POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Artificial Neural Neworks for the approximate solution of Partial Differential Equations

Tesi di Laurea Magistrale in
Mathematical Engineering - Ingegneria Matematica

Author: **Beatrice Crippa**

Student ID: 921284
Advisor: Prof.ssa Paola Francesca Antonietti
Academic Year: 2020-21

Abstract

Many physical, financial and social phenomena can be modeled by means of Partial Differential Equations, that may however be very difficult to be solved analytically. For this reason their numerical solution is a central topic in the applied mathematics field, mostly focused on Galerkin-type praradigms. Such methods require the construction of a mesh discretization of spatial domains, that can be very complicated, and heavy operations to be performed at every node, and then to build a suitable discrete (i.e finite dimensional) approximation space on top of that. When problem dimensionality increases, the number of points and consequently the computational cost arise, incurring in the so-called curse of dimensionality. Machine learning provides powerful and innovative tools that has proved to be able to overcome this issue and work with huge quantity of multidimensional data. This thesis presents therefore a fully data-driven approach to the approximate solution of Partial Differential Equations, based on a coupling of two artificial neural networks that predict the valuation of the solution on coordinate points given as input. One structure works on the boundary and the other in the interior of the domain, in order to exploit all the information about the physics of the problem given by the equation and the (boundary, initial) data.
The absence of particular theoretical background in the subject does not allow a rigorous proof of convergence results and error bounds. On the other hand, the method applied to a large class of problems appears to be rapidly convergent, even if to values of the error of order no lower than 10^{-3}. Moreover, the dimensionality increase slows the convergence down but does not produce relevant obstacles in the training phase.
Fast convergence gives a good black-box method, that cannot however substitute the traditional approaches since it shows lacks in recognizing particular features of the solution (e.g. irregularities, boundary/interior layers) even when the number of hidden layers increases.

Keywords: Partial Differential Equations, Machine Learning, Artificial Neural Networks, Data-driven approximation schemes.

Abstract in lingua italiana

Molti fenomeni fisici, finanziari e sociali possono essere modellati per mezzo di equazioni a derivate parziali, che tuttavia sono spesso difficili da risolvere in modo analitico. Per questa ragione la loro soluzione numerica è un argomento centrale nel campo della matematica applicata, soprattutto focalizzata su paradigmi di tipo Galerkin. Tali metodi richiedono la costruzione di mesh per la discretizzazione di domini spaziali, che possono essere molto complicati, e l'esecuzione di pesanti operazioni ad ogni nodo. Quando la dimensionalità dei problemi aumenta, il numero di punti e di conseguenza il costo computazionale crescono, incorrendo nella cosiddetta curse of dimensionality. Il machine learning fornisce strumenti potenti ed innovativi che si sono dimostrati in grado di sorpassare questo problema e lavorare con grandi quantità di dati multidimensionali. Questa tesi presenta quindi un approccio completamente data-driven alla risoluzione approssimata delle Equazioni a Derivate Parziali, basato su un accoppiamento di due reti neurali artificiali che predicono la valutazione della soluzione in coordinate di punti date in input. Una struttura lavora sul bordo e l'altra all'interno del dominio, con lo scopo di sfruttare tutte le informazioni riguardanti la fisica del problema fornite dall'equazione e dai dati (al contorno, iniziali).
L'assenza di un background teorico approfondito in materia non permette una dimostrazione rigorosa di risultati di convergenza o controllo dell'errore. D'altra parte, il metodo applicato a un'ampia classe di problemi sembra essere rapidamente convergente, anche se verso valori dell'errore di ordine non inferiore a 10^{-3}. Inoltre, l'aumento della dimensionalità rallenta la convergenza ma non produce ostacoli rilevanti nella fase di training.
La rapida convergenza fornisce un buon metodo black-box, che non può tuttavia sostituirsi agli approcci tradizionali poiché mostra falle nel riconoscimento di caratteristiche particolari della soluzione (come irregolarità, strati al bordo/interni) anche quando il numero di strati nascosti aumenta.

