιο του παραδείγματος της επιβλεπόμενης μάθησης, με την έννοια ότι για την εκπαίδευση του νευρωνικού δικτύου για την επίλυση των ΕΑΝ, δεν είναι απαραίτητα δεδομένα από πειράματα ή εξομοιώσεις, ιδιότητα η οποία τα καθιστά τη μόνη επιλογή σε περιπτώσεις όπου η παραγωγή σεσημασμένων δεδομένων είναι δαπανηρή, χρονοβόρα, ή ακόμα και αδύνατη. Σε πρώτη φάση παρέχουμε ένα περίγραμμα του συστήματος Διαφορικών Εξισώσεων Μερικών Παραγώγων (ΔΕΜΠ), οι οποίες περιγράφουν τις ΕΑΝ και δίνουμε και μερικές βασικές ιδιότητές τους. Κατόπιν, εισάγουμε την ιδέα της χρήσης των ΦυΓΝεΔ ως έναν «μη-συμβατικό» επιλύτη αυτών των ΔΕΜΠ. Προκειμένου να επικυρώσουμε τον επιλύτη, χρησιμοποιούμε τα ΦυΓΝεΔ σε ορισμένα προβλήματα αναφοράς, αυξανόμενης αριθμητικής δυσκολίας, προκειμένου να αποδείξουμε την επάρκεια των ΦυΓΝεΔ ως επιλύτη των ΕΑΝ. Στη συνέχεια, εστιάζουμε στην απόδοση των ΦυΓΝεΔ και στην επίδραση της στρατηγικής της δειγματοληψίας των σημείων εκμάθησης (τόσο του πεδίου όσο και των ορίων) που χρησιμοποιούνται στην εκπαίδευση των ΦυΓΝεΔ, σε μια προσπάθεια να φωτίσουμε αυτήν την πλευρά της εκπαίδευσης των ΦυΓΝεΔ, όταν αυτά χρησιμοποιούνται στην επίλυση των ΕΑΝ, σε ένα απαιτητικό πρόβλημα Riemann.

ACKNOWLEDGEMENTS

I would like to grab this opportunity and thank God for giving me the potential and the blessing to attend this Master's Degree on Applied Mathematics, which is completed with this thesis. I had the honor to cooperate with Professor A. Delis to whom I am grateful for encouraging me and guiding me to the novel and challenging idea of Physics Informed Neural Networks and its application on the solution of the Shallow Water Equations. I would also like to thank Professors E. Papadopoulou and I. Nikolos for their participation in the examination board and the support they provided to this Master's thesis. I would also like to thank PhD candidate George Kokkinakis, for sharing his experience in the field of the development tools available for deep learning models, at the initial stages of this work. Throughout the attendance of this degree, I was given the chance to realize that Applied Mathematics is an active Science, that can provide many interesting and useful results to the scientific community, as well as to industry and to Society in general.

I was deeply inspired and motivated by the memory of my brother George, who passed away during the course of this research effort and the preparation of this thesis. Moreover, this thesis and the Master's degree as a whole, could not have been possible to complete (or even start it in the first place) without the patience, understanding and support from my wife Sofia and our children, to which I owe and dedicate this work. Especially for the children, I hope and pray that this effort will be an example to them, to not only love to study Science, but also to learn to focus on their goals and struggle to achieve them and finding the courage to overcome any obstacles that they may encounter, while pursuing these goals.

Στη μνήμη του Αν. Καθ. Εμμανουήλ Μαθιουδάκη

```
!use openacc
      parameter n=1000000000
      implicit real*8 (a-h,o-z)
      real*8 A(n),B(n),C(n)
      do i=1,n
        A(i)=6.5d0
        B(i)=1.5d0
      enddo
      time=omp_get_wtime()
      do i=1,n
       C(i)=2.0d0*A(i)+4.0d0*B(i)
      enddo
      time=omp_get_wtime()-time
      print*,'CPU time = ',time
      print*,C(1),C(n)
      time=omp_get_wtime()
!acc  kernel copyin(A(:),B(:)) copyout(C(:))
      do i=1,n
       C(i)=2.0d0*A(i)+4.0d0*B(i)
      enddo
!acc  end kernel
      time=omp_get_wtime()-time
      print*,'GPU time = ',time
      print*,C(1),C(n)
      stop
      end
```

Ο τελευταίος «παράλληλος» κώδικας που δίδαξε
(ΜΕΘΟΔΟΙ ΑΝΑΠΤΥΞΗΣ ΕΦΑΡΜΟΓΩΝ ΥΨΗΛΩΝ ΕΠΙΔΟΣΕΩΝ, 17/3/2021)

LIST OF FIGURES

LIST OF TABLES

TABLE OF CONTENTS

INTRODUCTION

> **“ἔστω γὰρ αὕτη ἡ ἀπόδειξις βελτίων τῶν ἄλλων τῶν αὐτῶν ὑπαρχόντων, ἡ ἐξ ἐλαττόνων αἰτημάτων ἢ ὑποθέσεων ἢ προτάσεων”**
>
> Αριστοτέλης (384-322πΧ), *Ἀναλυτικὰ ὕστερα*, Βιβλίο Α’, Κεφάλαιο 25.
>
> **[We may assume the superiority, other things being equal, of the demonstration which derives from fewer postulates or hypotheses or propositions.]**

This phrase from Ancient Greek philosopher Aristotle, is considered the origin of the well-known “Occam’s razor” [1], the popular phrase from Scholastic philosopher William of Ockham (1285–1347/49), who stated that “*pluralitas non est ponenda sine necessitate*”, meaning “plurality should not be posited without necessity.” Occam’s razor manifests itself, implicitly or explicitly, as a rule of thumb in many facets of our lives. This rule states that among competing hypotheses that explain known observations equally well, one should choose the “simplest” one. Simplest does not necessarily mean the one that is “easier” to explain. It rather means the one with the fewer variables in the equation or fewer types of abstract ideas or fewer guesses or the one with the fewest assumptions.

This rule of thumb is the underlying foundation of our pursuance for parsimonious models in contemporary Scientific Machine Learning. The task of making an algorithm perform well on new inputs and not only on the training data (a process known as Generalization), is one of the biggest challenges in machine learning and is a field of extensive research. The main tool to diminish the test error without increasing the training error, is Regularization. Ideally, Regularization includes strategies that trade a significant reduction of variance in model error, for a slightly increased bias in model error. There exist multiple approaches to regularize a machine learning model and in particular neural networks. One option is to formulate certain constraints, for example, by directly restricting the parameter values or by adding an extra term to the objective function that constrains the parameters indirectly. Some constraints and restrictions can alter an undetermined problem into a determined one, while others prefer simpler models for better generalization properties.

Overall, these constraints and restrictions are often designed to encode prior knowledge about the problem and, if chosen properly, can help to reduce the generalization error. Fine-tuning deep learning algorithms to build an ideal model capacity that matches the complexity of the underlying problem and avoid underfitting (high bias) and overfitting (high variance), constitutes a clear instantiation of the aforementioned rule, i.e. Occam's razor.

In this thesis we show how to solve a system of Partial Differential Equations that describe incompressible flow, known as Shallow Water Equations, using the underlying Physics Laws that govern this flow, as constraints that encode prior knowledge about the problem. The physics are embedded as a Regularization factor, in the learning process of a special architecture of Neural Networks, known as Physics-Informed Neural Network (PINN). PINN regularization consists of a loss function, which in turn is the sum of weighted loss terms, each one corresponding to underlying physics law, boundary, initial conditions and experimental data (if available) respectively. The necessity of few or no experimental data in the PINN methodology, is the novel characteristic, that brings them in the middle of the Machine Learning world, as shown in Figure I.1.

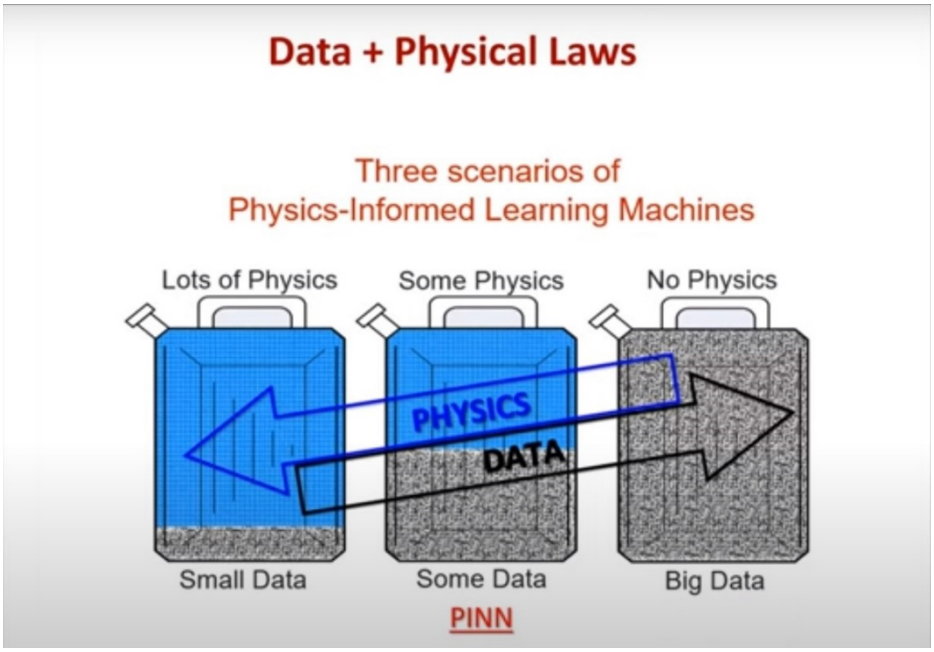


Figure I.1: Location of PINN model in the Machine Learning frame (adopted from [2]).

The structure of this thesis is as follows: In the first Chapter we introduce the SWE and some of their properties, that will be useful in the sequel. In the second Chapter we provide an overview of the PINN concept, as a special paradigm of the Neural Network family of Machine Learning tools. In the first part, we validate the PINN as a SWE solver and subsequently, in the second part, we use the PINN to solve a popular Riemann problem in the SWE context, specifically a Dam break on a wet domain. The study in

the second part of this Chapter aims to investigate the variance (if any) in the performance of the PINN solver, with respect to the selection of the residual data sampling strategy. The ultimate goal is to extract useful conclusions and directions regarding the sampling strategy to follow, when PINNs are used to solve the SWE, directions that previous work in the field has proved that they are problem-specific. The final Chapter provides some conclusions from the findings in the third Chapter, as well as some future directions and suggestions to expand the work conducted in this thesis.

1 SHALLOW WATER EQUATIONS

1.1 Introduction

Shallow Water Equations (SWE) have been proposed by Adhémar Barré de Saint-Venant in 1871 to model flows in a channel [3]. Nowadays, they are widely used to model flows in various contexts, such as: overland flow, rivers, flooding, dam breaks, nearshore, tsunami and meteorology [4]. These equations consist in a nonlinear system of Partial Differential Equations (PDEs), more precisely conservation laws [5], describing the evolution of the height and mean velocity of the fluid. In real situations (realistic geometry, sharp spatial or temporal variations of the parameters in the model, etc.), there is no hope to solve explicitly this system of PDEs, i.e. to produce analytic formulas for the solutions. It is therefore necessary to develop specific numerical methods to compute approximate solutions of such PDEs for each setting [6] [7]. A variety of methods [8], numerical schemes [9] [4], and tools [10] [11], have been released over the years, that tackle the problem of approximating solutions of various SWE problems.

1.2 Mathematical expression

The unknowns of the equations are the water height h(t; x; y) and u(t; x; y), v(t; x; y) which constitute the horizontal components of the vertically averaged velocities, as shown in Figure 1.1.

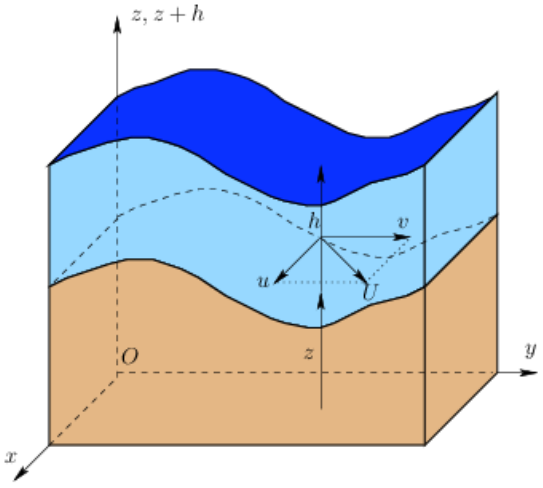


Figure 1.1: Definition of the main variables in the 2-D Shallow Water setting (adopted from [3]).

The derivation of these equations is obtained, 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 and accordingly, the vertical dynamics are negligible;

• the pressure distribution is hydrostatic.

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 (in vector representation):

$$U_t + F(U)_x + G(U)_y = S(U) \tag{1.1}$$

where:

$$U = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, F(U)_x = \begin{bmatrix} hu \\ hu^2 + \frac{g}{2}h^2 \\ huv \end{bmatrix}, G(U)_y = \begin{bmatrix} hu \\ huv \\ hv^2 + \frac{g}{2}h^2 \end{bmatrix} \; and$$

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

The equation that is derived by considering the first row of the above vector equation, expresses the mass balance, while the other two express moment balances. The constant $g = 9.81 m/sec^2$ is the gravitational acceleration. Conventionally it is considered constant for a setting that involves a “small” area, however it actually differs depending on the specific globe location. Function *R* expresses rain intensity, with physical dimensions [L/T] and is a given function of time and space. *I* is the infiltration rate ([L/T]), while S_{0_x} and S_{0_y} are dimensionless quantities and are the opposite of the inclination of the basin in the directions x and y respectively. The topography z may be a variable over the time, in case there is erosion due to flow. The slope terms can be expressed as:

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

S_{f_x} and S_{f_y} are dimensionless quantities and are used to model the friction between fluid and basin ground. The terms μS_{d_x} and μS_{d_y} are the viscous terms, which can be modeled by the Laplace operator, where μ is the kinematic viscosity of the fluid.

For a one-dimensional Shallow Water setting, with a geometrical source term (bottom tomography), equation (1.1) reduces to:

$$U_t + F(U)_x = S(U)$$

where

$$U = \begin{bmatrix} h \\ hu \end{bmatrix}, F(U) = \begin{bmatrix} hu \\ hu^2 + \frac{g}{2}h^2 \end{bmatrix} \text{ and } S(U) = \begin{bmatrix} 0 \\ -ghZ' \end{bmatrix}$$

The above system describes the flow at time $t \geq 0$ and at point $x \in \mathbb{R}$, where $h(x,t) \geq 0$ is the total water height above the bottom, $u(x,t)$ is the average horizontal velocity, $Z(x)$ is the bottom height function and g the gravitational acceleration in the area where the flow occurs.

The left-hand side of this system is the transport operator, corresponding to the flow of an ideal fluid in a flat channel, without friction, rain or infiltration. This is the original model introduced by Saint-Venant and incorporates some important properties of the flow. Specifically, the function *F(U)* expresses the flux of the equation. The transport is more clearly evidenced in the non-conservative form, where A(U) = F'(U) is the matrix of transport coefficients. We rewrite Equation (1.1) as:

$$\partial_x U + A(U)\partial_t U = 0,$$

where

$$A(U) = F'(U) = \begin{bmatrix} 0 & 1 \\ -u^2 + gh & 2u \end{bmatrix}$$

When $h > 0$ the matrix *A(U)* becomes diagonalizable, with eigenvalues:

$$\lambda_1(U) = u - \sqrt{gh} < u + \sqrt{gh} = \lambda_2(U)$$

This important property is called strict hyperbolicity. The eigenvalues express velocities of surface waves on the fluid, which are basic characteristics of the flow. The eigenvalues coincide if *h=0 m*, which holds true for dry zones. In this case, the system is no longer hyperbolic and this induces difficulties at both the theoretical and the numerical level. Numerical schemes developed to solve the SWEs should also consider the importance of ensuring that *h>0* throughout the solution.

From these formulas we derive a useful classification of flows, based on the relative values of the velocities of the fluid, u and of the waves $\sqrt{gh}$. Indeed, if $|u| < \sqrt{gh}$, the characteristic velocities have opposite signs and information propagates both upward and downward the flow, which case is known as subcritical (or fluvial). When $|u| > \sqrt{gh}$ the flow is supercritical (or torrential) and all information goes downward. A transcritical regime holds when some parts of flow are subcritical and others are supercritical.

Since we have two unknowns, namely *h* and *u* (or equivalently, *h* and *q=hu*), a subcritical flow is therefore determined by one upstream and one downstream value, whereas a supercritical flow is determined by the two upstream values. Thus, we have to impose one variable for subcritical inflow/outflow. We impose both variables for supercritical inflow, while for supercritical outflow we consider free boundary conditions.

Furthermore, two quantities are considered useful in the SWE context. The first one is a dimensionless parameter called the Froude number. The flows described by the SWE for a perfect fluid can be classified according to the value of this number, which is defined as:

$$Fr = \frac{|\boldsymbol{u}|}{\sqrt{gh}}$$

If $\mathrm{Fr} < 1$, the flow is characterized as subcritical and information can travel upstream as well as downstream. If $\mathrm{Fr} > 1$, the flow is known as supercritical and information cannot travel upstream. Finally, when $\mathrm{Fr} = 1$, then the flow is named critical and upstream propagation remains stationary. The notion of the Froude number is analogous to that of the Mach number in gas dynamics.
Another important quantity is the *critical height* h_c, defined as:

$$h_c = \left(\frac{q}{\sqrt{g}}\right)^{2/3}$$

for a given $q=hu$. It provides yet another criterion for criticality: if $h > h_c$ ($h < h_c$), the flow is characterized as subcritical (supercritical).

1.3 Scaling of the SWE

It is known that the Neural Networks that are the basis for the solvers that we address in this study, perform better when both the domain of the problem at hand and the solution, belong to the $\mathcal{O}(1)$ scale. Scaling problems to $\mathcal{O}(1)$ in the context of Neural Networks, refers to normalizing the problem's variables and parameters so that they are of the order of magnitude close to one [12]. This practice is crucial for several reasons, primarily related to numerical stability, convergence rates, and the overall efficiency of the neural network training process. Specifically, several works have reported the considerable contribution of scaling to the alleviation of the issue of Gradient Explosions and Vanishing Gradients: When the input variables or parameters span several orders of magnitude, the computed gradients during backpropagation can become extremely large or small, leading to exploding or vanishing gradients. This can severely affect the convergence of the network [13], [14], [15].

Moreover, with scaling precision issues are also avoided. Since computers have finite precision, and operations involving very large or very small numbers can introduce significant rounding errors, scaling variables to $\mathcal{O}(1)$ reduces the risk of such precision issues.

Scaling of variables involved in the SWE problem is done as follows: Suppose that

$$x = L\tilde{x}, y = L\tilde{y}, z = H\tilde{z}, t = T\tilde{t}$$

for the input variables and:

$$h = H\tilde{h}, u = U\tilde{u}, v = U\tilde{v}$$

for the output variables. If we replace the above conventional equations into Equation (1.1), then we get the scaled version, which becomes:

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

Note that for the sake of simplicity, the terms for viscosity and friction have been neglected in the above system of scaled equations. The scaled gravitational constant becomes:

$$\tilde{g} = \frac{H}{U^2}g$$

The contribution of scaling in the case of SWE can be important, especially in cases where the SWE are utilized to model large scale problems, i.e. problems that last for several thousands of seconds, or problems that span a wide range of real estate, or both. Note however that in terms of source code practice, depending on the setting, some of the scaling factors L, H, T, and U can be set to 1, if this leads to a simpler expression for the equations, provided that the problem remains close to the $\mathcal{O}(1)$ scale and the performance of the training process of the neural network solver is not negatively affected. Any variable that has been scaled during the training process of a Neural Network model, should be “de-scaled” back to its original dimensions, in order to make comparisons of model predictions to ground truth data, meaningful and interpretable.

2 DEEP LEARNING AND PHYSICS-INFORMED NEURAL NETWORKS

2.1 Deep Learning and Neural Networks

The use of Neural Networks as universal function approximators, has been introduced in the work by C. Hornik et al since 1989 [16]. The authors proved that a fully connected neural network with one hidden layer of appropriate width (number of neurons), could approximate any function with arbitrary precision.

The above idea was soon extended to the approximation of functions that constitute solutions of Ordinary and Partial Differential Equations. Despite the importance of the aforementioned findings, the solution proposed there was data-driven: provided the availability of enough data, the neural network could provide a solution, after a training phase [17], [18].

In contemporary scientific and industrial research, the availability of enough data that would ensure the applicability of data-driven methods, is not always guaranteed, either due to costly data production processes, or due to time limitations. In these frameworks, data-driven solutions are not fruitful. A different approach was sought, that would exploit the physics laws underlying the studied phenomena. This need signified the introduction of Physics-Informed Neural Networks [19].

NNs are widely used in many visual based application fields such as autonomous driving, extracting information from text files or teaching an industrial robot to monitor a production process [20]. More relevant for our purposes is that they provide a robust approach of approximating nonlinear mappings [15]. The form of the mapping is governed by adjustable parameters that have to be trained. The NN requires a dataset that is divided into a training and test dataset under a certain ratio. The training is performed on the usually large training dataset in order to learn the task. The test dataset is used during training as well, in order to examine if the model is able to produce proper results when using data that is not part of the training data (unseen data). Additionally, it provides an indication if the current network architecture is sufficient to learn the task or has to be adjusted. A part of the test data is used to make predictions afterwards. The goal is to acquire a model that is able to perform well on unseen data [14].

NNs are either used for classification or regression. In a classification task the NN has to decide to which class a certain input feature belongs to. A common example is the classification of handwritten digits

where the well-known MNIST dataset is used [21]. A regression task refers to learning a (nonlinear) function which we address in this thesis.

2.1.1 Network Architecture

The architecture of an artificial neural network is based on the concept of the biological analogue - the brain [22]. An important role is played by the interconnected neurons that are responsible for the exchange of information.

Definition 2.1 ***(Neuron)*** *[21]*

Let $d \in \mathbb{N}$. An artificial neuron $v: \mathbb{R}^d \to \mathbb{R}$ with weight $\boldsymbol{w} \in \mathbb{R}^d$, bias $b \in \mathbb{R}$ and activation function $\sigma: \mathbb{R} \to \mathbb{R}$, is defined as the map $v(\boldsymbol{x}) = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$, for $\boldsymbol{x} \in \mathbb{R}^d$.