Parole chiave: Equazioni a Derivate Parziali, Machine Learning, Reti neurali artificiali, Schemi di approssimazione data-driven.

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 An overview of Partial Differential Equations **7**
1.1 Elliptic equations . . . 8
1.1.1 Variational formulation . . . 9
1.2 Parabolic equations . . . 11
1.2.1 The abstract parabolic problem . . . 12
1.3 Hyperbolic equations . . . 14
1.4 Galerkin discretization . . . 15
1.4.1 The Finite Element method . . . 15

2 An overview of Artificial Neural Networks **19**
2.1 Activation function . . . 21
2.2 Backpropagation . . . 24
2.3 Gradient descent optimization algorithm . . . 28
2.3.1 Adam optimizer . . . 31
2.4 Universal approximation property . . . 33

3 Numerical solution of PDEs via ANNs **37**
3.1 Network structure . . . 41
3.2 Training algorithm . . . 44
3.3 Error metrics . . . 46
3.4 Extensions . . . 46
3.4.1 Steady advection-diffusion equation . . . 47

3.4.2 Evolutionary problems 48

4 Numerical results: set up of the test cases **51**

4.1 Elliptic problems 52

4.1.1 Poisson problem 52

4.1.2 Steady state advection-diffusion problem 53

4.2 Parabolic problems 54

4.2.1 Heat equation 55

4.2.2 Time-dependent advection-diffusion equation 56

4.3 Hyperbolic problems 56

5 Numerical results: elliptic problems **59**

5.1 Poisson-Dirichlet problem 59

5.1.1 TC.E1: Smooth solution 60

5.1.2 TC.E2: Solution with a peak 62

5.1.3 TC.E3: Solution with low regularity 66

5.1.4 Tuning of the hyperparameters 68

5.1.5 Extension to N-dimensional elliptic PDEs 77

5.2 Advection-diffusion problem 79

5.2.1 Solution with the tuned hyperparameters 83

6 Numerical results: evolutionary problems **87**

6.1 Parabolic problems 87

6.1.1 TC.P1: smooth initial condition 87

6.1.2 TC.P2: initial condition with a peak 89

6.1.3 TC.P3: Solution with low regularity 91

6.1.4 Time-dependent advection-diffusion equation 92

6.2 Hyperbolic problems 95

6.2.1 TC.H1: smooth initial condition 95

6.2.2 TC.H2: initial condition with a peak 96

7 Conclusions and future developments **99**

Bibliography **101**

A Python code **111**

List of Figures **117**

List of Tables **121**

Acknowledgements **123**

Introduction

Partial differential equations (PDEs) are one of the main mathematical tools for the description of physical phenomena, whose main application fields consist of quantum mechanics (Schrödinger equation), relativity, electromagnetism (Maxwell equations), fluid mechanics (Navier-Stokes) and thermodynamics (heat equation). PDEs are also largely adopted e.g., for models in finance (Black-Scholes equation for pricing), chemistry, neuroscience and many other fields. Although the theoretical analysis of PDEs is huge and extensive, their solution is often very difficult in practice. As a consequence, many numerical schemes were developed for their approximation, such as Galerkin Finite Elements [17], Spectral Elements [52], Finite Volumes [53], Finite Differences [54]. Such schemes are mostly based on the definition of a suitable discrete space built on top of a suitable mesh, that is a very expensive computational task [4], especially when the domains are complicated surfaces or volumes, that may also vary over time. Moreover, grid-based methods suffer from the curse of dimensionality, i.e. the computational cost grows exponentially with the domain dimension. For example, a grid made of N points per dimension in a domain of $\mathbb{R}^m$ is made of N^m total nodes, in correspondence of which shape functions must be defined and numerical integrals computed. Many modifications to these methods were proposed to overcome the above-mentioned issues, like least squares finite elements [13] or even mesh-free Galerkin [29] methods.

Recently a new class of methods was developed for the numerical solution of differential equations, thanks to their proven success in dealing with high-dimensional datasets [12, 41, 48]. Artificial neural networks (ANNs) were introduced in 1943 by McCulloch and Pitts [63], and were mainly applied to classification [72] and pattern recognition [3, 95] tasks. ANNs are learning structures that mimic the information processing of synaptic interactions in the human brain, in order to transform data inputs into numeric outputs. They are made of nodes that represent neurons, connected to each-other and exchanging information. These units are no more than very simple parametric functions of the input, whose specific parameters are automatically set during the learning process by optimization of a loss functional, often expressed as sum of square errors with respect to the target output. The ability of this machine learning structure of approximating a large set of

functions was proved by Cybenko [24], Hornik et al. [39] [38] and Gallant and White [33] in the early '90s, leading to applications in many fields, including the approximate solution of PDEs. Many approaches were introduced for the direct solution of the problems, based on architectures taking as input data given at coordinate points and learning by minimization of the approximation error given as the difference between the right-hand-side of the PDE and the equation evaluated at the input points [7, 19, 27, 87]. The initial conditions in time-dependent problems are usually approximately satisfied thanks to the addition of an appropriate penalization term in the loss functional, while the boundary constraint is often treated independently by specifying a lifting operator that makes the output of the neural network exactly satisfy it [20, 49, 50, 58]. Some specific architectures based on the Finite Element method were also introduced, for both the approximate solution of a PDE on a grid [9, 78], the creation of adaptive meshes, exploiting their ability in pattern recognition, [44, 59], or even the automatic tuning of some hyperparameters in the traditional Galerkin-type schemes [26, 84]. Another powerful application of deep learning to Partial Differential Equations is the data-driven model discovery, where the form of the PDE is determined by suitable observations [55, 76, 77]. Although the theory related to ANNs is relatively poor, some convergence results concerning these methods were proved, in addition to more general approximation properties [12].

ANN-based methods are also able to deal with huge amount of data, just as with high dimensional problems without incurring in the curse of dimensionality, and more flexible than the traditional schemes because mesh-free. On the other hand, some downsides consist in a scarse theoretical background that does not allow to prove convergence results yet and the disregard of physical information, that is not taken into account by these totally data-driven models other than weakly in the loss functional.

In this thesis I consider a ANN-based method for the approximate solution of PDEs based on the work of Xu et al. [91] and Karniadakis, Raissi et al. [77]. This exploits the idea of numerically solve the equation as an unconstrained problem with an Artificial Neural Network taking as input random points of the domain and then adjust the output by means of a lifting operator aimed at satisfying the boundary condition, that is not fitted exactly in this case, unlike in [49], [20] and [58], but only approximately. This choice was made because of the high computational cost required by the definition of such operator when the condition is not of fully Dirichlet-type [50]. In particular, the estimation is performed by an independent neural network, processing points on the boundary and setting the nodes parameters by minimizing the sum of square errors related to the constraint. The two structures are then coupled by expressing the final approximation result as a sum of their outputs, in which the result of the main PDE network is multiplied by a

function whose value is 0 on the boundary. The test cases analyzed in the last chapters of this thesis only involve Dirichlet boundary conditions, but the extension to any other type of boundary conditions is straightforward. As we will see, the boundary network is a very simple structure, that also works on a domain whose dimensionality is 1 less than the one of the PDE, and therefore the error produced is much lower and convergence to the fixed threshold is achieved in very few iterations. This implies that the substitution of a lifting operator that fits the boundary condition exactly with an approximate one does not produce large variations in the overall error. The introduction of Neumann or Robin-type boundary conditions should not affect much the method performance because no high-order derivatives computations are involved, and therefore the learning process stability is not affected, just like the complexity of the boundary network.

In this thesis I have extended the idea proposed by Xu et al. [91] for elliptic PDEs to parabolic and hyperbolic PDEs. The time component was just inserted as an additional dimension of the input variables and no discretization was introduced. I have studied parabolic and hyperbolic problems, including the Burgers equation and other conservation laws.