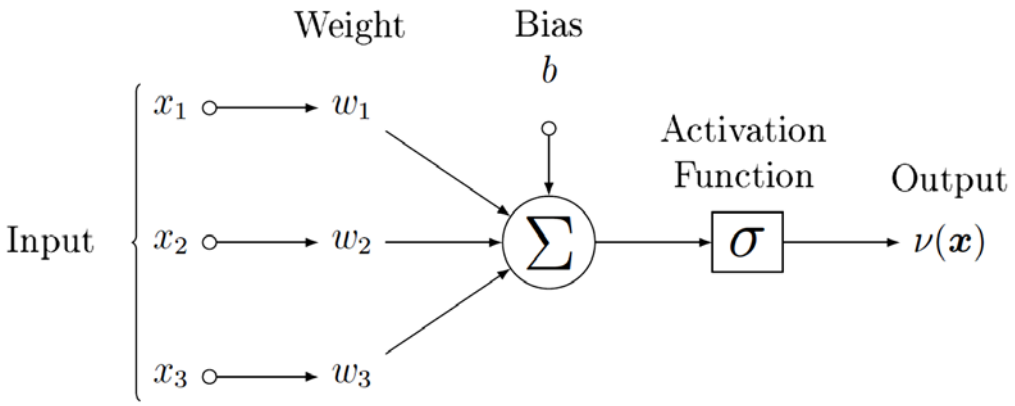


Figure 2.1: Graphical representation of an artificial neuron [21].

The neurons are arranged in layers, so that a NN can then be regarded as a parallel arrangement of concatenated neurons. This is depicted in Figure 2.2, which presents the architecture of the so called Feedforward NN, which is explained in detail in the sequel. The special property of the FNN, compared to other architectures, is that its graph is acyclic and has only edges which follow the direction from input to output.

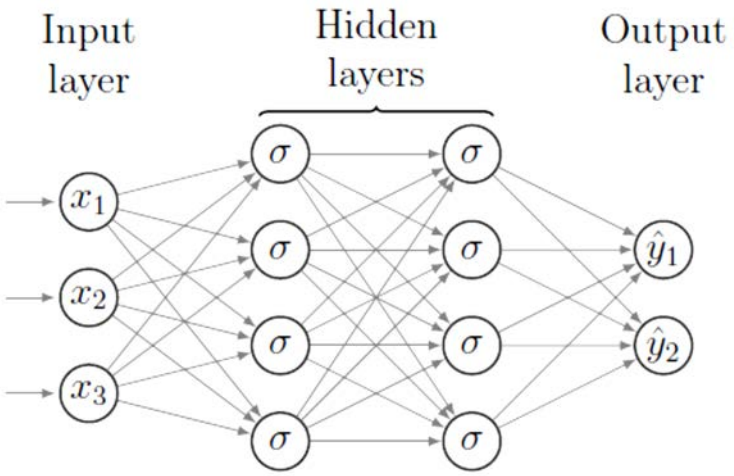


Figure 2.2: The architecture of a typical Feedforward Neural Network, with 2 hidden layers [21].

Though the initial approach of NN function approximators only included a single layer, it was soon realized that more layers would facilitate the training process and thus the overall performance of the NNs. Indeed, the idea of Deep Learning involves the use of more than one hidden layers, probably with less neurons per layer, compared to the width of a NN with a single layer. Depending on the setting, the contribution of each layer in the learning process is that layers that are closer to the input are able to learn simpler structures and patterns in input data and layers closer to the output are able to learn more complex patterns. So depending on the complexity of the setting, the more the layers the better performance is expected from the NN, with a concomitant increase in computational cost. Let us provide a more formal definition of the most popular, deep learning architecture, the Feedforward NN (FNN).

Definition 2.2 ***(FNN)*** *[21]*

Let L, d, $n_1, \dots, n_L \epsilon \mathbb{N}$ and $n_0 := d$. A L-layer Feedforward Neural Network $\hat{u}: \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ with affine linear maps $A_l: \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}$, $\boldsymbol{x} \in \mathbb{R}^d, A_l = \boldsymbol{W}_l \cdot \boldsymbol{x} + \boldsymbol{b}_l$, with $\boldsymbol{W}_l \in \mathbb{R}^{n_l \times n_{l-1}}$, $\boldsymbol{b}_l \in \mathbb{R}^{n_l}$ and activation functions $\sigma_l: \mathbb{R} \to \mathbb{R}$, and $l = 1, \dots, L$, is defined as:

- $\hat{u}^{(0)}(\boldsymbol{x}) = \boldsymbol{x} \in \mathbb{R}^d$ (Input Layer)
- $\hat{u}^{(l)}(\boldsymbol{x}) = \sigma_l\big(\boldsymbol{W_l} \cdot \hat{u}^{(l-1)}(\boldsymbol{x}) + \boldsymbol{b}_l\big) \in \mathbb{R}^{n_l}$ (Hidden Layer)
- $\hat{u}(\boldsymbol{x}) = \boldsymbol{W_L} \cdot \hat{u}^{(L-1)}(\boldsymbol{x}) + \boldsymbol{b}_L \in \mathbb{R}^{n_L}$ (Output Layer)

The activation functions are used component-wise. Here, d is the dimension of the input layer, L denotes the number of layers of $\hat{u}$ (also known as depth) and $n_1, \dots, n_{L-1}$ denote the number of neurons in each layer for each of the L-1 hidden layers, which are also called widths of the layers. Note that L is the number of neurons of the output layer, and this is determined by the number of variables that describe the outputs of the problem to be solved. The matrices $\boldsymbol{W_l}$ contain the weights of the synapses that are attached to each

neuron of layer l. Specifically, the index convention of the elements of this matrix is $[w_l\ ]_{jk}$, where row index j corresponds to neuron j of current layer *l*, while column index k corresponds to the synapse between neuron j of current layer and neuron k of the previous layer *l-1*. Finally, vectors $\boldsymbol{b}_l$ contain the biases of each neuron of layer *l*. Elements of matrices $\boldsymbol{W_l}$ along with those of vectors $\boldsymbol{b}_l$, constitute the so-called learnable parameters of the NN and are usually accommodated into the same, "tall" vector symbolized as $\boldsymbol{\Theta_l}$, so that the complete learnable parameters vector becomes $\boldsymbol{\Theta} = [\boldsymbol{\Theta_1} \dots \boldsymbol{\Theta_L}]^T$. Since the NN prediction function $\hat{u}$ depends on the parameters vector $\boldsymbol{\Theta}$, the NN output is also written as $\hat{u}_{\boldsymbol{\theta}}(\boldsymbol{x})$ to emphasize this dependence.

The activation function "decides whether a neuron is excited or inhibited through other neurons". Moreover, it is responsible for the nonlinear transformation of the input, which prevents the model from being a linear regression model. Commonly used activation functions are pictured in Figure 2.3. Each activation function has its advantages and disadvantages. Notice, that there is no specific rule which activation function to choose for a given task. One of the most popular and widely used activation function is the hyperbolic tangent function:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

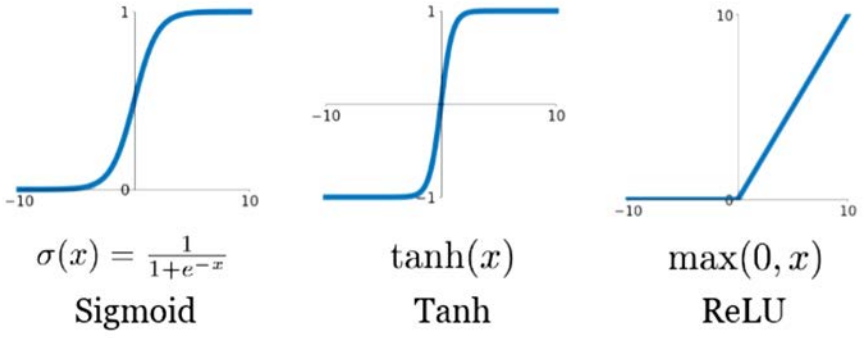


Figure 2.3: Examples of some popular activation functions [21].

Some of the advantages of the hyperbolic tangent function is that it provides a zero-centered output, which can make optimization process faster, since its gradients are more symmetrically distributed around zero. Moreover, it provides steeper gradients compared, e.g. to the sigmoid, which helps mitigate the vanishing gradient problem, to some extent.

2.1.2 Training of NN

The training of the network's parameters reduces to an optimization task, which involves the minimization of a specified loss function [23]. For a given input $\boldsymbol{x}$, the loss function describes a measurement of the error between the network's output $\hat{u}_{\boldsymbol{\theta}}(\boldsymbol{x})$ and the ground truth u($\boldsymbol{x}$), i.e. the desired output. There are several loss functions that are used for machine learning applications, depending on the task they have to solve, among which the most popular one is the Mean Squared Error (MSE).

Definition 2.3 ***(MSE Loss)*** *[21]*

Let $\hat{u}_{\boldsymbol{\theta}}: \mathbb{R}^d \to \mathbb{R}^{n_L}$ with $d, n_L \epsilon \mathbb{N}$ define a FNN. Let $\Omega \subset \mathbb{R}^d$ be a bounded domain and $u: \Omega \to \mathbb{R}^{n_L}$ be the ground truth function that the FNN approximates. Then, for a given set of training points $\mathcal{T} \subset \Omega$, the Mean Squared Error Loss, $\mathcal{L}_{\mathcal{T}}: \mathbb{R}^{\mu} \to \mathbb{R}$ is defined as

$$\mathcal{L}_{\mathcal{T}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{\mu=1}^{N} \|\hat{u}_{\boldsymbol{\theta}}(\boldsymbol{x}) - u(\boldsymbol{x})\|_2^2 \tag{2.1}$$

where $N = |T|$, counts the number of points from the domain of function u that have been selected for training.

As can be readily seen, this is actually the averaged L^2 squared norm. Due to the squaring, larger errors are penalized more heavily than smaller ones. In addition to that, quadratic functions are preferred over other mappings, such as the L^1 norm, since they are differentiable in each point. One can add an additional factor to the loss function containing a certain regularization.

2.1.3 Regularization

The purpose of regularization is to avoid overfitting, which would lead to a model that fits the training data too precisely. The other extreme is called underfitting, in which case the model barely fits the training data and is most likely not able to provide reliable predictions for unseen data. The principles of overfitting and underfitting are visualized in Figure 2.4 [14]. Both cases result in a model that is not able to perform well on unseen data and thus has poor generalization capabilities (high generalization error).

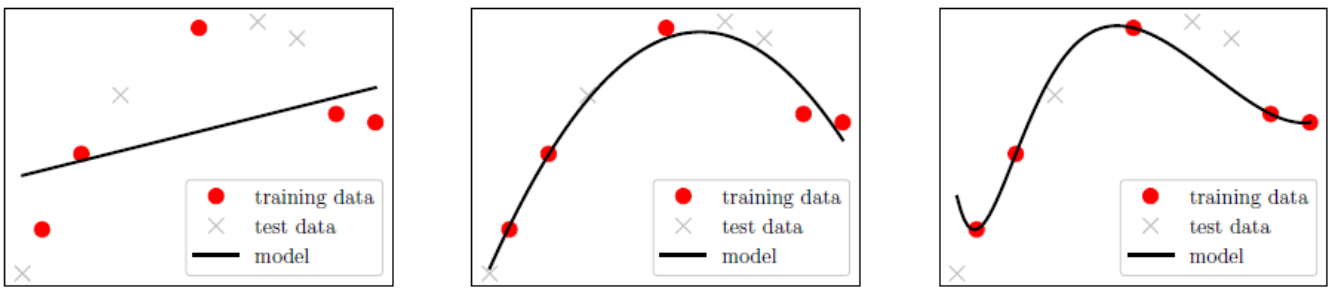


Figure 2.4: Examples of underfitting (left), overfitting (right) and ideal (balanced) fitting (center).

Consider a regression problem, like the one depicted in Figure 2.4, in which case the NN serves as a function approximation. If the model is overfitted, the resulting function will contain a lot of curvature. Suppose this function has the mathematical description of a polynomial. Then, the coefficients of higher degrees would be large. In order to balance the fitting and smoothen the function, these coefficients should become smaller. This is achieved by regularization: by penalizing the weights that are considerably large compared to the others, a balance in fitting can be achieved. There are different types of regularization such as the L_1 (Lasso) regularization or L_2 (Ridge) regularization. In the case e.g. of L_2 regularization, the MSE loss is:

$$\mathcal{L}_{\mathcal{T}}(\boldsymbol{\Theta}) = \frac{1}{N}\sum_{\mu=1}^{N} \|\hat{u}_{\boldsymbol{\Theta}}(\boldsymbol{x}) - u(\boldsymbol{x})\|_2^2 + \lambda \sum_{j \epsilon U \setminus B} \Theta_j^2$$

where U is the set of indices of all trainable parameters of the NN and B is the set indices of biases only, so that the Θ_j's that are included in the regularization, are the weights of the synapses of the NN. The regularization parameter λ, is a tunable parameter, which is tuned by a trial and error process. What regularization does in this case, is that it limits the search space in the space of the trainable parameters Θ_j, by penalizing the flexibility of the model.

The differentiability of the loss function follows from the differentiability of the activation function and the fact that the L^2-norm is differentiable as well. The training process of the network under the MSE loss can then be formulated as the optimization problem:

$$min_{\Theta}\mathcal{L}_{\mathcal{T}}(\boldsymbol{\Theta})$$

This optimization problem is hard non-convex with respect to the parameter space $\boldsymbol{\Theta}$, so that most optimization techniques known from convex optimization, would fail to provide an acceptable solution, most likely would fail to converge at all, due to the presence of many local minima.

The minimization is done using a gradient descent approach. The method of steepest descent (**Algorithm 1**) is the simplest gradient method for optimization. However, for non-convex functions it is possible that the algorithm fails to converge towards the global minimum.

Algorithm 1 *- Method of Steepest Descent*

Require: objective function $\mathcal{L}_{\mathcal{T}}(\boldsymbol{\Theta})$
Require: Initialization: $\boldsymbol{\Theta_0}$
$k \leftarrow 0$ ⇨ Initialize iteration step
$\alpha_k > 0$ ⇨ Compute step size via line search
while $\boldsymbol{\Theta}_{\mathrm{k}}$ not converged **do**
    $k \leftarrow k + 1$
    $g_k \leftarrow \nabla_{\Theta}\mathcal{L}_{\mathcal{T}}(\boldsymbol{\Theta}_{\mathrm{k}-1})$ ⇨ Compute gradient at iteration step k
    $\boldsymbol{\Theta}_{\mathrm{k}} \leftarrow \boldsymbol{\Theta}_{\mathrm{k}-1} - \alpha_{k-1} \cdot g_k$ ⇨ Update parameters
    $\alpha_k \leftarrow \alpha_{k-1}$ ⇨ Compute new step size
end while
return $\boldsymbol{\Theta}_{\mathrm{k}}$ ⇨ Resulting parameters

This is the case for the choice of a small learning rate for instance. The algorithm converges more slowly and possibly towards a local minimum. Otherwise, although a large learning rate could accelerate the process until convergence, there is the risk of not reaching the global optimum at all, as shown in Figure 2.5.

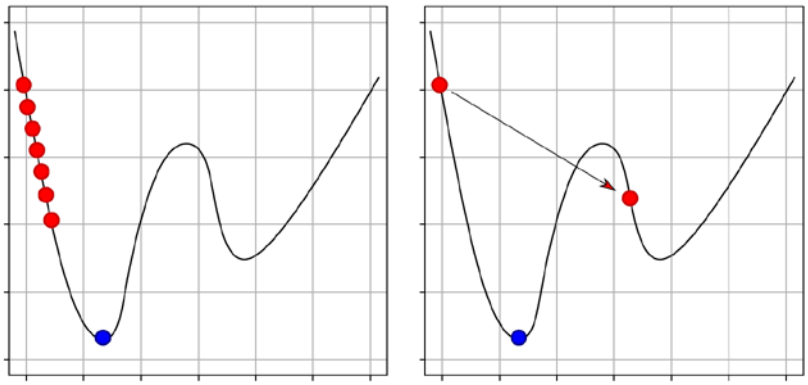


Figure 2.5: Examples of very small learning rate (left) and large learning rate (right). Blue point indicates global minimum, while red dots indicate instantaneous convergence points.

Therefore, this method is rarely used in practice. For our purposes, we consider here a more effective strategy, the Stochastic Gradient Descent (SGD) [24]. Unlike Gradient Descent, the SGD method only takes small subsets of the training dataset, so-called mini-batches, in each optimization step. This yields only approximations of the actual gradients leading to a noisy gradient in total. It is this very noise that makes the method capable of escaping possible local minima. One can draw an analogy to the momentum from physics, where it is described as a mass having a certain velocity while moving in a specific direction.

If the algorithm reaches a local minimum, it is likely to leaving it again due to the momentum and the algorithm continues optimizing. Instead, a gradient descent approach would end optimization when reaching a local minimum due to a zero gradient. Thus, SGD is effective for objective functions with a large amount of curvatures.

The SGD method we are dealing with is called ADAM (ADAptive Moment estimation), first proposed by Kingma and Ba in 2014 [25]. According to the authors, it only requires first-order gradients with little memory requirement. Also, the magnitudes of parameter updates are invariant to rescaling of the gradient. ADAM tackles the aforementioned problems by using estimations of first (mean) and second moments (uncentered variance) of the gradient to adapt the learning rate for the network's parameters in each update. By calculating the exponential moving average, it adds history to the parameter update equation based on the gradient encountered in the previous updates. The outline of the ADAM is shown in the frame that follows.

Algorithm 2 *– ADAM optimizer*

Require: step size α, decay rates β_1, $\beta_2 \epsilon [0, 1)$

objective function $\mathcal{L}_{\mathcal{T}}(\boldsymbol{\Theta})$

Require: Initialization: $\boldsymbol{\Theta_0}$

$k \leftarrow 0$ ◊ Initialize iteration step

$m_0 \leftarrow 0$ ◊ Initialize 1st moment vector

$v_0 \leftarrow 0$ ◊ Initialize 2nd moment vector

while $\boldsymbol{\Theta}_{\mathrm{k}}$ not converged **do**

$k \leftarrow k + 1$

$g_k \leftarrow \nabla_{\Theta} \mathcal{L}_{\mathcal{T}}(\boldsymbol{\Theta}_{\mathrm{k}-1})$ ◊ Compute gradient at iteration step *k*

$m_k \leftarrow \beta_1 \cdot m_{k-1} + (1 - \beta_1) \cdot g_k$ ◊ Update biased 1st moment estimate

$v_k \leftarrow \beta_2 \cdot v_{k-1} + (1 - \beta_2) \cdot {g_k}^2$ ◊ Update biased 2nd raw moment estimate

$\widehat{m}_k \leftarrow \frac{m_k}{1-\beta_1^k}$ ◊ Compute biased corrected 1st moment estimate

$\hat{v}_k \leftarrow \frac{v_k}{1-\beta_2^k}$ ◊ Compute biased corrected 2nd raw moment estimate

$\boldsymbol{\Theta}_{\mathrm{k}} \leftarrow \boldsymbol{\Theta}_{\mathrm{k}-1} - \alpha \cdot \frac{\widehat{m}_k}{\sqrt{\hat{v}_k} + \epsilon}$ ◊ Update parameters

end while

return $\boldsymbol{\Theta}_{\mathrm{k}}$ ◊ Output final parameters

In **Algorithm** 2 the factor ${g_k}^2 = (g_k \odot g_k)$, is the squared element-wise product. The ADAM's authors, suggest that α=0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ as default values are a good choice. In practice,

most libraries that implement ADAM, assume default values for β_1, β_2 and ϵ and the user selects an appropriate learning rate α, which is problem-specific and is determined by a trial and error process.

In deep learning, the gradient $\nabla_{\Theta}\mathcal{L}_{\mathcal{T}}(\boldsymbol{\Theta}_{\mathrm{k}-1})$ is obtained by backpropagation: the gradient of the loss function is expressed as a function of the partial derivatives of the loss function with respect to **every** single weight $[w_l]_{jk}$ and bias $[b_l]_j$ that describes the NN and this is done using the chain rule of differentiating composite functions.

With backpropagation, as the name suggests, the computation of the gradient of the loss function, commences from the differentiation of top level equation (2.1), which involves the output of the NN $\hat{u}_{\boldsymbol{\theta}}(\boldsymbol{x})$. The latter, expressed as a function of weights and biases of the output layer, is differentiated by using the chain rule and thus propagating backwards the calculation of the gradient through the network, back to the input layer. Machine learning frameworks such as TensorFlow (Google Brain Team [26]) and PyTorch (Paszke et al., 2019 [27]) have implemented the method of Automatic Differentiation (AD). Both methods are strongly related to each other, since backpropagation is considered a special case of AD.

Automatic differentiation [28] is an efficient alternative (as opposed to numerical approximation of derivatives), allowing the fast and precise numerical evaluation of higher-order derivatives. Generally, a computer program is nothing else than the sum of its basic arithmetic operations and elementary function evaluations. Automatic differentiation augments the code during execution, by immediately evaluating and storing the derivatives of each basic operation. Then, repeatedly applying the chain rule to the accumulated derivative values, allows to compute the derivative of the whole composition.

AD is one of the two pillars on which the very operation of Physics Informed Neural Networks has been built upon. The other one is the embedding of physics laws in the Loss function.

2.2 Physics-Informed Neural Networks

The NN is a data-driven method mainly used for applications where enough data is available. Especially image data can easily be gathered from open source databases. However, most real-world phenomena are not described by images, but physical laws characterizing conservation laws, diffusion processes, advection-diffusion reaction or other systems. It is possible that only partial data is available when observing such phenomena, which results in a poor predictive performance when using NNs. This motivated the idea to involve these physical laws, often formulated as Partial Differential Equations (PDEs), as additional source of information into practical computations and therefore paved the way for PINNs to emerge [19]. PINNs are NNs which are not only trained with ground truth data, but learn the scientific laws described by the considered PDE. It is actually possible to train the PINN without any

experimental data at all, relying solely on the physics laws as well as on the Boundary and Initial conditions of the problem at hand. The PINN idea has been used extensively since its conception in 2017, to solve a variety of diverse problems, ranging from industrial maintenance [29], scientific computations [30] and non-linear operators [31] and design of industrial materials [32] and biomedical applications [33], [2]. The numerical characteristics of the plain vanilla PINN such as convergence requirements and failure conditions, computational cost and training efficiency, have been investigated extensively in the literature and they remain a field of active research, as can be seen in the references in [34]. Before we explain the concept of a PINN in more detail, we introduce some basic definitions.

Definition 2.4 (Order of PDE) *[21]*

Let $u: \Omega \to \mathbb{R}$, with $\Omega \subset \mathbb{R}^d$ a domain. Let $\boldsymbol{k} = (k_1, \dots, k_d)^T \epsilon \mathbb{N}_0^d$ be a vector of indices of order $|\boldsymbol{k}| := k_1 + \cdots + k_d$, then

$$D^k u := \partial_{x_1}^{k_1} \dots \partial_{x_d}^{k_d} u$$

denotes a Partial Derivative of order $|\boldsymbol{k}|$.

Definition 2.5 – (Well-posed problem) *[21]*

Consider the following PDE:

$$f(\boldsymbol{x}, D^{k_1} u, \dots, D^{k_m} u) = h$$

on $\Omega \times (0, \mathrm{T}]$, with $\Omega \subset \mathbb{R}^d$, Boundary Condition (BC):

$$\mathcal{B}(u) = g, on\ \partial\Omega$$

and Initial Condition (IC):

$$\mathcal{I}(u) = u_0\ in\ \Omega \text{ at } \mathrm{t} = 0$$

Then the problem is characterized as *well-posed* if:

- There exists at least on solution for every data
- For every *h*, *g* and u_0, there exists at most one solution
- The solution $u = u(h, g, u_0)$ depends continuously on the data *h*, *g* and u_0, i.e. small changes in the data cause small changes in the solution.