The aim of my work was to provide an intuitive, fast but computationally expensive testing of the performance of ANN-based approximate solvers for a wide class of PDEs in terms of approximation error, stability and computational time. As we will see in Chapters 5 and 6, the approximation error tends to reach similar values for almost all the examples and many cases show a convergent trend. I have also applied the method to high dimensional problems and the performance seems not to suffer from the curse of dimensionality, even though such an extension is not fully exploited in this work. Further developments can focus on a more systematic study of the networks architecture, that is not fully automatized in this thesis. However, this issue can be overcome by further research and application of already known autotuning methods.

The thesis is organized as follows:

Chapter 1

Chapter 1 recalls the main ingredients on Partial Differential Equations, starting from the general formulations and some theoretical results. The partition into elliptic, parabolic and hyperbolic problems is introduced, in order to explain their roles and applications in the description of physical problems.
The last section of this chapter introduces the most common numerical scheme for the approximate solution of PDEs, i.e. the Galerkin method. The most relevant results

concerning stability and error bounds are also presented.

Chapter 2

Chapter 2 presents an overview on Artificial Neural Networks, their functioning and main applications.
First the neuron-like structure is described, with some pros and cons related to this approach, followed by specific insights into the selection of the activation functions and learning algorithms based on backpropagation of the error. Finally, some theoretical results concerning the application of ANNs to function approximation are presented in the final part of this chapter.

Chapter 3

Chapter 3 addresses the application of ANNs to the numerical solution of PDEs. The chapter starts with a brief introduction to the state of the art of already existing machine learning approaches, laying the foundations for my thesis, based on the work of Xu et al. [91].
The architecture consists in the coupling of two structures, made of two independent networks processing data in the form of locations from the interior and boundary of a spatial domain Ω in order to give as output the approximate solution of the PDE at those points. The measures of error related both to the specific networks and to the overall solution are also defined here.
In the first part of the chapter the method is described for its application to elliptic PDEs, while in the second part of the chapter its extension to time-dependent parabolic and hyperbolic PDEs is considered.

Chapter 4

Chapter 4 introduces the main test cases addressed in the numerical testing and constructs three examples for every class of equation of interest, having analytical solutions with different regularity.
As elliptic PDEs, the Poisson and stationary advection-diffusion equations are considered, while their evolution in time is analyzed in the parabolic example. Finally, also hyperbolic problems are introduced in the form of linear transport.

Chapter 5

Chapter 5 contains the numerical examples for elliptic-type PDEs.
The first test cases are analyzed, starting from the Poisson problem and going on to the steady advection-diffusion.
The first half of the chapter performs the experiments with neural architectures built as suggested by Xu et al. [91] and then justifies the choice of hyperparameters like the number of hidden layers, neurons and learning rate by trial and error. The resulting structure is then applied to higher dimensional extension of the studied problems and then to the advection-diffusion equation. In the last part of this chapter, the hyperparameters are retuned for the specific case where also the transport term is considered.

Chapter 6

Chapter 6 deals with the numerical testing of time-dependent extension of the problems studied in the previous chapter. In the first part, the method is applied to parabolic test cases with the corresponding previously tuned values of the hyperparameters. In the second part, we consider the extension of the proposed approach to hyperbolic conservation laws, in terms of transport equations.
The hyperparameters of the neural architectures are set according to the tuning made for the Poisson problems.

Chapter 7

Chapter 7 contains some concluding remarks on the application of the proposed method to the numerical solution of PDEs and a final comparison with Finite Element method presented in Chapter 1 are discussed in terms of approximation error, stability and computational cost.
In the end, some future developments of this approach are presented.

1 | An overview of Partial Differential Equations

A Partial Differential Equation is a relation that can be expressed as follows:

$$F(x_1, ..., x_n, u_{x_1}, ..., u_{x_n}, ..., u_{x_1,x_1}, u_{x_1,x_2}, ..., u_{x_n x_n}, u_{x_1 x_1 x_1}, ...) = 0, \tag{1.1}$$

with $u = u(x_1, ..., x_n)$. The equation **order** is given by the maximum order of derivative involved.