A PDE defined with a set of BC and IC constitutes an initial-boundary-value problem (IBVP). In order to deal with well-posed problems, we need to prescribe proper BCs that solution *u* assumes along the boundary $\partial\Omega$, which may be one of the following types:

- *Dirichlet*: $u = u_D$
- *Neumann*: $\boldsymbol{n} \cdot \nabla u = u_N$
- *Robin*: $\alpha \cdot \boldsymbol{n} \cdot \nabla u + \beta \cdot u = u_R$

- *Periodic*: $u_P(\boldsymbol{a}) = u_P(\boldsymbol{b}), u_P{}'(\boldsymbol{a}) = u_P{}'(\boldsymbol{b})$, where $\boldsymbol{a}, \boldsymbol{b} \in \partial\Omega$

for some given functions (data) u_D, u_N, u_R, or u_P: $\partial\Omega \to \mathbb{R}$ and constants $\alpha, \beta \in \mathbb{R}$. Here, vector $\boldsymbol{n}$ denotes the normal vector and $\nabla = \left(\frac{\partial}{\partial x_1}, \dots, \frac{\partial}{\partial x_d}\right)^T$ is the Gradient operator.

2.2.1 Architecture of a PINN

Let $u(\boldsymbol{x})$ be an unknown function of $\boldsymbol{x}$, defined in a spatio-temporal domain $\hat{\Omega} = \Omega \times [0, \mathrm{T}]$, such that $\Omega \subset \mathbb{R}^d$. In this case, we adopt a generalized definition for vector $\boldsymbol{x} := (x_1, \dots, x_d, t)^{\boldsymbol{T}} \in \hat{\Omega}$. We consider a PDE whose solution is the function $u: \hat{\Omega} \to \mathbb{R}$, defined as:

$$f(\boldsymbol{x}, D^{k_1}u, \dots, D^{k_m}u, \boldsymbol{\lambda}) = 0$$

where $\boldsymbol{\lambda}$ are the data on which the PDE depends:

$$\mathcal{B}(u, \boldsymbol{x}) = 0, \qquad \boldsymbol{x} \in \partial\hat{\Omega}$$

where the boundary $\partial\hat{\Omega} = \partial\Omega \times (0, T) \cup \Omega \times \{0\}$, is meant to for both spatial boundary points ($\partial\Omega \times (0, T)$) and for temporal boundary points, which are the Initial points ($\Omega \times \{0\}$).

In order to solve the PDE with a PINN, the following procedure should be followed:

1. Construct a NN $\hat{u}_{\boldsymbol{\theta}}$ with learnable parameters $\boldsymbol{\Theta}$.
2. Specify two sets of scattered training points, namely $\mathcal{T}_f \subset \hat{\Omega}$ and $\mathcal{T}_b \subset \partial\hat{\Omega}$, for the PDE and the BC/IC respectively.
3. Specify the loss function $\mathcal{L}_{\mathcal{T}}(\boldsymbol{\theta})$ as the weighted sum of MSE losses for both the PDE and BC/IC residuals.
4. Train the PINN by minimizing the loss function $\mathcal{L}_{\mathcal{T}}(\boldsymbol{\theta})$, until a desired number of iterations is reached, or a desired minimum of residual is reached.

A visualization of this procedure for the solution of the 1-D heat equation $u_t = \lambda u_{xx}$, with a mixture of Dirichlet and Neumann BCs, is depicted in Figure 2.6.

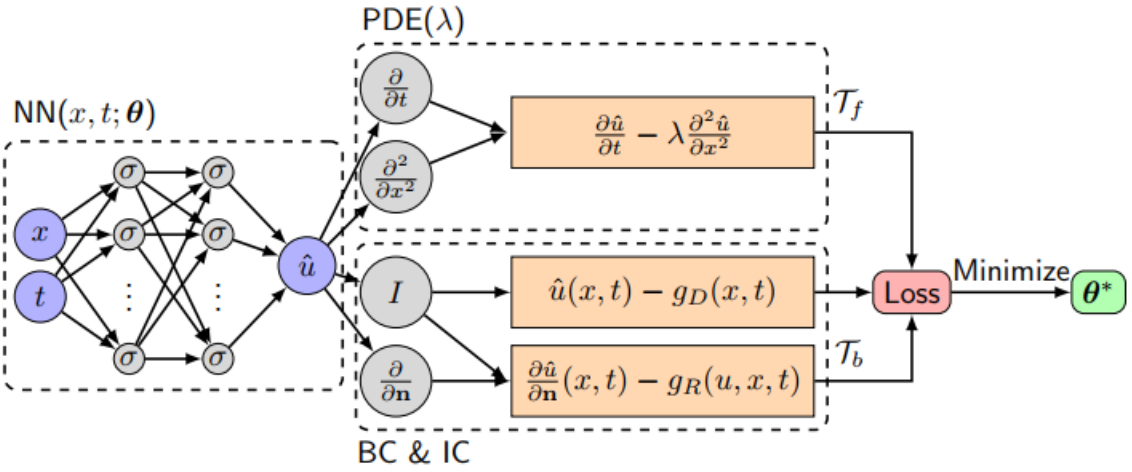


Figure 2.6: Schematic of the PINN solver for the 1-D heat equation with mixed BCs [35].

The Dirichlet BC is $u(x,t) = g_D(x,t)$, on $\Gamma_D \subset \partial\hat{\Omega}$, while the Neumann BC is $\frac{\partial u}{\partial \boldsymbol{n}}(x,t) = g_R(u,x,t)$ on $\Gamma_R \subset \partial\hat{\Omega}$. The IC in this case is treated as a special type of Dirichlet BC. Note that $\mathcal{T}_f$ and $\mathcal{T}_b$ denote the sets for training the PDE residual and the BC/IC residuals respectively.

The left side of the schematic shown in Figure 2.6 forms the NN $\hat{u}_{\boldsymbol{\theta}}$ with network trainable parameters $\boldsymbol{\Theta}$. The function $\hat{u}_{\boldsymbol{\theta}}$ constitutes a surrogate model of the true solution u. In order to solve the PDE, the computation of (partial) derivatives is essential. By the usage of AD we are able to compute the derivatives (of any order) of our network $\hat{u}_{\boldsymbol{\theta}}$ with respect to all relevant input variables, independent from the structure of the underlying programming code. Thus, we can include the PDE residual into our computations with no need for a mesh generation, as it is used e.g. in the case of the finite element methods. This inclusion is done by considering two subsets of the training data $\mathcal{T} \subset \hat{\Omega}$. As explained above, the set $\mathcal{T}_f \in \hat{\Omega}$ contains points in the domain (so called collocation points) and the set $\mathcal{T}_b \in \partial\hat{\Omega}$ contains points on the boundary and initial data. The distribution of the training points may follow any model desired, depending on the problem at hand, e.g. randomly or uniformly distributed [36]. As indicated in Figure 2.6, points in $\mathcal{T}_b$ are used as ground truth data so that the BC/ICs are satisfied. Points in $\mathcal{T}_f$ are used to calculate the given PDE residual towards its minimization. As such, the network on the right side of Figure 2.6 does not contain any trainable parameters, instead it utilizes output from the NN to calculate the residuals. This is the key difference from a standard NN, which uses solely ground truth data for all considered training data. In doing this, the model $\hat{u}_{\boldsymbol{\theta}}$ learns the physical law imposed by the PDE.

2.2.2 Training

Like NNs, PINNs aim to minimize a certain loss function. But different from standard NNs, the loss function of a PINN is a linear combination of two loss functions evaluated at points in $\mathcal{T}_f$ and $\mathcal{T}_b$, separately. It is convenient to choose the MSE as loss function for each training set. A more formal definition of the PINN loss function follows [21].

Definition 2.6 (PINN Loss)

Let $\hat{u}_{\boldsymbol{\theta}}: \mathbb{R}^d \to \mathbb{R}$ with $d \epsilon \mathbb{N}$ be a FNN. Let also $\widehat{\Omega} \subset \mathbb{R}^d$, be a bounded domain and $u: \widehat{\Omega} \to \mathbb{R}$ be the solution of the PDE $f(\boldsymbol{x}, D^{k_1}u, \dots, D^{k_m}u, \boldsymbol{\lambda}) = 0$, with boundary and initial data $\mathcal{B}(u, \boldsymbol{x}) = 0,\ \boldsymbol{x} \in \partial\widehat{\Omega}$. Then, for training points sets $\mathcal{T}_f \subset \hat{\Omega}$ and $\mathcal{T}_b \subset \partial\hat{\Omega}$, the PINN loss $\mathcal{L}_{\mathcal{T}}$ is given by the following weighted sum:

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

where

$$\mathcal{L}_{\mathcal{T}_f}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{T}_f|} \sum_{x \in \mathcal{T}_f} \| f(\boldsymbol{x}, D^{k_1}\hat{u}_{\boldsymbol{\theta}}, \dots, D^{k_m}\hat{u}_{\boldsymbol{\theta}}) \|_2^2$$

and

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

with tunable weights w_f and w_b.

The PINN loss is leveraging Regularization against overfitting, since the term $w_f \cdot \mathcal{L}_{\mathcal{T}_f}(\boldsymbol{\theta})$ acts as regularization, with regularization parameter w_f, in order to penalize those parameters $\boldsymbol{\Theta}$ which would lead to solutions that would not satisfy the PDE. Therefore, a key property of the PINNs, is that they can be effectively trained using small datasets. Also, it has been empirically reported that a properly chosen boundary weight w_b, could accelerate the overall training and result in a better performance. It is widely recommended in literature to choose w_b large and positive, in order to improve the accuracy of the model [14] [24]. Like NNs, the training corresponds to the minimization problem $min_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}}(\boldsymbol{\theta})$. Since the PINN loss is a linear combination of two MSE losses, it is differentiable as is for NNs. The training of the network's parameters $\boldsymbol{\Theta}$, is proceeded using a gradient descent approach like ADAM or L-BFGS, the latter being a popular quasi-Newton method that is based on low-rank approximations of the inverse Hessian of the loss function [14].

The required number of iterations highly depends on the problem (e.g. the smoothness of the solution). In each step of the training process, partial derivatives have to be calculated. Thus, if the number of collocation points is very large, it is computationally expensive to calculate the PINN loss in every

iteration. Thus, a number of alternative data sampling approaches have been proposed that accelerate the computation, without sacrificing accuracy. Since we are solving a non-convex optimization problem, there is no theoretical guarantee that this process converges to the global minimum. However, it has been reported that if the given PDE is well-posed, the PINN method is capable of achieving good prediction accuracy, given a sufficient number of collocation points. In fact, as Shin, Darbon, and Karniadakis presented [37], for linear second-order elliptic and parabolic type PDEs, the sequence of minimizers strongly converges to the PDE solution in C_0, where C_0 denotes the set of continuous functions. It has also been stated that PINNs are able to solve certain ill-posed problems as well, given only partial measurement data on the boundary.

The solution of a PDE problem using PINNs involves the determination of several parameters, tunable by the analyst, which are called *hyperparameters* [19]. It turns out that the number of these parameters is sometimes large, each hyperparameter affecting a different aspect of the performance of the PINN. Tuning these parameters is usually the result of a trial-and-error process, since there are no general rules that apply on the fixing of these parameters. In Table 2.1 we have gathered some of the most important of these parameters, however the plethora of them makes the design of PINNs an art, rather than a science. Experience in training PINNs to solve problems of "similar" complexity, may provide useful intuition to the tuning of the hyperparameters. However, it should be emphasized that due to the strong dependence of the solution of a PDE on its Initial data, it may happen that a PDE with different Initial data, would require PINN solvers with significant differences in hyperparameter selection, e.g. size of the PINN, learning algorithm, etc.

Table 2.1: Tunable hyperparameters of PINNs

Hyperparameter name	**Symbol/Note**
Depth of PINN	L
Width of each layer	n_l
Optimization algorithm	Adam, L-BFGS
Learning rate of Optimization algorithm	α
Neuron connectivity	FNN, Fully connected NN
Number of iterations (epochs)	-
Weights of components of loss function	w_f, w_b
Norm of Loss function	L^2, L^1
Number of collocation points (training points from the domain)	N_f
Number of boundary points (spatial and temporal boundaries)	N_b
Data Sampling strategy that generates the training points	Sobol, uniform, random
Method of weights and biases Initialization	Glorot
Activation function	tanh, sigmoid
mini-batch size	-

It has to be noted here that due to the limitations of the plain vanilla PINN to solve certain types of complicated problems, a number of alternative training methodologies have been investigated, such as transfer learning [38] [39] [40], variable boundary and initial conditions [41] [42], as well as curriculum learning [43], where the PINN is gradually exposed to the difficult learning aspects of the problem at hand. Also, a number of tools have been used to build PINN solvers for numerical problems, that include Matlab [44], TensorFlow [26] and PyTorch [27] and other linear algebra libraries focused on machine learning. The DeepXDE library [35] that we use in this work is a high level library that builds on top of other linear algebra libraries such as TensorFlow and PyTorch.

2.2.3 Error considerations

Though a rigorous analysis of the errors related to the use of a PINN to approximate a function (which may also be the solution to a Partial Differential Equation) is out of the scope of this thesis, it is considered beneficial to provide an overview of the errors involved in this context. 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, 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, are due to the fact that [35]:

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

2. The NNs 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: $u_{\mathcal{T}} = argmin_{f \epsilon \mathcal{T}} \mathcal{L}(f, \mathcal{T})$. Then, the **generalization error**, determined both by the density of the training points and by the NN's expressiveness, is defined as: $\boldsymbol{\mathcal{E}_{gen}} := \|u_{\mathcal{T}} - u_{\mathcal{F}}\|$.

3. Minimizing the non-convex, highly nonlinear, loss function can be computationally unmanageable, 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** is defined as: $\boldsymbol{\mathcal{E}_{opt}} := \|\tilde{u}_{\mathcal{T}} - u_{\mathcal{T}}\|$.

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

$$\mathcal{E} \overset{\text{def}}{=} \|\tilde{u}_{\mathcal{T}} - u\| \leq \|u_{\mathcal{F}} - u\| + \|u_{\mathcal{T}} - u_{\mathcal{F}}\| + \|\tilde{u}_{\mathcal{T}} - u_{\mathcal{T}}\| = \boldsymbol{\mathcal{E}_{app} + \mathcal{E}_{gen} + \mathcal{E}_{opt}}$$

As previously stated, the estimation of this error is currently still an open research problem. The graphical illustration of this error decomposition is shown in Figure 2.7.

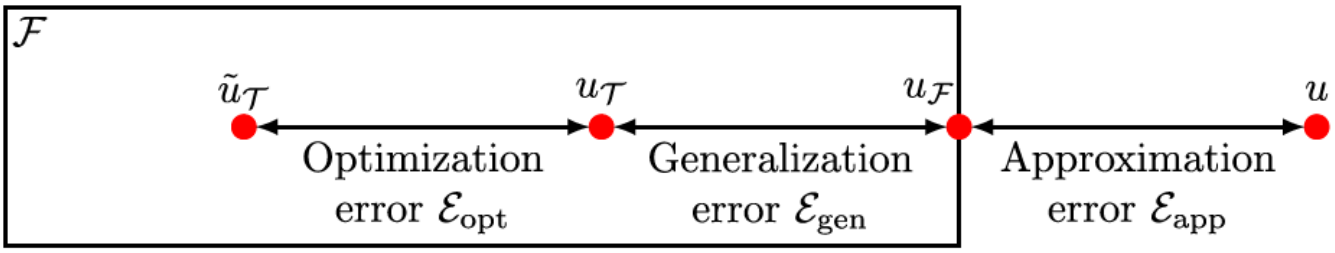


Figure 2.7: Breakdown of the error in its components [35].

The capacity of a model, more precisely, its ability to fit a wide variety of functions (features), plays an important role in terms of performance. Choosing the capacity in a way that suits the complexity of the problem or, in other words, finding the balance between underfitting and overfitting, is a central challenge in machine learning. The relationship between the training error and the generalization error with respect to the capacity is depicted in Figure 2.8. As indicated, the optimal capacity is reached when the generalization error is as low as possible.

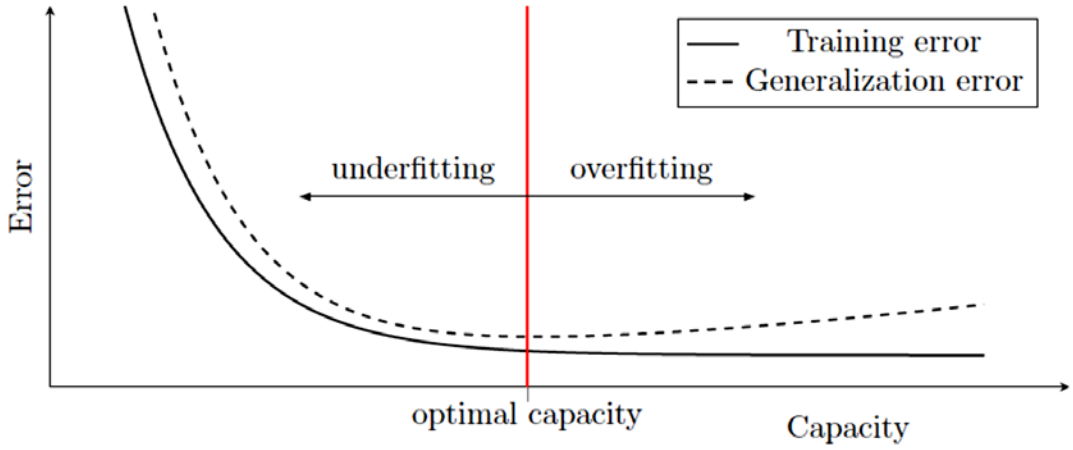


Figure 2.8: Relation between generalization error and model capacity [21].

Even in the case where the suitable number of features have been selected for the model, its generalization error is also affected by another factor, which is the number of samples that have been used to train the model parameters. As the amount of training data increases, the generalization error and the training error tend to coincide to an error value, that represents the best the model could do if it had an infinite number of training data. This error value constitutes a bias in the model and is a property of the kind of functions that are being used to approximate the true solution function. This behavior is depicted in Figure 2.9.

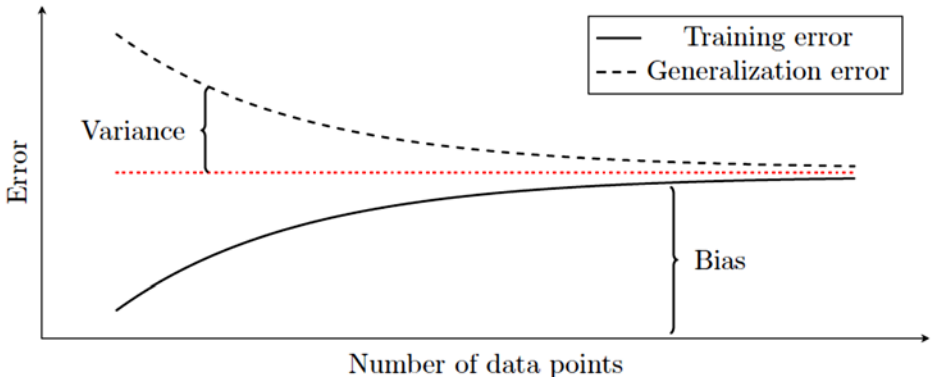


Figure 2.9: Relation between the generalization error and the number of training data points [21].

As can be seen in Figure 2.9, the generalization error can be decomposed into a bias, which measures how well we can hypothetically do in the infinite data limit, and a variance, which measures the typical errors introduced in training our model due to sampling noise from having a finite training set.

3 SIMULATIONS

3.1 Overview

The problem of solving the SWE system using PINN solvers is an open research topic. The SWE research community has been investigating the last few years, the applicability of the PINN on the solution of SWE problems, both academic ones and real life applications. A number of references in the literature exist that provide PINN-based solutions of the SWE for meteorology [45], river flow [46], free-surface flow [47], [48] and on grids of arbitrary geometry [49]. The results are promising since PINN-based solutions of SWE problems, provide solutions, which are comparable in terms of accuracy, to solutions from mature numerical schemes.

We begin by providing an overview of the methodology that we will follow in this Chapter in order to conduct the intended simulations. We propose an ML-based solution to the SWE problem, whereby the input layer $\mathbf{x}$ represents the independent variables and parameters of the problem, $\mathbf{x} = (x, y, t)$, and the trained model $\aleph$ is expected to provide an approximate solution for $h(\mathbf{x})$, $hu(\mathbf{x})$ and $hv(\mathbf{x})$ in the corresponding domain; in other words:$\boldsymbol{U}(\boldsymbol{x}) \cong \widetilde{\boldsymbol{U}}(\boldsymbol{x}) = \aleph(\boldsymbol{x}; \boldsymbol{\Theta})$, where $\widetilde{\boldsymbol{U}}(\boldsymbol{x})$ denotes the output from the PINN, which is in turn defined by the group of trainable parameters $\boldsymbol{\Theta}$ (i.e. weights and biases). The PINN models proposed are trained by minimizing the composite loss function, defined as:

$$\mathcal{L} = \lambda_1 \cdot \mathcal{L}_f + \lambda_2 \cdot \mathcal{L}_b + \lambda_3 \cdot \mathcal{L}_0$$

where $\mathcal{L}$ (a scalar) is the loss function to be minimized, λ_{1-3} are the penalty coefficients for every specific loss term; namely, $\mathcal{L}_P$ penalizes the residuals of the SWEs, $\mathcal{L}_b$ and $\mathcal{L}_0$ penalize the BCs (subscript b) and ICs (subscript 0) residuals, respectively. Based on the vector representation of the SWE, as given in Equation (1.1), these loss terms are in turn given by:

$$\left.\begin{aligned} \mathcal{L}_f &= \frac{1}{N}\sum_{i=1}^{N}\left|\partial_t \widetilde{\boldsymbol{U}}_i + \partial_x \boldsymbol{F}\left(\widetilde{\boldsymbol{U}}_i\right) + \partial_y \boldsymbol{G}\left(\widetilde{\boldsymbol{U}}_i\right) - \boldsymbol{S}\left(\widetilde{\boldsymbol{U}}_i\right)\right| \\ \mathcal{L}_b &= \frac{1}{N_b}\sum_{i=1}^{N_b}\left|\widetilde{\boldsymbol{U}}_{b,i} - \boldsymbol{U}_{b,i}\right| \\ \mathcal{L}_0 &= \frac{1}{N_0}\sum_{i=1}^{N_0}\left|\widetilde{\boldsymbol{U}}_{0,i} - \boldsymbol{U}_{0,i}\right| \end{aligned}\right\} \quad (3.1)$$

The tilde symbol (~) denotes neuronal network output and the subscript $i \in [1, N]$ refers to the ith collocation point. N represents the number of collocation points, which is defined from a discretized domain of the independent variables such that $N = n_x \times n_y \times n_t$, where n_x, n_y and n_t are the number of points used to discretize the domain along the x, y and t coordinates, respectively. The boundaries of the spatio-temporal domain are represented by a subset of N; in particular, the model will employ $N_b < N$ and $N_0 < N$ points to define the BCs and ICs, respectively. Therefore, the boundary and initial conditions of the problem are set via selected collocation points, with the loss function at these points evaluated using the above equations (3.1). For each approximate solution produced by the PINN, the partial derivatives in (3.1) are computed through the method of automatic differentiation (autodiff) (as has been explained in the previous Chapter), which back-propagates derivatives from the outputs to the targeted inputs, through the chain rule to compute the desired derivatives. Thus, the partial derivatives of the approximate solution with respect to the independent variables, can be computed without the errors common to numerical differentiation techniques. The loss function is minimized using the gradient descent method, with gradients of the loss function with respect to trainable parameters, computed by back-propagation. These parameters can be updated either using all, or a subset (batch) of the collocation points. The procedure followed throughout the subsequent simulations, is shown in Algorithm 3.

In this thesis, the 1-D variant of the SWE model will be used in the simulations. The derivation of the equations involved in the 1-D case is straightforward from the aforementioned equations (1.1). The PINN was developed using the DeepXDE library [35] and will be validated first for its numerical stability and accuracy, using some benchmark problems, of increasing numerical complexity. The development of the code was done on the Google Colab platform [50], based on the Tensorflow v1 and v2 libraries [26]. The runtime environment used, offered a variety of computational resources, depending on the level of user registration, that included CPUs, A100, T4 and L4 GPUs, as well as v2 TPUs [50]. The availability of those computational resources was varying, since Google Colab offers them on a sharing, best-effort, basis. The details of the PINN model utilized for the simulations are shown in Table 3.1.

Table 3.1: Hyperparameters of PINN model

Hyperparameter	**Value**
NN Depth (hidden layers)	4
Layer width	60
Optimizer	Adam
Learning rate	10^{-4}
Iterations	200,000
Collocation points (spatio-temporal domain)	4000
Boundary points	2000
Initial conditions points	1000
Weights initialization	Glorot
Activation function	tanh (hyperbolic tangent)
Sampling method	Random

Ground truth data that will be used in this thesis as reference for error metric calculations, were acquired from the library SWASHES (Shallow Water Analytic Solutions for Hydraulic and Environmental Studies). This library has been developed by the University of Orleans, Dept. of Mathematics, France [10]. It incorporates analytical solutions for a plethora of benchmark problems involving the SWE and has been developed for this very purpose of validating research code. Note that an extensive phase of hyperparameter optimization preceded the implementation phase, where suggested practices in the literature for the class and size of the problem at hand, were investigated, including the activation function to use [35], the NN architecture [51], the optimization algorithm [34] as well as the sampling strategies [36].

3.2 Code validation

3.2.1 Lake at rest with an immersed bump

The first case that we examine is the case of an immersed bump. This case is utilized as a benchmark to test the ability of the model under validation, to preserve steady states. A typical setting of this case is depicted in Figure 3.1. The topography of the bottom is completely immersed, flat at the boundaries, while a bell-like bump exists, centered at $x = 10m$ of the spatial domain. In this case, we examine the solution provided by the PINN at the steady state, so as to evaluate whether the solver, under conditions of null input, generates numerical noise on its own, noise that could be augmented as simulation time progresses.

Algorithm 3: *Proposed PINN-based algorithm:*

Input: 1. N_0 data for initial condition $\{0, x_0^i, u_0^i\}_{i=1}^{N_0}$

2. N_b data for boundary condition $\{t_b, x_b^i, u_b^i\}_{i=1}^{N_b}$

3. N_f collocation points $\{t_f, x_f^i\}_{i=1}^{N_f}$ internal to the domain Ω

4. Neural network architecture (input neurons, output neurons, hidden layers, hidden neurons per layer, activation function)

5. Balancing factors $\lambda_3, \lambda_2, \lambda_1$ for terms $MSE_0, MSE_b \; and \; MSE_f$ (empirically determined)

Output: 1. Neural Network parameters $\boldsymbol{\Theta}$ $(weights \{\boldsymbol{W}^l\}_{l=1}^L \; and \; biases \{\boldsymbol{b}^l\}_{l=1}^L \; for \; all \; layers \; L)$

2. Convergence performance plots

Step 1: Initialize network parameters vector $\boldsymbol{\Theta}$

- Set hyper-parameters for optimizer (e.g. for ADAM, define number of epochs, learning rate α)
- Generate training data set and test data set from acquired data for initial condition, boundary condition and collocation points.
- Define batch size

Step 2: **for** all epochs **do**

for all batches **do**

1. compute $\{u_{NN}(0, x_0^i, \boldsymbol{\Theta})\}_{i=1}^{N_{0-batch}}$
2. compute $\left\{\frac{\partial}{\partial x} u_{NN}(t_b^i, x_b^i, \boldsymbol{\Theta})\right\}_{i=1}^{N_{b-batch}}$
3. compute $\{f_{NN}(t_f^i, x_f^i, \boldsymbol{\Theta})\}_{i=1}^{N_{f-batch}}$
4. compute MSE_0, MSE_b, MSE_f
5. evaluate cost function $\mathcal{L} = \lambda_3 * MSE_0 + \lambda_2 * MSE_b + \lambda_1 * MSE_f$
6. update network parameters vector $\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \alpha \frac{\partial C}{\partial \boldsymbol{\Theta}}$ (according to optimizer of choice)

end for

end for

Step 3:

- **Output** $\boldsymbol{\Theta} = [\{\boldsymbol{W}^l\}_{l=1}^L \quad \{\boldsymbol{b}^l\}_{l=1}^L]^T$
- Plot cost function $\mathcal{L}$ vs epoch number for training and validation data

Following are the mathematical expressions that describe the case:

Bottom topography: $z(x) = \begin{cases} 0.2 - 0.05(x-10)^2 & if\ x \in (8,12) \\ 0 & otherwise \end{cases}$

Initial conditions: $\begin{cases} h = 0.5 - z\ m \\ q = 0\ \frac{m^2}{s} \end{cases}$

PDEs: $\begin{cases} h_t + (hu)_x = 0 \\ (hu)_t + (hu^2 + 0.5gh^2) = ghS_{0x} \\ q - hu = 0 \end{cases}$

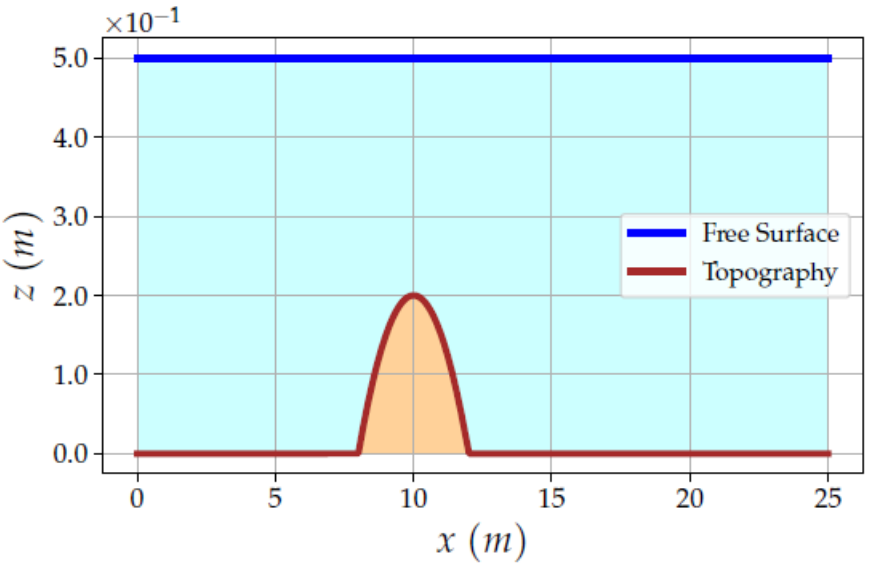


Figure 3.1: Snapshot of steady state for an Immersed bump.

The solutions derived from the simulations are depicted in Figure 3.2. This grid-like figure gathers the results from time instances T=0s and T=100s, for the output variables of interest, the height h, the horizontal velocity u, and the flow q. The ground truth solutions for a total of 100 points derived from SWASHES, have been superimposed for each case. It is reminded here that the SWASHES library provides the exact solutions for the problems implemented in its repertoire. The error metrics that assess the performance of the PINN model have been gathered in Table 3.2.

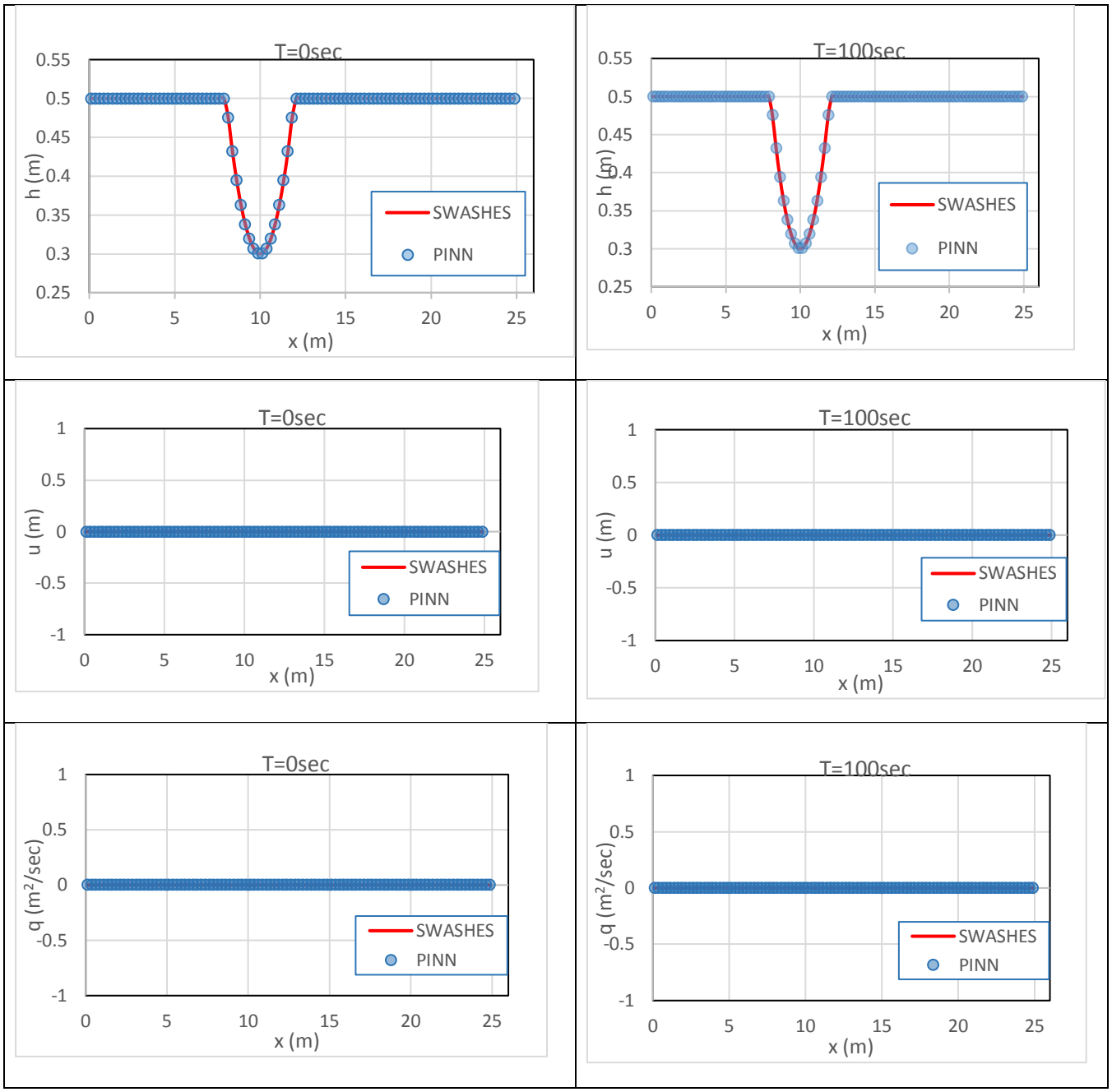


Figure 3.2: Simulation results for immersed bump. Left pane: h, u, q plots for time T=0s. Right pane: h, u, q plots for T=100s.

Table 3.2: L^2 errors for immersed bump simulations

Output variable	Metric	T=0sec	T=100sec
h	$\|h_{SW} - h_{PINN}\|_2 / \|h_{SW}\|_2$	0	3.6×10^{-12}
u	$\|u_{SW} - u_{PINN}\|_2$	0	1.7×10^{-8}
q	$\|q_{SW} - q_{PINN}\|_2$	0	5.2×10^{-9}

Note that the error metrics shown in *Table 3.2* are not the same for all output variables. The reason for this is because the ground truth for variables u and q in this particular setting, are both 0, so a relative error that requires a norm in the denominator cannot be defined. On the contrary, the metric for height variable h is the L^2 relative norm, which is well defined for this variable and this setting. Note that the initial conditions for h and q (and consequently for u) are part of the definition of the problem, thus they are incorporated into the learning process of the PINN as loss terms. By using a suitable number of iterations or a suitable weight for these loss terms, the PINN is forced to learn the initial conditions perfectly. It is also noteworthy to mention that the errors shown in *Table 3.2* for time instance T=100sec, could be diminished even further, if the computational resources available allowed more iterations (see concluding remarks).

3.2.2 Lake at rest with an emerged bump

In this setting the same bottom topography as with the previous case is retained, however, some parts of the bottom are above water height. This case tests the ability of the PINN to preserve steady state, while the treatment of wet/dry transition is also analyzed. The following conditions apply in this case:

Boundary conditions: $\begin{cases} h = 0.1\ m \\ q = 0 \frac{m^2}{s} \end{cases}$

Initial conditions: $\begin{cases} h = \max(0.1, z) - z\ m \\ q = 0 \frac{m^2}{s} \end{cases}$

The steady state is graphically represented in Figure 3.3: Emerged bump, topography and free surface at steady state.. The PINN in this case is tested in terms of its ability to retain this response for sufficient time (SWASHES max time is 100sec, as in the previous case).

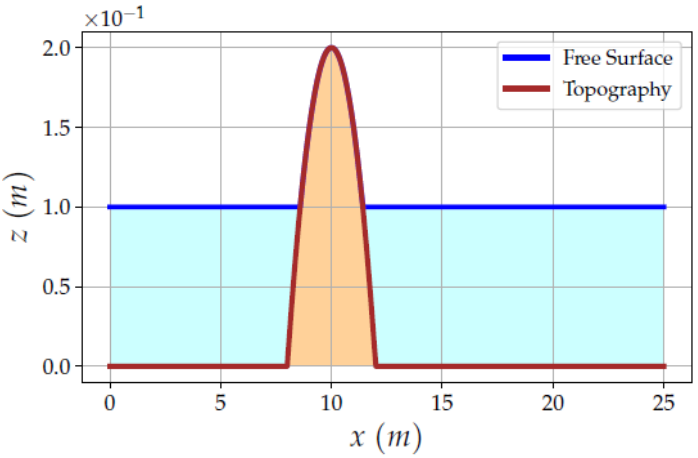


Figure 3.3: Emerged bump, topography and free surface at steady state.

Following the same presentation style as in the previous case, we present the responses of the PINN for the initial and final times of the simulations, for all three variables of interest, in Figure 3.4. The response of the SWASHES library has been overlaid for comparison. The errors for each case are shown in Table 3.3.

Table 3.3: L^2 errors for emerged bump simulations

Output variable	Metric	T=0sec	T=100sec
h	$\|h_{SW} - h_{PINN}\|_2 / \|h_{SW}\|_2$	0	3.6×10^{-10}
u	$\|u_{SW} - u_{PINN}\|_2$	0	7.9×10^{-8}
q	$\|q_{SW} - q_{PINN}\|_2$	0	5.2×10^{-11}

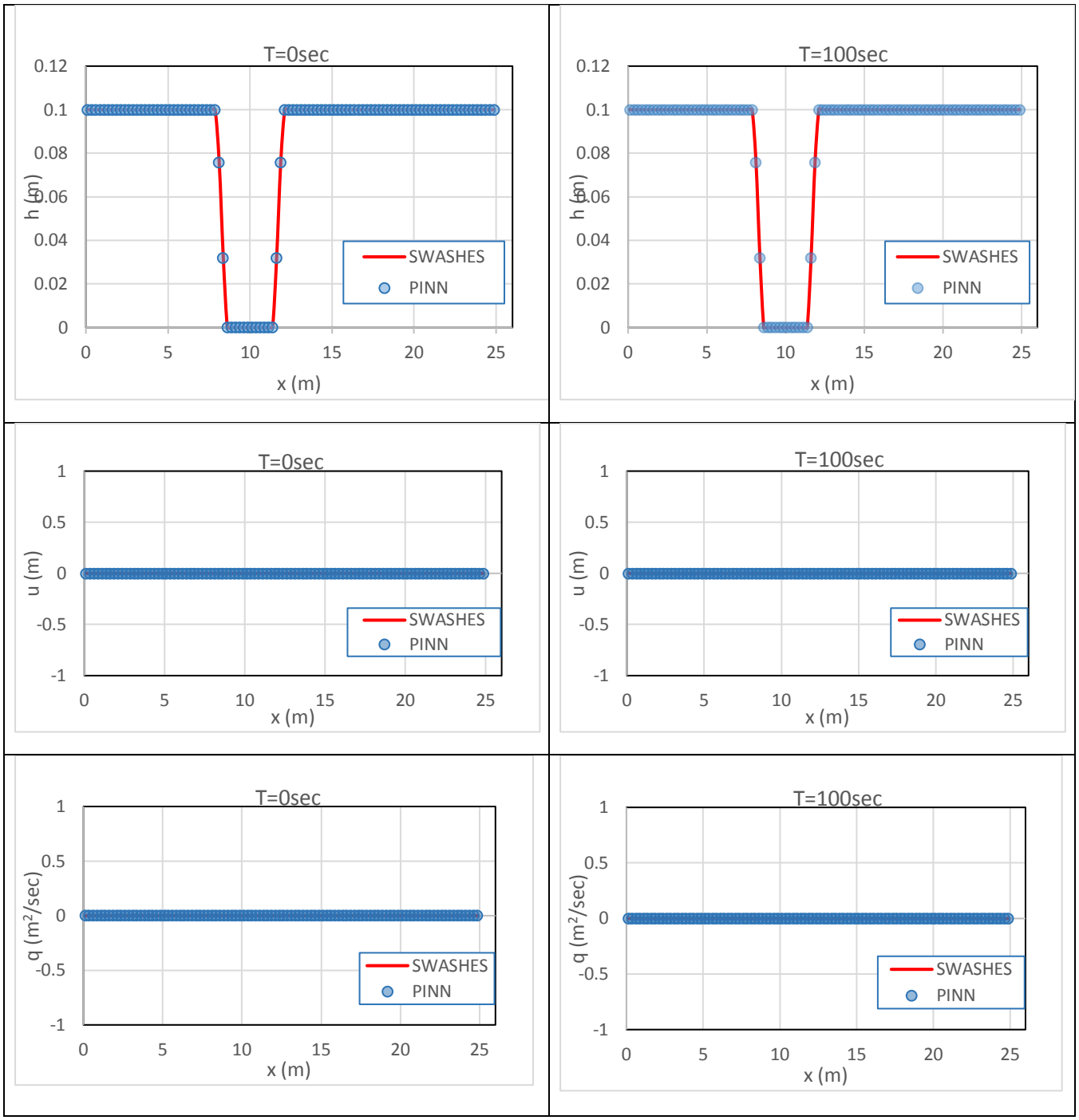


Figure 3.4: Simulation results for emerged bump. Left pane: h, u, q plots at time T=0s. Right pane: h, u, q plots for T=100s.

3.2.3 Subcritical flow

This case tests the ability of the PINN to catch steady states. The treatment of the impulsive boundary terms is also examined. The equations describing this case, are summarized below:

Initial Conditions: $\begin{cases} h = 2.0 - z\ m \\ q = 0\ \frac{m^2}{s} \end{cases}$

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

The initial conditions as well as the steady state are shown in Figure 3.5. The water height is constant when the topography is constant and it decreases (increases) when the bed slope increases (resp. decreases). The water height reaches its minimum at the top of the bump which is its center.

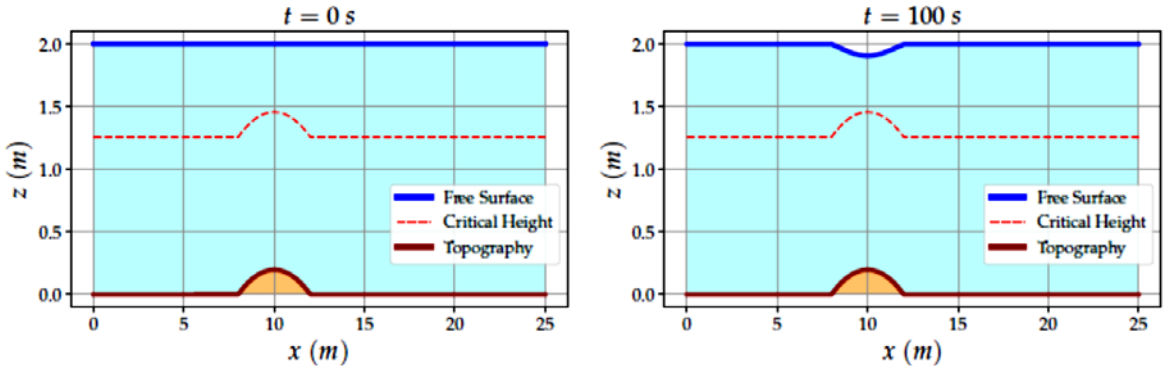


Figure 3.5: Left: the initial condition of subcritical flow. Right: the steady state of subcritical flow.

Following the same presentation style as in the previous case, we present the responses of the PINN for the initial and final times of the simulations, for all three variables of interest, in Figure 3.6. The response of the SWASHES library have been overlaid for comparison. The errors for each case are shown in Table 3.4.