If F is linear with respect to u and all of its derivatives, then the equation is called **linear**, while if it is nonlinear only with respect to u it is called **semilinear** and if it is linear only with respect to the derivatives of maximum order, with coefficient only dependent on $\mathbf{x}$ and u, **quasi-linear**. Finally, if F is nonlinear with respect to the derivatives of u of maxium order, the equation is called **completely nonlinear**. The theory about linear PDEs is sufficiently consolidated, while the complexity and variety of nonlinear ones hampers unified results.

Among second order linear PDEs an additional classification can be introduced. Indeed, in this case the equation (1.1) can be expressed as follows:

$$\sum_{i,j=1}^{n} a_{ij} \frac{\partial^2 u}{\partial x_i \partial x_j} + \textit{lower order terms},$$

where $x_1, ..., x_n$ are linearly independent variables.
Then, according to the eigenvalues of the matrix $\mathbf{A} = (a_{ij})_{i,j=1}^{n}$, the equation can be elliptic, parabolic or hyperbolic, as explained in the following sections. For further details, refer to Salsa [83].

Any problem can moreover be formulated in many ways, and to each of them a notion of solution is associated:

- **Strong solution:** $u \in C^2(\Omega) \cap C(\bar{\Omega})$

Derivatives up to the second order are defined in the pointwise sense, boundary data are assumed by taking limits in a pointwise sense and f is required to be continuous;

- **Strong solution:** $u \in H^2(\Omega)$
  Distributional derivatives are defined up to the second order, the differential equation is satisfied almost everywhere with respect to the Lebesgue measure and the boundary condition is intended in the sense of traces;
- **Variational/weak solution:** $u \in H^1(\Omega)$
  This is the notion of solution we are going to develop. The equation holds in the weak sense;
- **Solution in the distributional sense:** $u \in L^1_{loc}(\Omega)$
  The equation is intended in the sense of distributions, i.e. $-\int_\Omega u\Delta v = \int_\Omega fv \ \forall v \in D(\Omega)$, and the boundary condition is satisfied in a very weak sense;
- **Solution in the viscosity sense:** both an upper semicontinuous function such that $\forall \mathbf{x}_0 \in \Omega,\ \forall \phi \in C^2(\Omega)$ such that $\phi(\mathbf{x}_0) = u(\mathbf{x}_0),\ \phi \geq u$ in a neighborhood of $\mathbf{x}_0$, we have $F(x_0, \phi(x_0), D\phi(x_0), D^2\phi(x_0)) \leq 0$ (subsolution) and a lower semicontinuous function such that $\forall \mathbf{x}_0 \in \Omega,\ \forall \phi \in C^2(\Omega)$ such that $\phi(\mathbf{x}_0) = u(\mathbf{x}_0),\ \phi \leq u$ in a neighborhood of $\mathbf{x}_0$, we have $F(x_0, \phi(x_0), D\phi(x_0), D^2\phi(x_0)) \geq 0$ (supersolution).

According to the **coherence principle**, if the data and the solution are smooth, then all the notions must coincide.

All the results stated in the following sections are proved in [83].

1.1. Elliptic equations

The first type of equations considered is typically invovled in modeling steady state of phenomena like for instance electromagnetic potentials and elastic vibrations.