Table 3.4: L^2 errors for subcritical flow simulations

Output variable	Metric	T=0sec	T=100sec
h	$\|h_{SW} - h_{PINN}\|_2 / \|h_{SW}\|_2$	0	3.5×10^{-5}
u	$\|u_{SW} - u_{PINN}\|_2$	0	6.8×10^{-5}
q	$\|q_{SW} - q_{PINN}\|_2$	0	5.2×10^{-6}

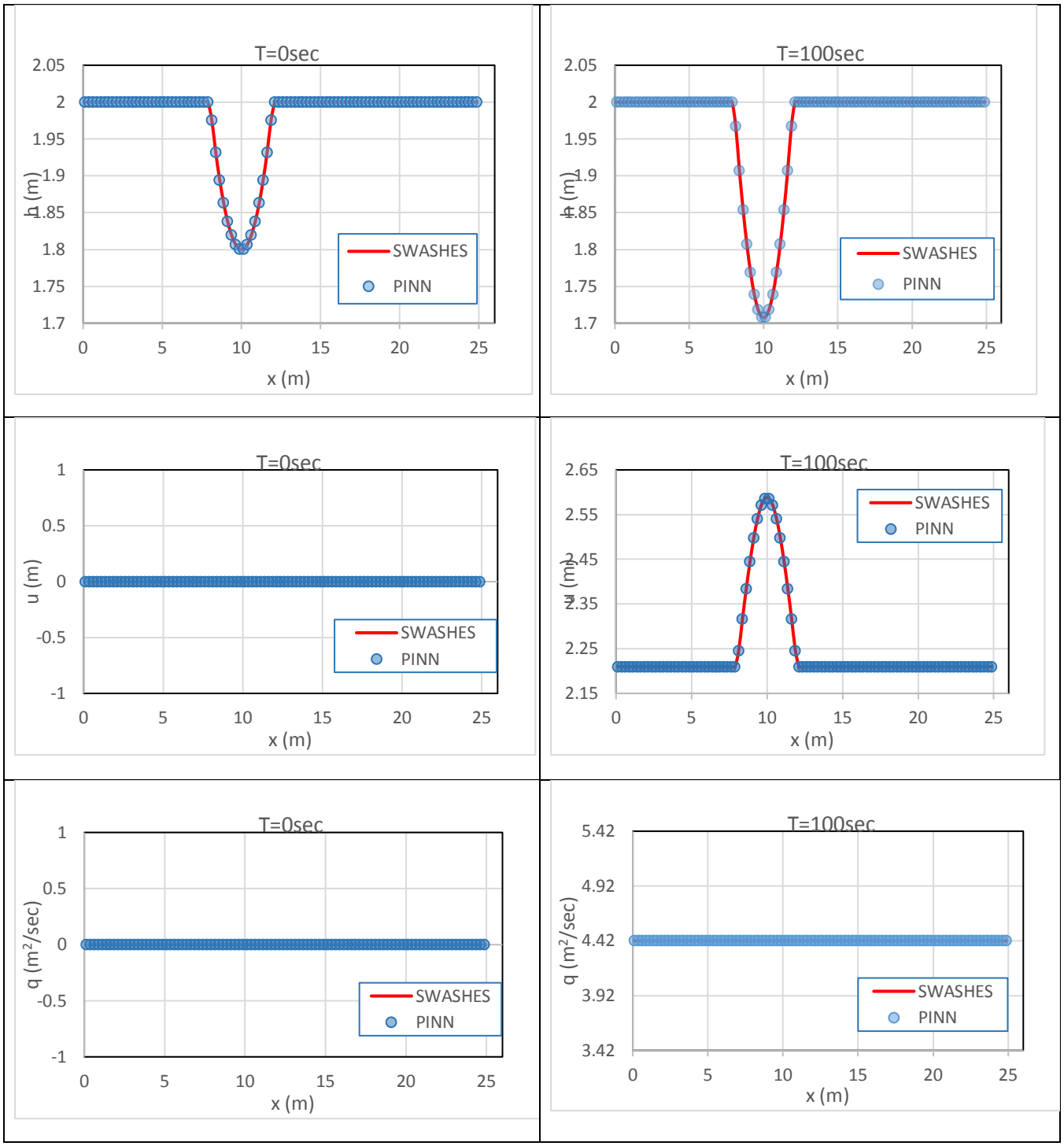


Figure 3.6 Simulation results for subcritical flow. Left pane: h, u, q plots at time T=0s. Right pane: h, u, q plots for T=100s.

3.3 Varying sampling strategies

In order to serve the purposes of the next section that investigates the efficacy of several popular sampling strategies, on the quality of the PINN solution for a SWE Riemann problem, we present here the sampling strategies that will be employed in the simulations. Though the list is not exhaustive, we believe, based on recent research literature, that the sampling strategies used in this study are the most frequently used in the PINN context [36], [52], [53].

3.3.1 Uniform sampling strategies

Here, we provide an overview of uniform sampling strategies that will be employed in the simulations. A brief presentation of the mechanisms employed in each case, are provided in Appendix A - I.

1. Equispaced uniform grid (Grid): The residual points are chosen as the nodes of an equispaced uniform grid of the computational domain.

2. Uniformly random sampling (Random): The residual points are randomly sampled according to a continuous uniform distribution over the domain. In practice, this is usually done using pseudo-random number generators such as the PCG-64 algorithm.

3. Latin hypercube sampling (LHS): The LHS is a stratified Monte Carlo sampling method that generates random samples that occur within intervals on the basis of equal probability and with normal distribution for each range.

4. Quasi-random low-discrepancy sequences:

(a) Halton sequence (Halton): The Halton samples are generated according to the reversing or flipping the base conversion of numbers using primes.

(b) Hammersley sequence (Hammersley): The Hammersley sequence is the same as the Halton sequence, except in the first dimension where points are located equidistant from each other.

(c) Sobol sequence (Sobol): The Sobol sequence is a base-2 digital sequence that fills in a highly uniform manner.

5. Random sampling with resampling (Random-R). This is a variation of the Uniformly random sampling strategy, but samples are resampled every N iterations. The resampling period N is a hyperparameter that needs to be determined, however, a typical value of N around 100-200 is often used as default value.

Figure 3.7 presents the results of the coverage generated from each of the aforementioned sampling strategies, for a typical square domain $[0, 1]^2$.

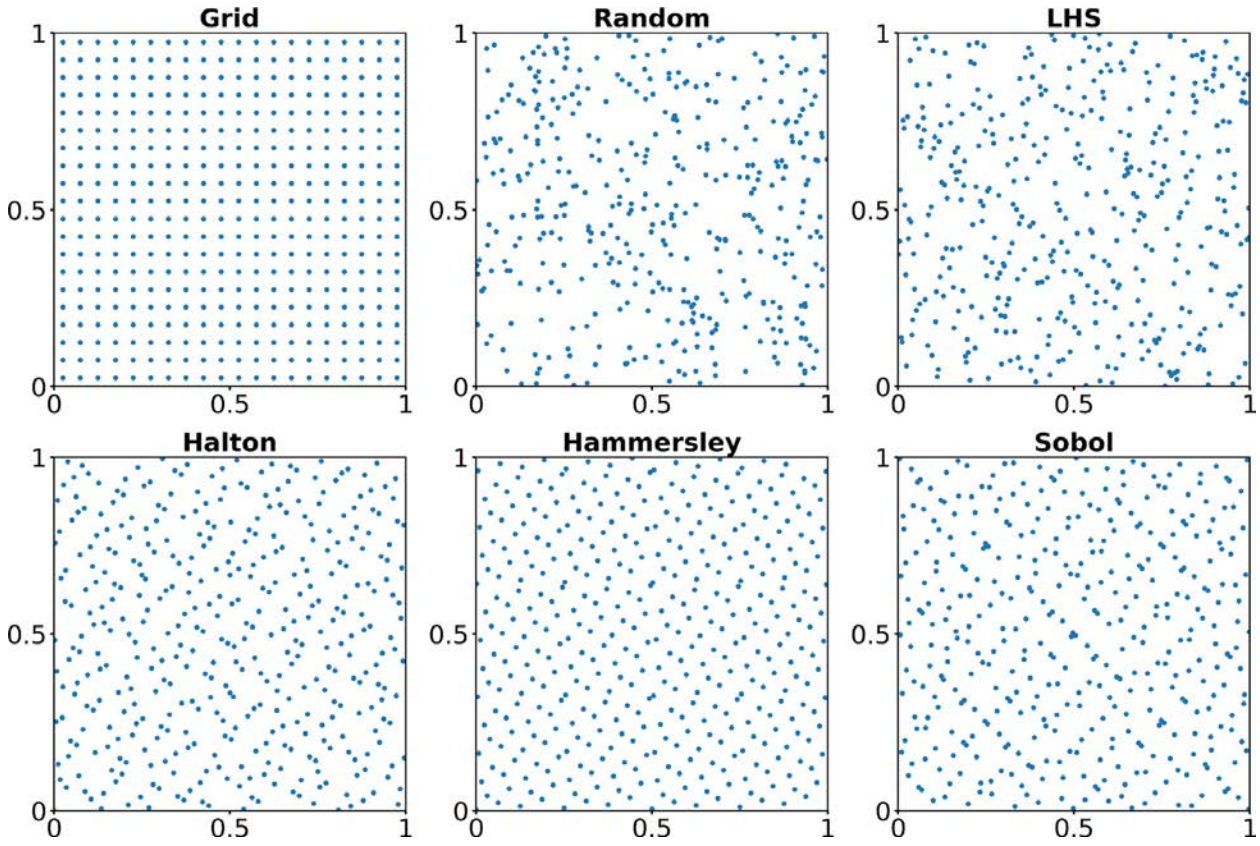


Figure 3.7: Examples of 400 points generated in $[0, 1]^2$ using different uniform sampling methods (From [52])

3.3.2 Non-uniform adaptive sampling strategies

In this paragraph, we briefly explain the non-uniform adaptive sampling strategies that we will employ.

1. Residual-based Adaptive Refinement with Greed (RAR-G) [35]. RAR-G aims to improve the distribution of residual points during the training process, by sampling more points in the locations where the PDE residual is large. Specifically, after every certain number of iterations, RAR-G adds new points in the locations with large PDE residuals. RAR-G only focuses on the points with large residual and thus it is a greedy algorithm. The RAR-G algorithm is outlined in the box labeled as ***Algorithm 4***.

Algorithm 4: RAR-G.

1 Sample the initial residual points T using one of the methods in paragraph 3.3.1;
2 Train the PINN for a certain number of iterations;
3 **repeat**
4 Sample a set of dense points $\mathcal{S}_0$ using one of the methods in paragraph 3.3.1;
5 Compute the PDE residuals for the points in $\mathcal{S}_0$;
6 $\mathcal{S} \leftarrow$ m points with the largest residuals in $\mathcal{S}_0$;
7 $T \leftarrow T \cup \mathcal{S}$;
8 Train the PINN for a certain number of iterations;
9 **until** the total number of iterations or the total number of residual points reaches the limit;

2. Residual-based Adaptive Distribution (RAD) [53]. In this strategy, all the residual points are resampled according to a probability density function (PDF) p(x) proportional to the PDE residual. Specifically, for any point x, we first compute the PDE residual $\varepsilon(\boldsymbol{x}) = |f(x, \hat{u}(\boldsymbol{x}))|$ and then compute a probability as:

$$p(\boldsymbol{x}) \propto \frac{\varepsilon^k(\boldsymbol{x})}{\mathbb{E}[\varepsilon^k(\boldsymbol{x})]} + c \qquad (3.2)$$

where $k \geq 0$ and $c \geq 0$ are hyper-parameters that determine the shape of the sampling PDF. The PDF is normalized, so that:

$$\tilde{p}(\boldsymbol{x}) = \frac{p(\boldsymbol{x})}{A}$$

where $A = \sum_{x \in S_0} p(\boldsymbol{x})$ is a normalizing constant. Practice has shown that a good default value for the hyperparameters is $k = 1$ and $c = 1$. Note also, that if $k = 0$ or $c \to \infty$, then RAD reduces to Random-R strategy. The RAD is outlined in the box labeled as ***Algorithm 5***.

Algorithm 5: RAD.

1 Sample the initial residual points T using one of the methods in paragraph 3.3.1;
2 Train the PINN for a certain number of iterations;
3 **repeat**
4 $T \leftarrow T \cup \{$A new set of points randomly sampled according to PDF of equation (3.2)$\}$;
5 Train the PINN for a certain number of iterations;
6 **until** the total number of iterations reaches the limit;

The two hyperparameters k and c control the profile of $p(\boldsymbol{x})$ and thus the distribution of the sampled points. An illustrative example of the coverage achieved for several combinations of values of these parameters, in case the loss function had the form (a bell-shaped function that resembles a local maximum of the loss function):

$$\varepsilon(x, y) = 2^{40} x^{10} (1 - x)^{10} y^{10} (1 - y)^{10}$$

is shown in Figure 3.8. It is noteworthy that when k=0, the distribution generated resorts to the Uniform distribution. As the value of k increases, more residual points with large PDE residuals are sampled. As the value of c increases, the residual points exhibit a tendency to be uniformly distributed. Compared to RAR-G, RAD provides more freedom to balance the points in the locations with large and small residuals, by tuning k and c. The optimal values of k and c are clearly problem-dependent.

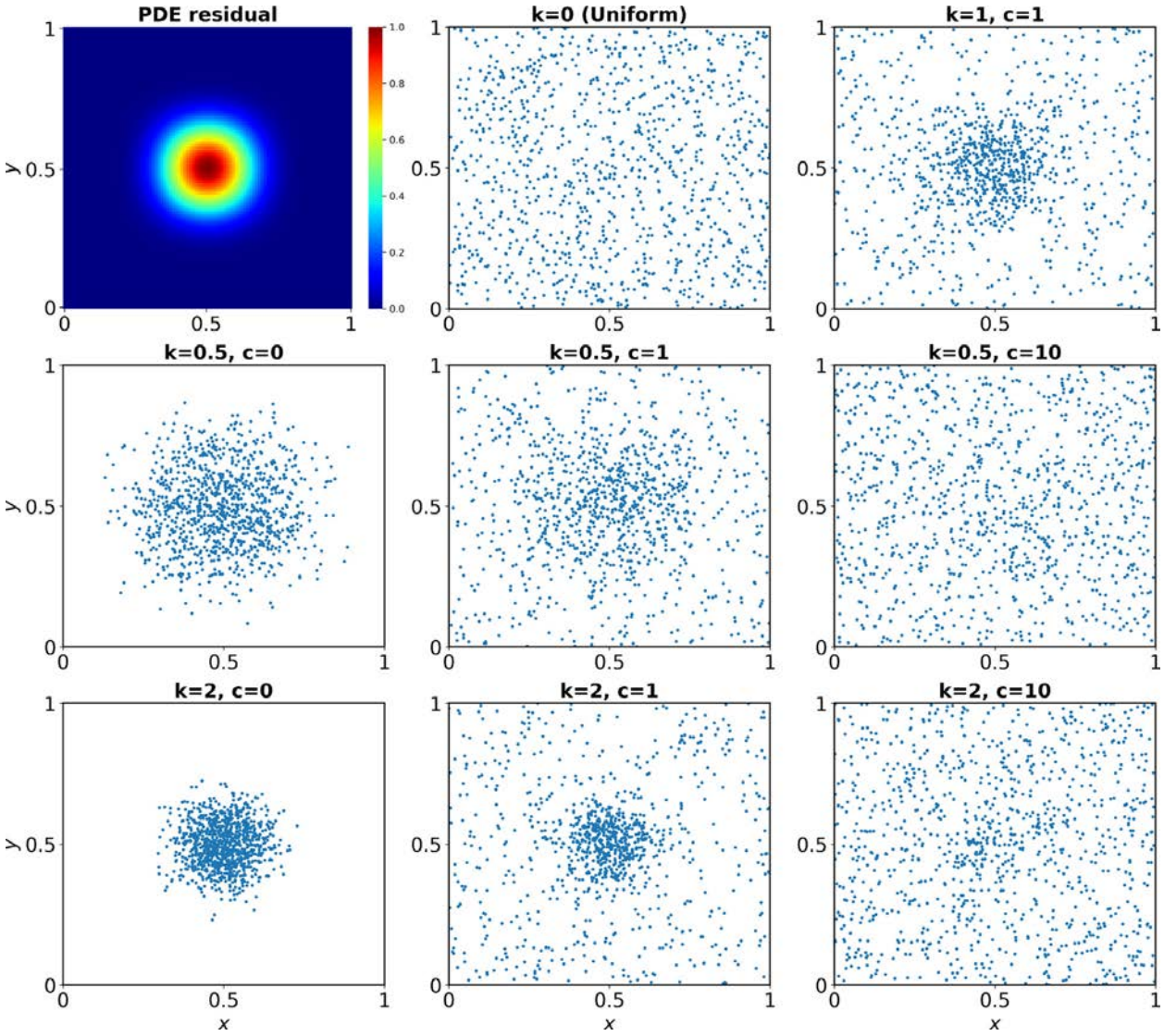


Figure 3.8: Examples of 1000 residual points sampled by RAD with different values of k and c for the PDE residual ε(x, y) (From [52])

3. Residual-based Adaptive Refinement with Distribution (RAR-D) [52]. This strategy stands between the previous two, where similar to RAR-G, RAR-D repeatedly adds new points to the training dataset; and similar to RAD, the new points are sampled based on the PDF of the residuals. Note that when $k \to \infty$, only points with the largest PDE residual are added, which recovers RAR-G. Once again, the optimal values of k and c are problem-dependent. The RAR-D is outlined in the box labeled ***Algorithm 6***.

Algorithm 6: RAR-D.

1 Sample the initial residual points T using one of the methods in paragraph 3.3.1;
2 Train the PINN for a certain number of iterations;
3 **repeat**
4 $\mathcal{S} \leftarrow$ m points randomly sampled according to PDF of equation (3.2);
5 $T \leftarrow T \cup \mathcal{S}$;
6 Train the PINN for a certain number of iterations;
7 **until** the total number of iterations or the total number of residual points reaches the limit;

3.4 Simulation results

In this section we investigate the relationship between the accuracy of the solution provided by the PINN that solves the SWE and the sampling strategy used to train the PINN solver. It has been shown in the literature that for several problems, the desired accuracy goals are achieved with less computational effort (less training iterations), when a suitable sampling method is adopted for the generation of training data for the PINN [53] [36]. The case study will be a SWE setting, namely the case of a dam break in a wet domain, without friction. This is a well-known Riemann problem in the SWE context and the efficacy of some sampling methods is expected to be challenged, mainly due to the discontinuity of the data at the initial phase, as well as the steepness of the solutions.

3.4.1 Case study: Dam break on a wet domain.

This case is known as the Stoker's solution and constitutes a classical Riemann problem. It 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 considered instantaneous, the bottom is flat and there is no friction. The initial conditions for the water height are:

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

and the initial condition for the velocity is:

$$u(x,0) = u = 0\ m/sec$$

At time $t \geq 0$, we have a left-going rarefaction wave that reduces the initial depth $h_l = 5 \times 10^{-3}\ m$ into a height $h_m < h_l$ and a right-going shock that increases the initial height $h_r = 1 \times 10^{-3}\ m$ into h_m. The system of SWE for this setting is:

$$\frac{\partial}{\partial t}\left(\begin{bmatrix} h \\ hu \end{bmatrix}\right) + \frac{\partial}{\partial x}\left(\begin{bmatrix} hu \\ hu^2 + \frac{g}{2}h^2 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

3.4.2 PINN setting.

The purpose of this section is two-fold: on the one hand, to solve the specific SWE problem using PINN and on the other hand, to evaluate the performance of the PINN when popular sampling strategies are used, as a solver applied on a difficult problem that involves coupled PDEs. While we do not neglect the former, the emphasis of this section is rather on the latter, because the ability of the PINN to provide solutions that exhibit steep regions, in an accurate and efficient manner, is an open research problem.

The evaluation of the results will be done using the L^2 norm relative error, for both output variables of interest, that is the height of the water above topography, h and the horizontal velocity u. In both cases, the ground truth is drawn from the SWASHES library.

$$L_h^2 = \frac{\|h_{SWASHES} - h_{PINN}\|_{L^2}}{\|h_{SWASHES}\|_{L^2}},\ L_u^2 = \frac{\|u_{SWASHES} - u_{PINN}\|_{L^2}}{\|u_{SWASHES}\|_{L^2}}$$

The evaluation will also include plots where the solutions of the PINN will be superimposed to the plot of the ground truth solutions, for both output variables, at the end of the time domain, $t = 6sec$. Moreover, the learning curves will also be presented, which show the evolution of the Train loss with respect to training time (number of iterations). The Train Loss is calculated from the Loss function of the PINN, by applying the samples acquired at the specific iteration. It is common practice in the evaluation of the PINN, to also include the evolution of the Test loss, which is calculated by using a reserved set of samples, constant throughout the training process, unseen from the PINN, to evaluate the loss function of the PINN. This test helps to assess the generalization error of the PINN.

The DeepXDE library that we use to build the simulation code, provides another set of illustrative plots that present the evolution of the solution of the output variables in the spatio-temporal domain of the simulation. Examples of theses plots are shown in Figure 3.9. However, due to the high level nature of the DeepXDE library, user control over the features of these plots is limited. Thus, to simplify the analysis, we do not show these plots in the subsequent analysis.

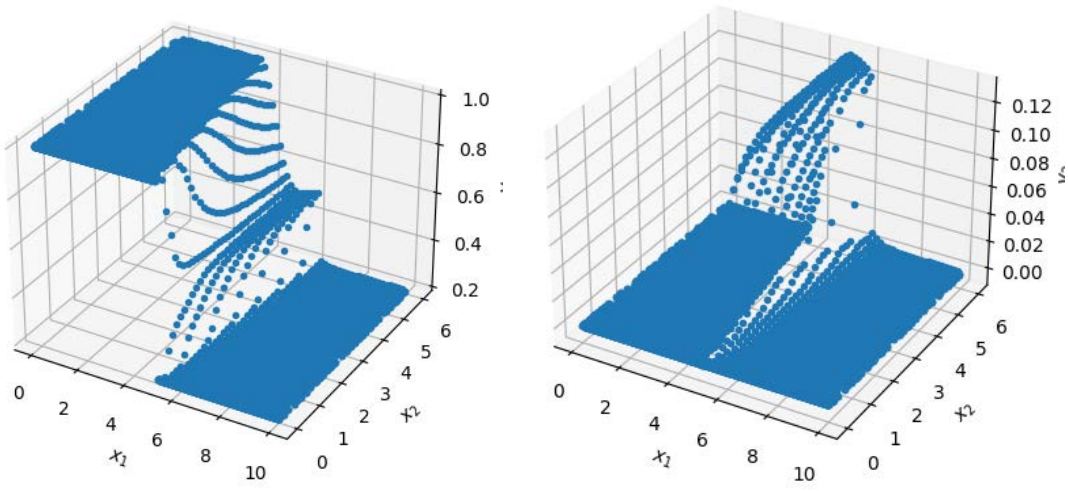


Figure 3.9: Illustration of the evolution of the provided PINN solution for the height (left plot) and the velocity (right plot) in the spatio-temporal domain.

Note that variable x_1 shown in Figure 3.9, corresponds to space (covers the range 0-10 meters), while variable x_2 corresponds to time (covers the range 0-6 seconds). The dots correspond to individual test samples engaged in the training for a specific iteration.

In order to formulate an unbiased evaluation setting for all sampling strategies, we use the same PINN architecture for all experiments. The number of training iterations is also held the same for all cases, specifically 200,000. The optimization algorithm used was ADAM, with a learning rate of 0.0001. Due to the adaptive sampling nature of the RAR-G and RAR-D algorithms, that do not retain the same number of samples throughout the training process, the sample size was held the same for all strategies, up to iteration number 20,000. Extended experimentation has shown that this number of iterations is enough for a SWE problem of this nature, to allow for sufficient initialization of neural network parameters. Beyond iteration 20,000, we allow each strategy to reveal its potential. The features common to all tests conducted, have been gathered in Table 3.5.

Table 3.5: Common simulation features

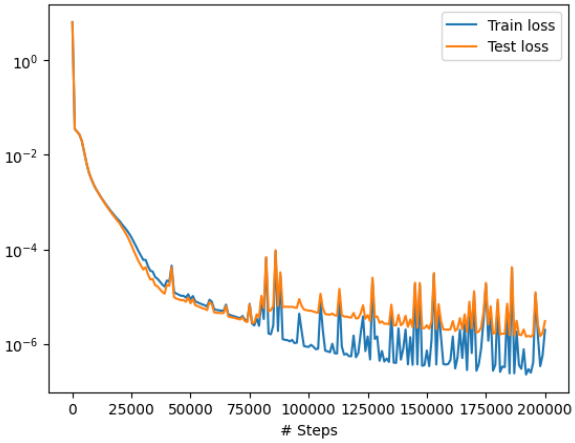
Feature	Value
PINN depth (number of layers)	4
PINN layer width (neurons per hidden layer)	60
Optimization algorithm	ADAM
Learning rate	0.0001
Activation function	tanh (hyperbolic tangent)
Number of iterations	200,000
Number of collocation points (PDE residuals)	4,000
Number of initial conditions points	1,000
Number of test points	2,000

Because the result of PINN is based on statistical inference rather than on deterministic numerical calculations (as is the case of conventional numerical schemes), it has randomness due to random sampling, network initialization and optimization. Therefore, for each case we run the same experiment for a sufficient number of times and then computed the mean and the standard deviation of the measured errors.

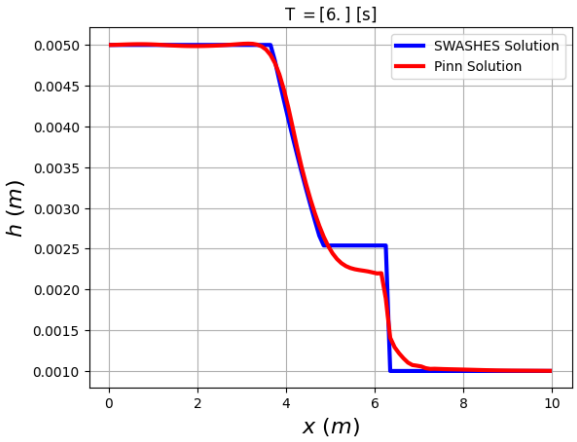
The presentation of the results in the following paragraphs, implies a loose taxonomy of the sampling schemes: uniformly distributed non-adaptive vs nonuniform adaptive is the first level of taxonomy. The second level has to do with whether the residual points remain fixed or not throughout the training process. And finally, an extra level of grouping inside the uniformly distributed non-adaptive group, stems from the discrepancy level of each scheme, since some of them have been intentionally invented to provide low-discrepancy sampling sequences, especially in a multi-dimensional setting (Halton, Hammersley and Sobol).

3.4.2.1 Uniformly distributed non-adaptive sampling

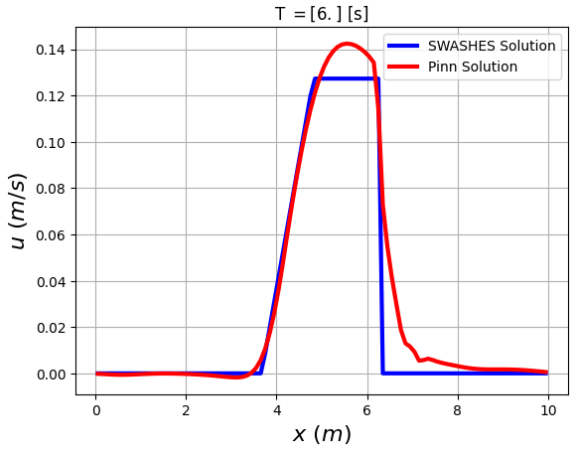
The sampling strategies of this group include Uniform, Random, Latin Hypercube (LHS), Halton, Hammersley, Sobol and Random sampling with Resampling (Random-R). For each case, we present the learning curves (train and test losses) and the profile for height and velocity solution at the end of the simulated time ($T_{max} = 6sec$). It is emphasized here that the learning profiles and the solution graphs presented in the sequel, are only indicative for each sampling scheme and are not intended to show a rigid behavior. The learning profiles and the solution graphs taken for every repetition of the same experiment, are only variations of the graphics shown in the following figures for the corresponding scheme. The standard deviation calculated for each scheme, indicates the adherence of the corresponding solution to the presented figures.



a

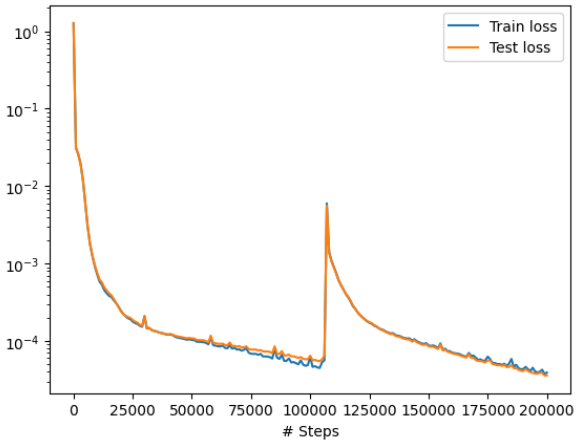


b

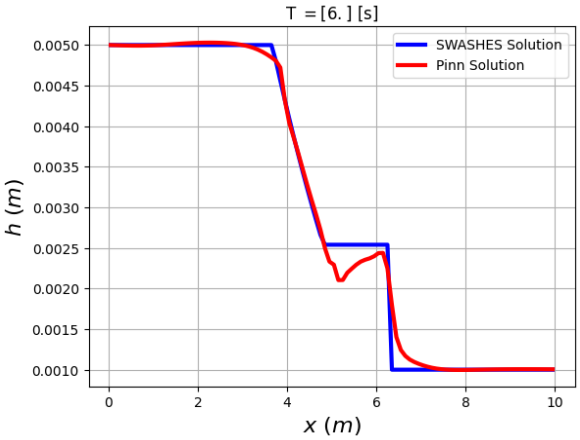


c

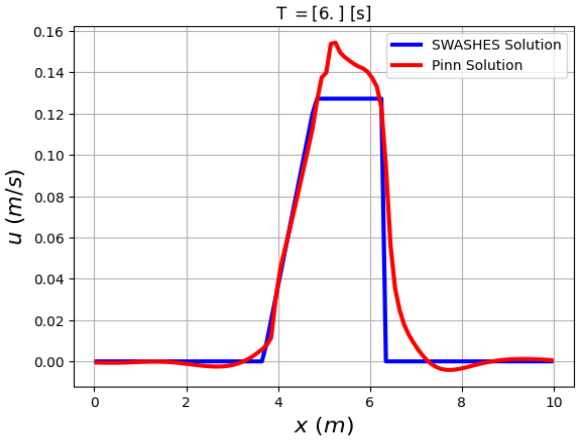
Figure 3.10: Grid sampling: a. Learning curves (train and test losses), b. Height solutions (PINN and SWASHES), c. Velocity solutions (PINN and SWASHES)



a

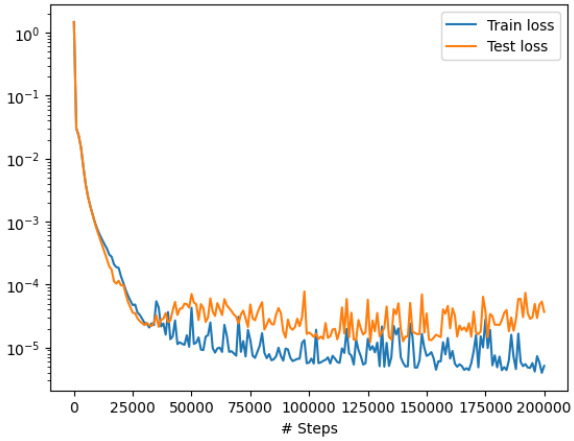


b

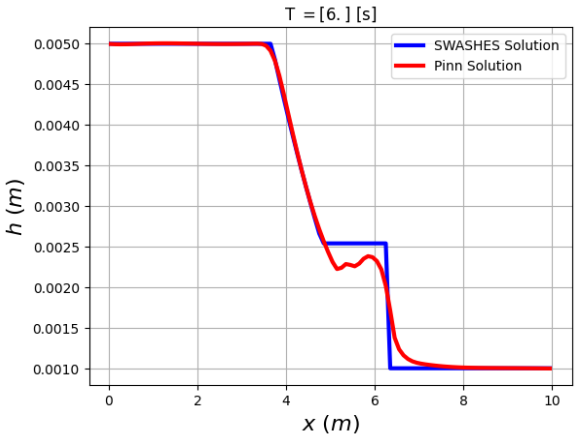


c

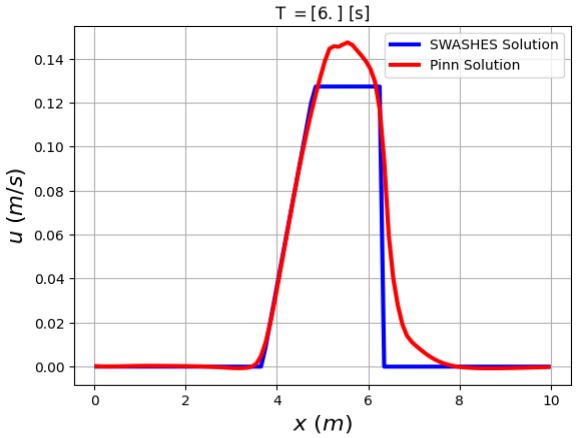
Figure 3.11: Random sampling: a. Learning curves (train and test losses), b. Height solutions (PINN and SWASHES), c. Velocity solutions (PINN and SWASHES)



a

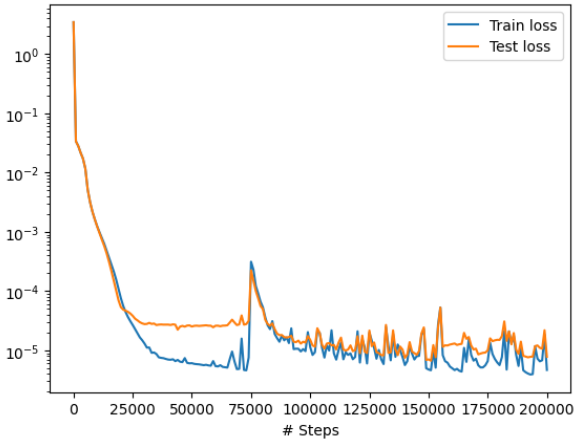


b

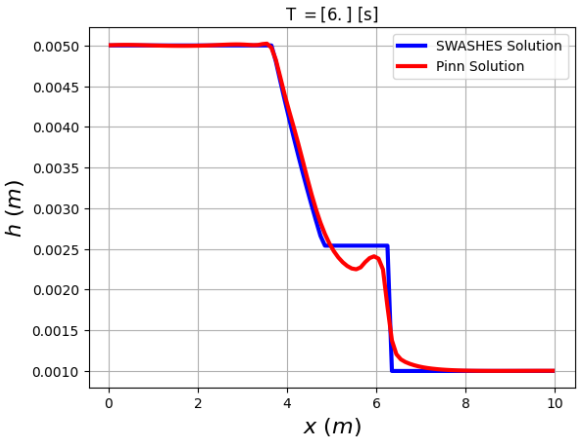


c

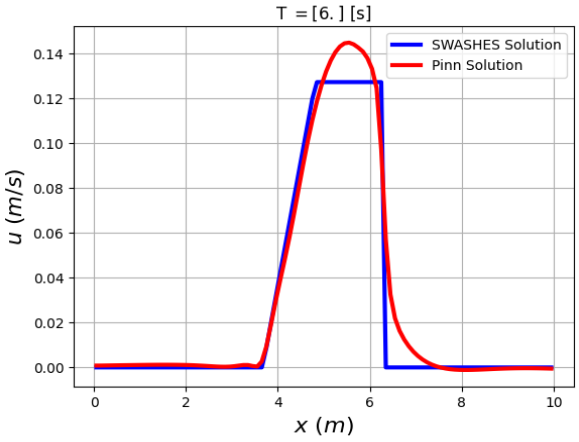
Figure 3.12: LHS sampling: a. Learning curves (train and test losses), b. Height solutions (PINN and SWASHES), c. Velocity solutions (PINN and SWASHES)



a

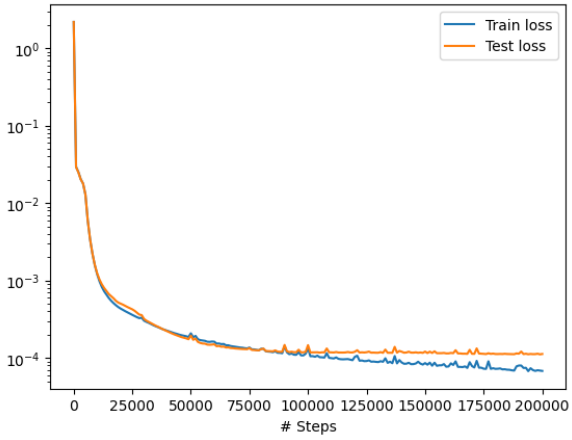


b

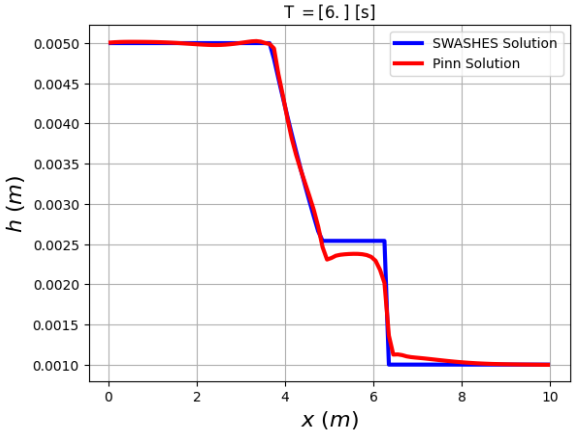


c

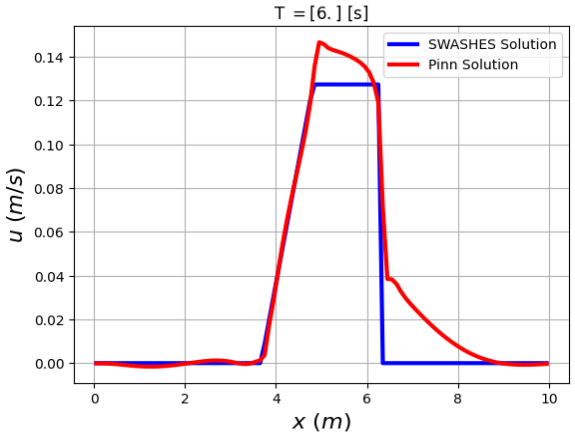
Figure 3.13: Halton sampling: a. Learning curves (train and test losses), b. Height solutions (PINN and SWASHES), c. Velocity solutions (PINN and SWASHES)



a

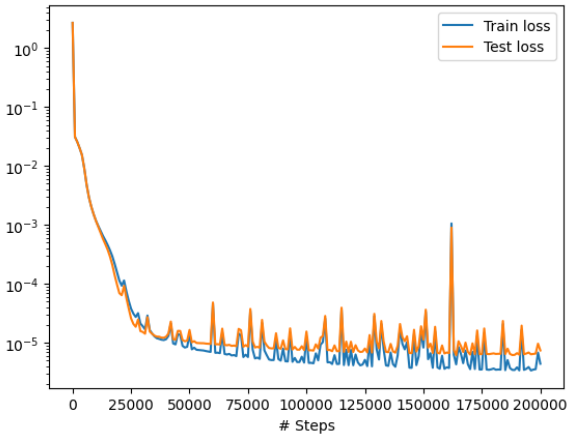


b

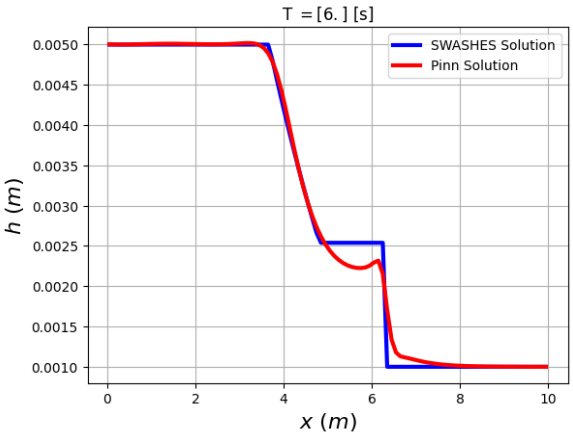


c

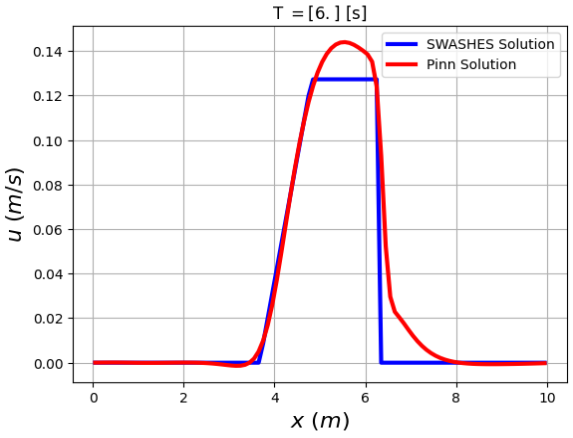
Figure 3.14: Hammersley sampling: a. Learning curves (train and test losses), b. Height solutions (PINN and SWASHES), c. Velocity solutions (PINN and SWASHES)



a

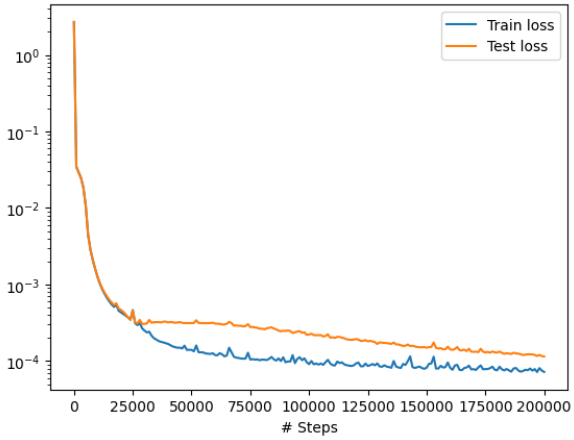


b

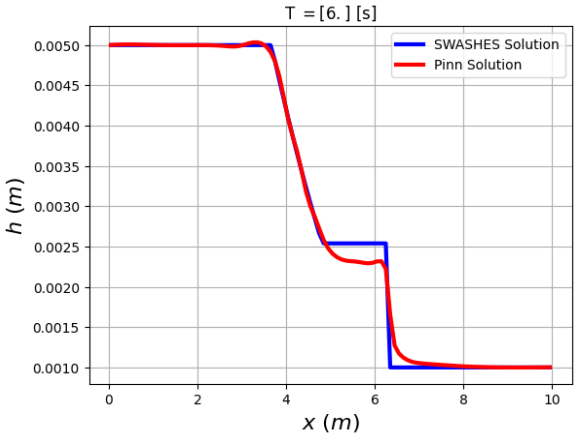


c

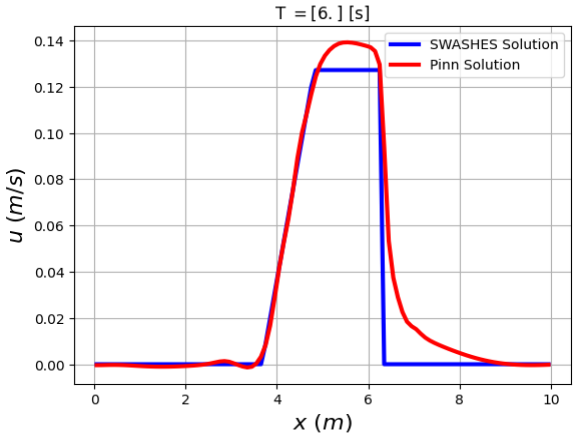
Figure 3.15: Sobol sampling: a. Learning curves (train and test losses), b. Height solutions (PINN and SWASHES), c. Velocity solutions (PINN and SWASHES)



a



b



c

Figure 3.16: Random sampling with resampling (Resampling period: 1000):
a. Learning curves (train and test losses), b. Height solutions (PINN and SWASHES),
c. Velocity solutions (PINN and SWASHES)

With reference to the Random sampling with resampling scheme (Random-R), it is worth noting that the resampling period N is an important hyperparameter that affects the performance of the specific scheme, so an extra step was taken in order to optimize this parameter. A number of tests were conducted with varying N, covering a wide range of values (10, 100, 500 and 1000). The results are shown in Figure 3.17.

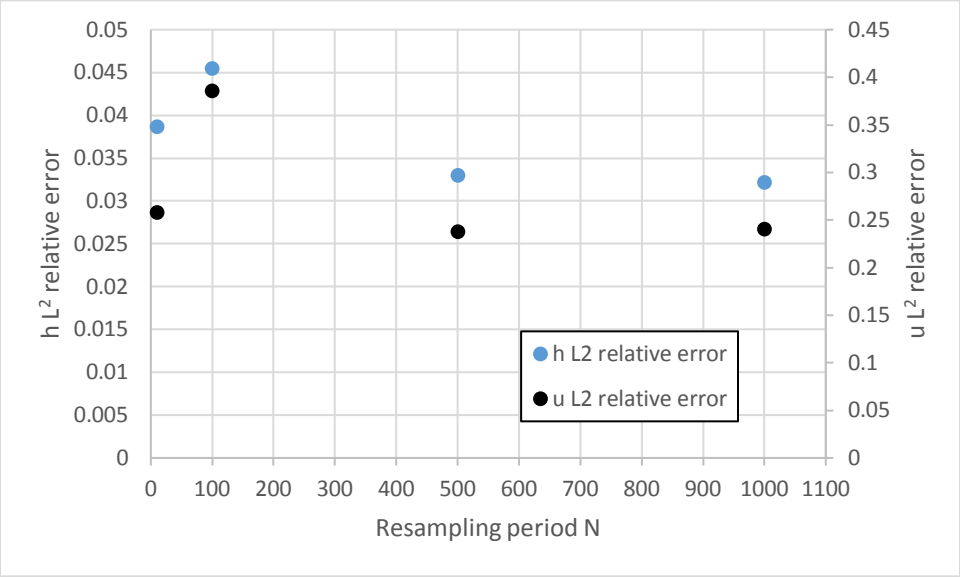
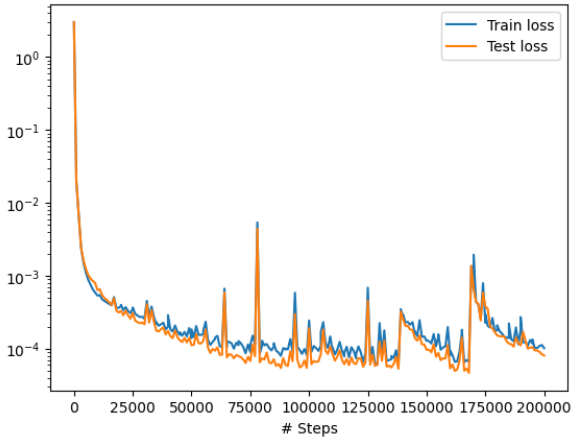


Figure 3.17: Dependence of L^2 relative error of height and velocity solution, on resampling period N.

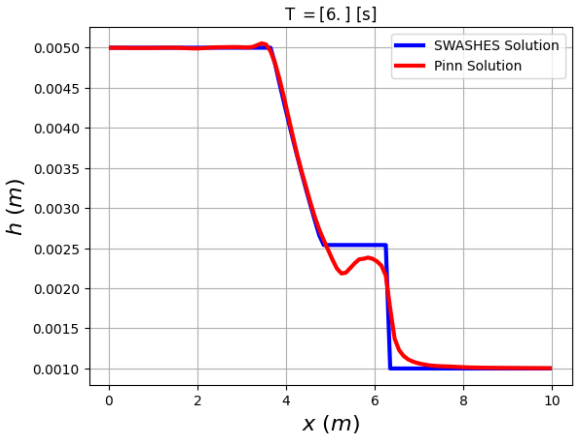
We note that the L^2 relative errors tend to decrease with increasing N. Respecting the statistical variation of the noted errors, we used a resampling period of N=1000, although the errors stemming from a resampling period of N=500 are comparable. During this study, however, it was noted that the solutions taken from the experiments with N=1000 where more smooth (less ripple around steep regions). The lateral advantage with using a larger resampling period is decreased computational cost, since the resampling process is engaged less times per simulation session.

3.4.2.2 Non-uniform adaptive sampling

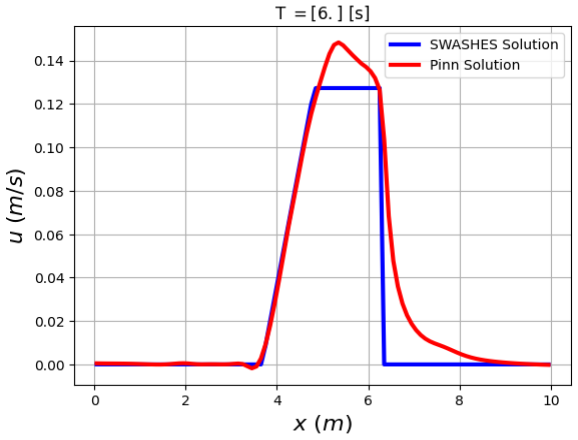
In this paragraph, we present the training results when the three non-uniform adaptive schemes are used, namely RAR-G, RAD and RAR-D. As noted in the previous section, the adherence of the results to the presented figures, strongly depends on the standard deviation. Note that the resampling period for all sampling schemes has been set to 5000 iterations. This means that, after an initial training phase lasting for 20,000 iterations, where the samples remain fixed (sampled randomly), the training is re-initiated every 5,000 iterations, with different samples. The mechanisms for sample acquisition for each scheme, are specified by the algorithms presented earlier, namely ***Algorithm 4***, ***Algorithm 5*** and ***Algorithm 6*** for RAR-G, RAD and RAR-D respectively.



a

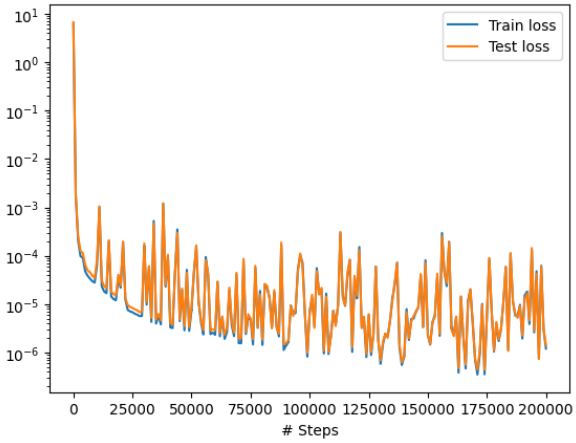


b

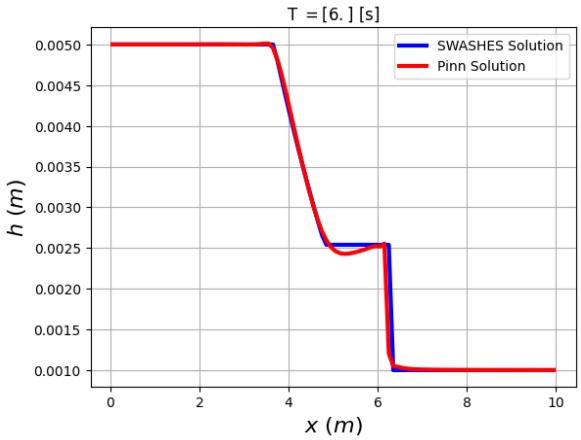


c

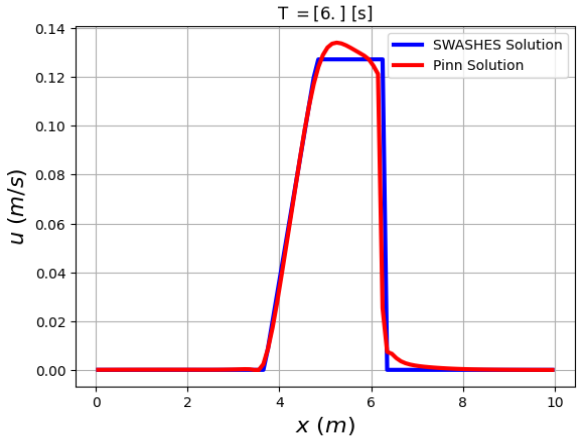
Figure 3.18: RAR-G sampling: a. Learning curves (train and test losses), b. Height solutions (PINN and SWASHES), c. Velocity solutions (PINN and SWASHES)



a

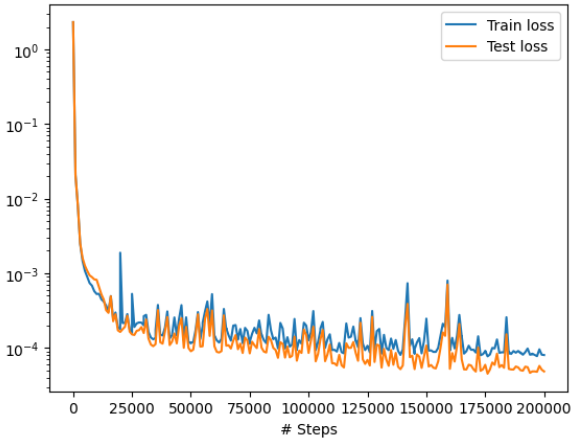


b

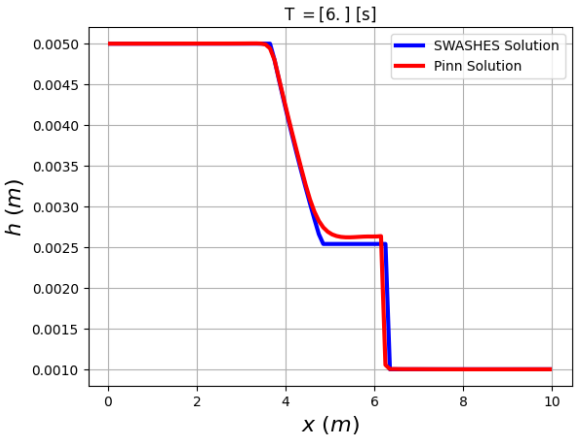


c

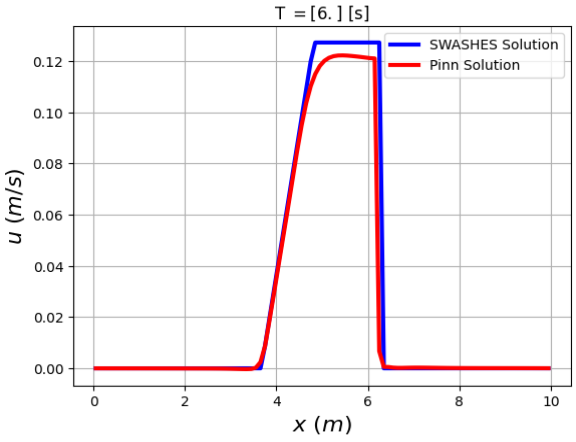
Figure 3.19: RAD sampling (k=1, c=1): a. Learning curves (train and test losses), b. Height solutions (PINN and SWASHES), c. Velocity solutions (PINN and SWASHES)



a



b



c

Figure 3.20: RAR-G sampling (k=2, c=2): a. Learning curves (train and test losses), b. Height solutions (PINN and SWASHES), c. Velocity solutions (PINN and SWASHES)

We note here that, as explained in section 3.3.2, sampling schemes RAD and RAR-D rely on the two hyperparameters, k and c. Fine-tuning of those hyperparameters requires a lot of effort, as was shown in reference [52], where they were introduced. We resorted to the default values suggested by the authors.

3.4.3 Overview of experimental results

In this section we provide the error statistics from the experiments conducted in the previous section. As mentioned previously, the statistics are organized based on a loose taxonomy of the sampling schemes examined. The results include the mean error and 1 Standard Deviation from the mean. They are gathered in Table 3.6.

Table 3.6: L^2 relative errors of the PINN solution for the SWE problem "Dam break on wet domain".

Taxonomy		Scheme	height h	velocity u
Uniformly distributed non adaptive sampling	Fixed residual points	Grid	0.095±0.01	0.458±0.05
		Random	0.073±0.05	0.250±0.18
		LHS	0.041±0.02	0.225±0.11
		Halton	0.036±0.02	**0.182±0.12**
		Hammersley	**0.030±0.02**	0.241±0.19
		Sobol	0.046±0.04	0.232±0.01
	Resampling of residual points	Random-R	0.032±0.03	0.240±0.23
Non-uniform adaptive sampling		RAR-G	0.037±0.01	0.284±0.09
		RAD	**<u>0.015±0.01</u>**	**<u>0.078±0.02</u>**
		RAR-D	**0.018±0.01**	**0.083±0.03**

Referring to Table 3.6, bold font indicates the smallest three errors for each one of the output variables and underlined text indicates the smallest error for each variable. Our main findings from the results are as follows:

- As can be seen, the RAD sampling scheme exhibits the smallest errors, with a small standard deviation. The RAR-D also yields errors of similar magnitude and variance.
- The low discrepancy algorithms constitute the next group of schemes, in terms of performance, since Hammersley and Halton provide very good results. Random-R which stands in between the non-adaptive group and the adaptive group, also provides comparable errors.
- Among the fixed residual point schemes, the Grid and the Random provide the largest errors.

- The fact that the schemes based on resampling, provide better results than the other schemes, is compatible to intuition: a sampling scheme that is able to refresh (and increase in some cases) the samples used to calculate the loss, based on the distribution of the loss from previous iterations, has the potential to minimize the loss faster, compared to a scheme that is using the same, fixed set of samples throughout the training process. For the Riemann problem that we are trying to solve, this virtue of the adaptive schemes is of paramount importance. Note however, that the performance of all schemes is expected the same, in case of a problem with smooth solution.
- Based on the profiles of the solutions depicted in the Figures provided, we notice that the PINN has predicted correctly the location of the shock, for all sampling schemes. However, we observe a variation regarding the response of the PINN depending on the scheme, in the vicinity of the shock: with the exception of RAD and RAR-D, in most other cases, the solution tends to deviate from the shock, following a smooth curve, either before the shock or after the shock. We also note a ripple in the profiles of all schemes (excluding RAD and RAR-D), which is an expected behavior of most numerical schemes, in regions where the approximated curve is steep.
- We note that the predicted solution for the height, independent of the scheme utilized for sampling, is more successful (less error) that the solution provided for the velocity. We believe that this is related to the fact that the exact solution for the height, starts at time t=0sec with a shock which is imposed by the initial conditions and ends to a smoother profile. On the other hand, the solution for velocity follows a different learning path, because it starts at time t=0sec from a zero value and ends to a shock. Because initial conditions are incorporated into the loss terms involved in the training of the PINN, the PINN is forced to learn the initial conditions, with minimal error. This is in contrast to the solution for the velocity at the end of simulation time t=6sec, where the PINN has to learn this solution from the dynamics of the physics laws incorporated into the loss terms.

4 CONCLUSIONS

4.1 General remarks

In this thesis we investigated the applicability of the Physics-Informed Neural Network concept on the solution of the Shallow Water Equations for several data, boundary and initial conditions. The main focus of the work presented, was on the characterization of the efficiency of the most popular sampling strategies, based on the prediction error of the PINN, when these strategies are utilized in residual points sampling. As the accuracy and computational efficiency of the PINN is usually compared to conventional numerical schemes, the role of the sampling strategy in the optimization of the PINN solver for a certain problem, is a open research topic. The contribution of this work is related to the evaluation of the popular sampling strategies in terms of accuracy of the PINN solution, when utilized in the solution of a Riemann problem in the context of the SWE. To the best of our knowledge, the evaluation of the performance of sampling strategies when applied on systems of Partial Differential Equations and especially the SWE, has not been treated before in the literature.

The PINN has proven very efficient in terms of computational cost, as the dependence of the computational time from the number of residual points, follows a linear trend. The time required to process 1000, 2000 and 4000 residual points for 200K iterations on the Tensor Processing Unit (TPU) provided by the Google Colab environment, was 56’50’’, 65’14’’ and 87’23’’ respectively. This computational profile, promotes the use of the PINN, without considering the number of data as a main obstacle to experimentation, as opposed to conventional numerical schemes. Although the problem solved here is only two-dimensional, this virtue of the PINN becomes more important in multi-dimensional problems.

Although the environment of the Google Colab platform is powerful enough and efficient for the type of problems we considered, it is offered on a “best effort”, interactive and user shareable basis and not as a “guaranteed service”. As such, even the service for registered users, does not guarantee full time access to the high performance TPU runtime environment. This variation in service throughout this study, has affected the number of repetitions of the experiments conducted, since the time to complete an experiment on a CPU or even a low-end GPU, is in the order of several hours, as opposed to the 1-$1^{1/2}$ hours required on a TPU. Nevertheless, the statistics provided for each sampling strategy are still of value, because they represent the tendency for each strategy in terms of efficiency.

The use of the ADAM optimizer, being a first order stochastic gradient descent algorithm, has allowed to train the PINN sufficiently in all experirments at 200K iterations. However, combined with a second order optimizer, such as the quasi-Newton, Limited memory, Broyden–Fletcher–Goldfarb–Shanno (also known as L-BFGS) optimizer, as briefly noted in this document, would allow to take into account important information about the curvature of the loss function. This combination of optimizers would allow faster convergence and consequently fewer iterations, to reach the desired accuracy levels. Put in another perspective, for a training session combining ADAM followed by L-BFGS optimizer and for the same number of 200K iterations, the performance of all sampling strategies considered in this study, would be quite different (to the better). Unfortunately, at the time that this research was carried out, the L-BFGS with Tensorflow backend in Google Colab, was not functioning as expected (it would spontaneously stop training after 27-30 iterations without any improvement in the train loss, no matter the tuning of the L-BFGS performance parameters), so it was decided to proceed the experiments based solely on the ADAM optimizer.

It is also noteworthy that the PINN as a numerical framework, encapsulates a phase of hyperparameter fine-tuning, besides the pure time allocated on the computing machine to build, compile and train the PINN. This effort for hyperparameter fine-tuning is totally dependent on the experience gained on PINN design and the actual PDE problem at hand. Hyperparameter fine-tuning is a laborious trial-and-error process, which, nevertheless, is expected to smoothen, as experience is gained over time. Knowledge of the loss function landscape (saddle points, large valleys), of the peculiarities of the expected solution (smooth vs steep regions) and of the peculiarities of the boundary and initial conditions of the problem, plays an important role in the design of the PINN architecture and its fine-tuning in general.

4.2 Future directions

As noted in the previous section, one direction to investigate would be the combined use of ADAM and L-BFGS optimizers (probably by using a difference backend such as PyTorch, that supports the L-BFGS). After an initial training phase of 15,000-20,000 iterations with the ADAM optimizer, the use of the second-order L-BFGS optimizer is known to accelerate training, by contributing important information about loss function curvature, in numerous diverse settings. This is also expected to happen in the case of solving the SWE, even for demanding Riemann problems.

A second direction of special interest to scientific research, would be the use of two separate PINNs, each one devoted to the training of the two output variables (height and velocity). This idea for the three-dimensional (time, x and y) SWE case is shown in Figure 4.1, where each one of the separate PINNs, share the same inputs, but provide a single output (η denotes the height and u, v the velocity components). This separation is justified from the fact that a single PINN, as shown on the experimental results in this study, can learn the height solution better that the solution for the velocity, according to the errors observed

from all experiments. This implies that learning the solution for the velocity would require more samples. This logic promotes the use of separate sampling strategies for each PINN. In order to better visualize this when a single PINN is used for both output variables and a distributed-based adaptive sampling scheme has been chosen, it is possible that during a training iteration, the loss term for velocity has bigger value (eg one order of magnitude) than the loss term for height, thus the former would hide the latter from the mechanism of the sampling scheme that decides the distribution of the extra samples. In practice, the adaptive sampling schemes would acquire more samples to cover regions where the loss term for velocity is big, while neglecting regions where the loss term for height is moderate, but much smaller that the loss for velocity.

In the case of two separate PINNs however, the aforementioned illness can be remedied. The fact that the control of sample sizes is distinguished, the different distribution profiles between the losses of the "height PINN" and the "velocity PINN", would allow different sample sizes, saving the whole training process from computational cost. It may also occur that for one of the two variables (probably height), an adaptive sampling scheme would not even be required, thus resorting to a more conventional, but less complicated from a computational point of view, sampling scheme.

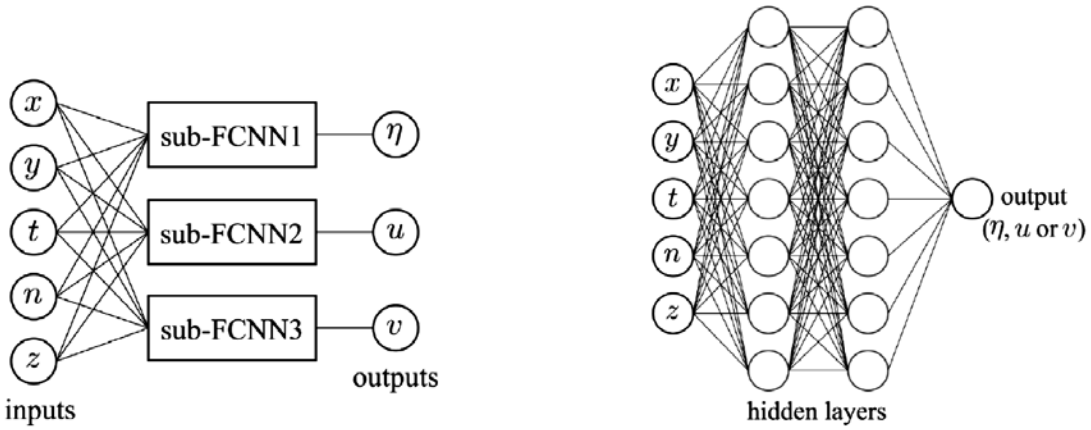


Figure 4.1: Separate Fully-Connected Neural Networks (FCNN) for 2-D SWE (adopted from *[47]*).

Another potential provided by the use of separate PINNs for each variable, is that different Neural Network architectures would also be possible to be designed and used. Customizing each of these PINNs separately in terms of depth (number of layers) and width (nodes per layer), is expected to further optimize the whole PINN solver for the problem at hand.

One more possibility that would improve the PINN performance is the use of additional loss terms as regularization terms. Not neglecting the implications of Occam's razor on machine learning, but based on

the results observed from the experiments conducted, the existence of regions of negative velocity values would justify the inclusion of an extra loss term, such as:

$$u(x,t) \geq 0 \frac{m}{sec}, \forall x,t \in \Omega \cup T$$

This term would facilitate the learning process, by limiting the search space to the direction of non-negative velocities. Following this same idea, an extra loss term could also be added, that would incorporate a number of training points for which the solution of the SWE problem is known. Selecting these points (also known as anchor points) in regions where the solution is known to be steep, would further facilitate the learning process, towards a more accurate and smooth solution. The use of anchor points carefully selected across the domain, is expected to alleviate the ripples observed in the solutions provided in some cases during the experiments, as well as the excessive peaks.

REFERENCES

[1] P. Domingos, "The Role of Occam's Razor in Knowledge Discovery," *Data Mining and Knowledge Discovery,* vol. 3, pp. 409-425, 1999.

[2] G. Karniadakis, "BINNS: Biophysics-Informed Neural Networks," 7 Apr. 2023. [Online]. Available: www.youtube.com/@IBSBiomedicalMathematicsGroup (youtube channel of BIMAG). [Accessed 15 May 2024].

[3] O. Delestre, C. Lucas, P.-A. Ksinant, F. Darboux, C. Laguerre, T. Vo, F. James and S. Cordier, "SWASHES: a compilation of Shallow Water Analytic Solutions for Hydraulic and Environmental Studies," *Int. Journal for Numerical Methods in Fluids,* vol. 72, no. 3, pp. 269-300, 2013.

[4] A. I. Delis, I. K. Nikolos and (editors), Shallow Water Equations in Hydraulics: Modelling, Numerics and Applications (Reprinted from "Water" Journal), MDPI, 2022.

[5] L. Jakobi and S. Krotsch, "Testing Physics-Informed Neural Networks for the Solution of Hyperbolic Conservation Laws," in *Finite Element Methods and Physics Informed Neural Networks*, Research in Groups - Numerical Mathematics and Applied Analysis at the University of Wurzburg, 2024, pp. 1-53.

[6] R. J. Leveque, Finite Volume Methods for Hyperbolic Problems, Seattle: Cambridge University Press, 2001.

[7] R. J. Leveque, Finite Difference Methods for Ordinary and Partial Differential Equations, Philadelphia: SIAM, 2007.

[8] L. Lundgren, "Efficient numerical methods," Uppsala Universitet, 2018.

[9] A. I. Delis and T. Katsaounis, "Relaxation schemes for the shallow water equations," *Int. J. Numer. Meth. Fluids, John Wiley & Sons,* vol. 41, p. 695–719, 2003.

[10] O. Delestre, C. Lucas, P.-A. Ksinant, F. Darboux, C. Laguerre, T. Vo, F. James and S. Cordier, "SWASHES: a compilation of Shallow Water Analytic Solutions for Hydraulic and Environmental Studies (Complimentary results version, 2018)," *Int. Journal for Numerical Methods in Fluids,* vol. 72, no. 3, pp. 269-402, May 2013.

[11] "Hydrologic Engineering Center's (CEIWR-HEC) River Analysis System (HEC-RAS) website," US Army Corps of Engineers, Hydrologic Engineering Center, 21 05 2024. [Online]. Available: https://www.hec.usace.army.mil/software/hec-ras/default.aspx.

[12] P. Koprinkova and M. Petrova, "Data-scaling problems in neural-network training," *Engineering Applications of Artificial Intelligence,* vol. 12, no. 3, pp. 281-296, 1999.

[13] P. Sharma, L. Evans, M. Tindall and P. Nithiarasu, "Stiff-PDEs and Physics-Informed Neural Networks," *Archives of Computational Methods in Engineering,* vol. 30, p. 2929–2958, 2023.

[14] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, MIT Press, 2016.

[15] C. C. Aggarwal, Linear Algebra and Optimization for Machine Learning, Springer, 2020.

[16] K. Hornik and M. Stinchcombe, "Multilayer feedforward networks are universal approximators," *Neural Networks,* vol. 2, no. 5, pp. 359-366, 1989.

[17] T. Dockhorn, "A Discussion on Solving Partial Differential Equations using Neural Networks," 15 Apr 2019. [Online]. Available: arXiv:1904.07200v1. [Accessed 2023].

[18] H. Jung, J. Gupta, B. Jayaprakash, M. Eagon, H. P. Selvam, C. Molnar, W. Northrop and S. Shekhar, "A Survey on Solving and Discovering Differential Equations Using Deep Neural Networks," *J. ACM, Article 111,* vol. 37, no. 4, pp. 1-29, August 2023.

[19] M. Raissi, P. Perdikaris and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics,* vol. 378, pp. 686-707, 2019.

[20] B. M. Wilamowski, "Understanding of Neural Networks," in *The Industrial Electronics Handbook - Intelligent Systems,* CRC Press, 2018, pp. 5-1:5-11.

[21] S. Kollmannsberger, D. D'Angella, M. Jokeit and L. Herrmann, Deep learning in Computational Mechanics, an introductory course, Springer, 2022.

[22] C. F. Higham and D. J. Higham, "Deep Learning: An Introduction for Applied Mathematicians," *SIAM Review,* vol. 61, no. 4, pp. 860-891, 2018.

[23] B. M. Wilamowski, "Neural Networks Learning," in *The Industrial Electronics Handbook - Intelligent Systems*, CRC Press, 2018, pp. 11-1:11-18.

[24] G. Strang, Linear Algebra and Learning From Data, Wellesley: Wellesley-Cambridge Press, 2019.

[25] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference on Learning Representations*, 2014.

[26] "TensorFlow Documentations," 2024. [Online]. Available: https://tensorflow.org.

[27] A. Paszke and e. al., "PyTorch: an imperative style, high-performance deep learning library," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems, pp 8026 - 8037*, 2019.

[28] A. G. Baydin, B. A. Pearlmutter, A. A. Radul and J. M. Siskind, "Automatic Differentiation in Machine Learning: a Survey," *Journal of Machine Learning Research,* vol. 153, no. 18, pp. 1-43, 2018.

[29] R. G. Nascimento, K. Fricke and A. C. Viana, "A tutorial on solving ordinary differential equations using Python and hybrid physics-informed neural networks," *Engineering Applications of Artificial Intelligence*, vol. 96, no. 103996, pp. 1-11, 2020.

[30] D. Katsikis, A. D. Muradova and G. E. Stavroulakis, "A Gentle Introduction to Physics-Informed Neural Networks, with Applications in Static Rod and Beam Problems," *Journal of Advances in Applied & Computational Mathematics,* vol. 9, pp. 103-128, 2022.

[31] L. Lu, P. Jin, G. Pang, Z. Zhang and G. E. Karniadakis, "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators," *Nature - Machine Intelligence*, vol. 3, pp. 218-229, March 2021.

[32] L. Lu, R. Pestourie, W. Yao, Z. Wang, F. Verdugo and S. G. Johnson, "Physics-Informed Neural Networks with Hard Constraints for Inverse Design," *SIAM Journal for Industrial and Applied Mathematics*, vol. 43, no. 6, pp. B1105-B1132, 2021.

[33] S. Alkhadhr and M. Almekkawy, "Wave Equation Modeling via Physics-Informed Neural Networks: Models of Soft and Hard Constraints for Initial and Boundary Conditions," *Sensors,* vol. 23, no. 2792, pp. 1-20, 2023.

[34] S. Cuomo, V. Schiamo Di Cola, F. Giampaolo, G. Rozza, M. Raissi and F. Piccialli, "Scientific Machine Learning through Physics-Informed Neural Networks: Where we are and What's next," *Journal of Scientific Computing,* vol. 92, no. 88, pp. 1-62, 2022.

[35] L. Lu, X. Meng, Z. Mao and G. E. Karniadakis, "DeepXDE: A deep learning library for solving differential equations," *SIAM Review,* vol. 63, no. 1, pp. 208-228, 2021.

[36] W. Wu, M. Daneker, M. A. Jolley, K. T. Turner and L. Lu, "Effective data sampling strategies and boundary condition constraints of physics-informed neural networks for identifying material properties in solid mechanics," *Applied Mathematics and Mechanics (English edition),* vol. 44, no. 7, pp. 1039-1068, 2023.

[37] Y. Shin, J. Darbon and G. E. Karniadakis, "On the Convergence of Physics Informed Neural Networks for Linear Second-Order Elliptic and Parabolic Type PDEs," *Commun. Comput. Phys.,* vol. 28, pp. 2042-2074, 2020.

[38] S. Markidis, "The Old and the New: Can Physics-Informed Deep-Learning Replace Traditional Linear Solvers?," *Frontiers in Big Data,* vol. 4, no. 669097, pp. 1-15, 2021.

[39] R. Pellegrin, B. Bullwinkel, M. Mattheakis and P. Protopappas, "Transfer Learning with Physics-Informed Neural Networks for Efficient Simulation of Branched Flows," in *36th Conference on Neural Information Processing Systems (NeurIPS 2022),* 2022.

[40] S. Desai, M. Mattheakis, H. Joy, P. Protopappas and S. Roberts, "One-Shot Transfer Learning of Physics-Informed Neural Networks," in *2nd AI4Science Workshop at the 39th Int. Conf. on Machine Learning (ICML2022),* 2022.

[41] V. Schafer, "Master Thesis, Generalization of PINNs for Various Boundary and Initial Conditions," University of Kaisserslautern, Faculty of Mathematics, Kaisserslautern, 2022.

[42] Y. Nakamura, S. Shiratori, H. Nagano and K. Shimano, "Physics-Informed Neural Networks with Variable Initial Conditions," in *Proceedings of the 7th World Congress on Mechanical, Chemical and Material Engineering (MCM'21),* 2021.

[43] S. Monaco and D. Apiletti, "Training physics-informed neural networks: One learning to rule them all?," *Results In Engineering, Science Direct, Elsevier,* vol. 18, no. 101023, pp. 1-9, 2023.

[44] M. M. Alqezweeni, V. I. Gorbachenko and Z. A. Karmanova, "Physics-Informed Neural Networks and their Implementation in MATLAB," in *8th Int. Conf. on Contemporary Information Technology and Mathematics (ICCITM2022),* Mosul University, Mosul-Iraq, 2022.

[45] A. Bihlo and R. O. Popovych, "Physics-informed neural networks for the shallow-water equations on the sphere," *Journal of Computational Physics,* vol. 456, no. 111024, pp. 1-18, 2022.

[46] L. F. Nazari, E. Camponogara and L. O. Seman, "Physics-Informed Neural Networks for Modeling Water Flows in a River Channel," *IEEE Transactions on Artificial Intelligence*, vol. 5, no. 3, pp. 1001-1015, 2024.

[47] X. Qi, A. M. de Almeida and S. Maldonado, "Physics-informed neural networks for solving flow problems modeled by the 2D Shallow Water Equations without labeled data," *Journal of Hydrology*, vol. 636, no. 131263, pp. 1-17, 2024.

[48] R. Leiteritz, M. Hurler and D. Pfluger, "Learning Free-Surface Flow with Physics-Informed Neural Networks," in *20th IEEE Int. Conf. on Machine Learning and Applications*, 2021.

[49] H. Cho, "Physics Informed Neural Networks for Solving the Shallow Water Equations on Grids of Arbitrary Geometry," in *SoutheastCon 2024, IEEE,* 2024.

[50] "Google Colab Documentation," 2024. [Online]. Available: https://colab.research.google.com.

[51] D. Hunter, H. Yu, M. S. Pukish, J. Kolbusz and B. M. Wilamowski, "Selection of Proper Neural Network Sizes and Architectures - A Comparative Study," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 1-13, 2012.

[52] C. Wu, M. Zhu, Q. Tan, Y. Kartha and L. Lu, "A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks," *Computer Methods in Applied Mechanics and Engineering*, vol. 403, no. 115671, pp. 1-23, 2023.

[53] M. A. Nabian, R. J. Gladstone and H. Meidani, "Efficient training of physics-informed neural networks via importance sampling," *Computer-Aided Civil and Infrastructure Engineering*, vol. 36, no. 8, pp. 962-977, 2021.

[54] C. Lucas, "Documentation of SWASHES v1.04.00," University Orleans France, 2022.

APPENDIX A: SAMPLING SEQUENCES

Appendix A - I. Low-Discrepancy sequences

A sequence (s_1, s_2, s_3, ...) of real numbers is said to be *equidistributed* on a non-degenerate interval $[a, b]$, if for every subinterval $[c, d]$ of $[a, b]$ we have:

$$\lim_{n\to\infty} \frac{|\{s_1, \ldots, s_n\} \cap [c, d]|}{n} = \frac{d-c}{b-a}$$

We define the **discrepancy** D_N for a sequence (s_1, s_2, s_3, ...) with respect to the interval $[a, b]$ as

$$D_N = \sup_{a\le c\le d\le b} \left| \frac{|\{s_1, \ldots, s_n\} \cap [c, d]|}{n} - \frac{d-c}{b-a} \right|$$

A sequence is thus equidistributed if the discrepancy D_N tends to zero as N tends to infinity.

Appendix A - II. Latin Hypercube Sampling (LHS)

Latin Hypercube Sampling (LHS) is a statistical method used for generating a sample of plausible collections of parameter values from a multidimensional distribution. It is a type of stratified sampling that ensures that the entire range of possible values for each parameter is explored more systematically and thoroughly than simple random sampling.

Key Concepts

Stratified Sampling: LHS divides the range of each parameter into intervals of equal probability. It then samples one value from each interval. This ensures that the sample is spread evenly across the entire range of each parameter, reducing the likelihood of clustering in certain regions.

High-Dimensional Sampling: LHS is particularly useful when sampling in high-dimensional spaces. In such cases, simple random sampling may require a very large number of samples to cover the space adequately, whereas LHS can achieve better coverage with fewer samples.

How LHS Works

1. Divide the Range: The range of each input parameter is divided into nnn equally probable intervals, where nnn is the number of samples desired.
2. Sample Within Intervals: One value is randomly selected from each interval for each parameter.
3. Permutation: The selected values for each parameter are then paired randomly with the other parameters' sampled values, ensuring that each combination is unique.

Comparison with Simple Random Sampling

Coverage: In random sampling, there is a possibility of some regions of the parameter space being oversampled while others are under-sampled. LHS, on the other hand, ensures that the samples are more evenly distributed across the entire space.

Efficiency: LHS can achieve the same level of accuracy with fewer samples compared to random sampling, making it more computationally efficient, especially in high-dimensional spaces.

Example of LHS
Suppose we want to sample from two variables X_1 and X_2, each ranging from 0 to 1. If we want to generate 5 samples:
Divide the range: Each variable's range [0, 1] is divided into 5 intervals: [0,0.2], [0.2,0.4], [0.4,0.6], [0.6,0.8], [0.8,1].
Random Sampling: For each variable, a random value is selected from each interval.
Combine the Values: The sampled values of X_1 and X_2 are combined to form 5 pairs. The pairs are formed such that each pair is unique, ensuring a more comprehensive coverage of the input space.

Appendix A - III. Halton sampling

Halton sampling is a method used for generating sequences of numbers that are more uniformly distributed across a multi-dimensional space than those generated by purely random sampling. This technique is particularly useful in numerical integration, computer graphics, and optimization, where it is essential to cover a space evenly and efficiently.
Halton sampling is based on low-discrepancy sequences, also known as quasi-random sequences. These sequences are designed to fill the space more uniformly compared to random sampling, thus providing better convergence properties in many applications. The Halton sequence is one of the most commonly used low-discrepancy sequences. It is generated using a process based on the radical inverse function with respect to a given base, typically a sequence of prime numbers for multi-dimensional spaces. In Halton sampling, each dimension of the sample space is associated with a different prime number base. This ensures that the points in the sequence are spread out more evenly across the entire space.
How Halton Sampling Works
Generating a Halton Sequence
Base Representation: For each dimension i in the d-dimensional space, choose a prime number p_i as the base. For instance, for a 2-dimensional space, we may use base 2 for the first dimension and base 3 for the second dimension.
Radical Inverse Function: The radical inverse function $\varphi_p(n)$ for a number n in base p, is calculated by reversing the digits of n when expressed in base p, and then converting the result into a fraction. Mathematically, for a number n in base p:

$$\varphi_p(n) = \sum_{k=0}^{m-1} a_k p^{-(k+1)}$$

where $n = a_m p^m + a_{m-1} p^{m-1} + \cdots + a_0 p^0$ is the base-p representation of n.
Constructing the Sequence: For each dimension i, the n-th point in the sequence, is given by the radical inverse function with respect to the base p_i.
The n-th point in the d-dimensional Halton sequence is:

$$Halton_p(n) = (\varphi_{p_1}(n), \varphi_{p_2}(n), \dots, \varphi_{p_d}(n))$$

This process generates points in a space that are distributed more uniformly than simple random samples.
Example of Halton Sampling
Suppose we want to generate a 2-dimensional Halton sequence:
Choose Bases: Use base 2 for the first dimension and base 3 for the second dimension.
Calculate Points:
For n=1:
$\varphi_2(1)$ in base 2 is 0.1, which is 0.5.
$\varphi_3(1)$ in base 3 is 0.1, which is 0.333.

The point is (0.5,0.333).
For n=2:
$\varphi_2(2)$ in base 2 is 0.01 which is 0.25.
$\varphi_3(2)$ in base 3 is 0.2 which is 0.666.
The point is (0.25,0.666).
This process continues to generate a sequence of points in the 2-dimensional space.

Appendix A - IV. Hammersley sampling

Hammersley sampling is a technique used to generate low-discrepancy sequences for quasi-random sampling in multidimensional spaces. Hammersley sampling aims to distribute sample points more uniformly across the space than purely random sampling, which is particularly useful in numerical integration, optimization, and computer graphics. The Hammersley sequence is a specific type of low-discrepancy sequence. It is similar to the Halton sequence but typically used when the number of points to be sampled is known in advance. The key difference is that the Hammersley sequence fixes the first coordinate of the sequence to provide even better uniformity.
How Hammersley Sampling Works. Hammersley sampling generates a sequence of points in a multidimensional space by using a combination of radical inverse functions and a regular sequence for one of the dimensions.
Generating a Hammersley Sequence: Suppose we want to generate N points in a d-dimensional space.
Base Representation for Radical Inverse: For each dimension i (except the first one), choose a prime number p_i as the base. For instance, for a 2-dimensional space, we could use base 2 for the second dimension.
Radical Inverse Function: For each dimension i, except the first, the radical inverse function $\varphi_p(n)$ is computed based on the chosen base. This is the same process as used in Halton sequences.
For the first dimension, the points are simply evenly spaced fractions of N, such as n/N for n=0,1,…,N−1.
Constructing the Sequence: The Hammersley sequence in d-dimensions is constructed as follows:

$$Hammersley_p(n) = (\frac{n}{N}, \varphi_{p_1}(n), \varphi_{p_2}(n), \dots, \varphi_{p_{d-1}}(n))$$

Here, $\frac{n}{N}$ represents the first coordinate, and $\varphi_{p_i}(n)$ represents the coordinates for the other dimensions using the radical inverse function.
Example of Hammersley Sampling. Consider generating 4 points in a 2-dimensional space:
Dimension 1 (First coordinate): Use $\frac{n}{N}$, where N=4.

- For n=0: 0/4=0
- For n=1: 1/4=0.25
- For n=2: 2/4=0.5
- For n=3: 3/4=0.75

Dimension 2 (Second coordinate): Use the radical inverse function in base 2.

- For n=0: $\varphi_2(0)$=0
- For n=1: $\varphi_2(1)$=0.5
- For n=2: $\varphi_2(2)$=0.25
- For n=3: $\varphi_2(3)$=0.75

The 4 Hammersley sequence points in 2D space would be:
(0,0), (0.25,0.5), (0.5,0.25), (0.75,0.75)

Appendix A - V. Sobol sampling

Sobol sampling refers to a method for generating low-discrepancy sequences known as Sobol sequences. These sequences are used in quasi-random sampling, which aims to cover a multi-dimensional space more uniformly than purely random or pseudo-random sequences. Sobol sequences are particularly effective in high-dimensional spaces and are widely used in numerical integration (e.g., quasi-Monte Carlo methods), optimization, and other applications requiring efficient sampling of parameter spaces. The Sobol sequence is a type of low-discrepancy sequence that is generated using a recursive algorithm based on direction numbers. It is one of the most widely used sequences for quasi-random sampling, particularly in high-dimensional spaces. Sobol sequences are generated using a method involving binary representation and bitwise operations. The sequences are constructed to ensure that the points are uniformly distributed in all dimensions.

Steps to Generate a Sobol Sequence:

Initialization: Choose the number of dimensions d for the space in which we want to generate points. For each dimension, certain parameters, such as primitive polynomials and direction numbers, are predefined or computed.

Direction Numbers: For each dimension, a set of direction numbers $v_{i,j}$ is used to generate the sequence. These numbers are crucial for ensuring the low discrepancy of the sequence and are derived from the binary representation of the points.

Recursive Construction: The n-th point in the Sobol sequence is constructed recursively using the direction numbers and the points generated so far. The sequence is designed to fill the space uniformly, ensuring that the points in the sequence avoid clustering.

Binary Representation: The coordinates of the Sobol sequence are generated by considering the binary digits of the index n. This ensures that the points are spread uniformly across the space, even as the sequence length increases.

Multi-Dimensional Sequence: In d-dimensional space, the sequence $\left(x_1^{(n)}, x_2^{(n)}, \dots, x_d^{(n)}\right)$ is generated such that each $x_i^{(n)}$ is calculated independently using the corresponding direction numbers for that dimension.

Example of Sobol Sampling. Consider generating a 2-dimensional Sobol sequence:

Initialization: For 2 dimensions, predefined direction numbers are used for each dimension. Typically, the first dimension uses direction numbers derived from the binary representation of integers, and the second dimension might use a different set of direction numbers.

Recursive Calculation: The first few points in the sequence might look like: $\left(x_1^{(1)}, x_2^{(1)}\right) = (0.5, 0.5)$, $\left(x_1^{(2)}, x_2^{(2)}\right) = (0.75, 0.25)$, $\left(x_1^{(3)}, x_2^{(3)}\right) = (0.25, 0.75)$, $\left(x_1^{(4)}, x_2^{(4)}\right) = (0.375, 0.375)$.

The points continue to fill the space in a pattern that is more uniform than random sampling.

APPENDIX B: DEEPXDE APPLICATION LIBRARY

Lu et al. developed a software library called DeepXDE which enables the user to solve forward and inverse PDEs via PINNs. Additionally, it is able to solve other kinds of problems such as forward and inverse integro-differential equations. The library was designed to make the user code stay compact and manageable, resembling closely the mathematical formulation. Thus, it simplifies the process of analyzing scientific problems when using machine learning methods. DeepXDE supports the Python libraries Tensorflow 1.x, 2.x and PyTorch.

A useful feature of the library is that it supports complex geometry domains using the technique of Constructive Solid Geometry (CSG). Although DeepXDE already supports basic geometries (rectangle, circle, etc.), the library allows the user to define more complex geometries. CSG supports two- and three-dimensional domains.

One major advantage is that the user does not have to provide training data. The user defines the PDE and BC/IC as functions represented in analytical form. One can either specify the locations of the points or only set the number of points and then DeepXDE will sample the required number randomly on a grid covering the specified domain. Another advantage is that DeepXDE increases the training efficiency by employing intelligent sampling strategies, like the Residual-based Adaptive Refinement (RAR) method. In some cases where functions contain areas with steep gradients, it is useful to choose more training points in these areas. However, in many applications the shape of the solution is unknown in advance. Thus, the design of the distribution of considered training points poses a challenge. To this end, RAR is a method which improves the distribution of training points during training process by adding more points in the locations where the PDE residual is large.

Further, the user has the ability to add so-called callback functions. This enables the user to monitor the training process and to make modifications in real time, e.g. change the learning rate, if necessary. Possible callback functions can for example save the model after certain epochs, calculate the first derivative of the outputs with respect to the inputs, or monitor a movie of the spectrum of the function's Fourier transform. It is convenient to define customized callback functions which are called at specified stages of the training process.

An overview of the usage of DeepXDE for solving PDEs is depicted in Figure B. 1 [35]. The corresponding procedure is summarized as follows.

1. Specify the computational domain using the geometry module.

2. Specify the PDE using the grammar of TensorFlow.

3. Specify the boundary and initial conditions.

4. Combine geometry, PDE and BCs/ICs together into data:PDE or data:TimePDE for time-independent or time-dependent problems, respectively. Specify the training data by either setting the specific point locations or only set the number of points.

5. Construct a NN using the maps module.

6. Define a model by combining the PDE problem in 4 and the NN in 5.

7. Call model:compile to set the optimization hyperparameters such as optimizer and learning rate. The weights w_f , w_b can be set here by loss_weights.

8. Call model:train to train the network from random initialization or a pretrained model using the argument model_restore_path. It is extremely flexible to monitor and modify the training behavior using callbacks.

9. Call model:predict to predict the PDE solution at different locations.

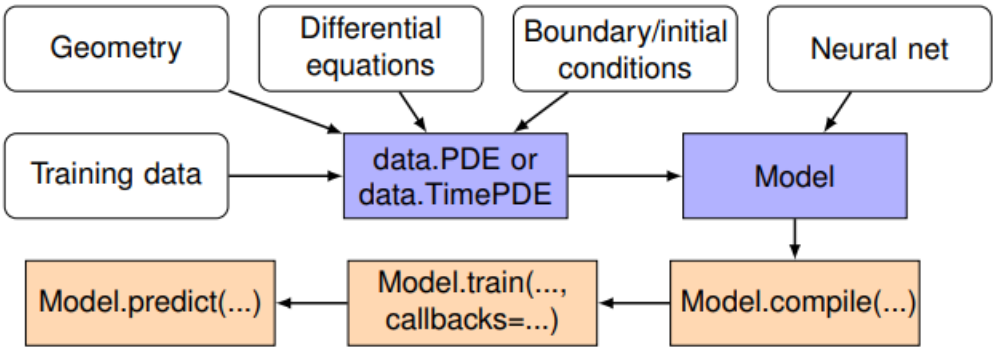


Figure B. 1: Simplified flow chart of DeepXDE library.