Definition 1.1.1 (Elliptic equation). *Let $\Omega \subseteq \mathbb{R}^n$ be a convex compact domain, $\mathbf{A} = \mathbf{A}(\mathbf{x}) = a_{ij}(\mathbf{x})$ a square real matrix of order n, $\mathbf{b} = \mathbf{b}(\mathbf{x})$ and $\mathbf{c} = \mathbf{c}(\mathbf{x})$ vector fields in $\mathbb{R}^n$, $a = a(\mathbf{x})$ and $f = f(\mathbf{x})$ real functions. Then, the equation*

$$-\nabla \cdot (\mathbf{A}\nabla u) + \nabla \cdot (\mathbf{b}u) + \mathbf{c} \cdot \nabla u + au = f \tag{1.2}$$

*is called **elliptic** if $\mathbf{A}$ is positive definite in Ω, i.e. if the following **elliptic condition***

holds:

$$\sum_{i,j=1}^{n} a_{ij}(\mathbf{x})\xi_i\xi_j > 0, \quad \forall \mathbf{x} \in \Omega,\ \forall \xi \in \mathbb{R}^n,\ \xi \neq \mathbf{0}.$$

The equation (1.2) is expressed in **divergence form**, highlighting the structure of the left hand side, made of three terms. The first one usually models **diffusion** in non-homogeneous and/or anisotropic means: $-\nabla \cdot (\mathbf{A}(\mathbf{x})\nabla u)$, with u representing for instance the temperature or concentration of a substance. $\nabla \cdot (\mathbf{b}u) + \mathbf{c} \cdot \nabla u$ models **convection** (or transport), and in particular when $\nabla \cdot (\mathbf{b}) = 0$, it becomes $(\mathbf{b} + \mathbf{c}) \cdot \nabla u$. Finally, au is called **reaction** term and f represents the action of an **external source**, distributed over Ω.

The problem we want to solve is to find u satisfying equation (1.2) for given specific $\mathbf{A}$, $\mathbf{b}$, $\mathbf{c}$, a, and f and some boundary conditions. In this project we will only consider **Dirichlet** boundary conditions, that set the exact value of the solution $u = g_D$ on $\partial\Omega$.

1.1.1. Variational formulation

In this section we are going to derive the weak formulation of the elliptic problem with Dirichlet boundary conditions. We start from the case with homogeneous boundary conditions and then extend the result for the nonhomogeneous case.

Consider a test function $v \in C_0^1(\Omega)$, multiply both sides by v and integrate over the domain:

$$-\int_\Omega \nabla \cdot (\mathbf{A}(\mathbf{x})\nabla u - \mathbf{b}u)v + \int_\Omega \mathbf{c} \cdot \nabla uv + \int_\Omega auv = \int_\Omega fv.$$

Integrate by parts (according to the divergence theorem) on the left hand side and apply the boundary condition:

$$\int_\Omega \mathbf{A}(\mathbf{x})\nabla u \cdot \nabla v - \int_\Omega u\mathbf{b} \cdot \nabla v + \int_\Omega \mathbf{c} \cdot \nabla uv + \int_\Omega auv = \int_\Omega fv \quad \forall v \in C_0^1(\Omega). \tag{1.3}$$

Let us now introduce the Sobolev spaces $H^1(\Omega)$ and $H_0^1(\Omega)$:

Definition 1.1.2. $H^1(\Omega) := \{v : \Omega \to \mathbb{R} \quad st \quad v \in L^2(\Omega), \quad \nabla v \in [L^2(\Omega)]^2\}$.

Definition 1.1.3. $H_0^1(\Omega) := \{v \in H^1(\Omega) \quad st \quad v = 0 \ on\ \partial\Omega\}$.

Since $H_0^1(\Omega)$ is the closure of $C_0^1(\Omega)$ with respect the norm $||\nabla(\cdot)||_0$, then we can look for a solution $u \in H_0^1(\Omega)$.

Definition 1.1.4 (Variational/weak elliptic problem). *Find $u \in H_0^1(\Omega)$ such that equation* (1.3) *is satisfied* $\forall v \in H_0^1(\Omega)$.

Let $V = H_0^1(\Omega)$ and introduce an operator $F: V \longrightarrow \mathbb{R}$ such that $F(v) = \int_\Omega fv \quad \forall v \in V$ and a bilinear form $a : V \times V \to \mathbb{R}$ such that $a(w,v) = \int_\Omega \mathbf{A}(\mathbf{x})\nabla w \cdot \nabla v - \int_\Omega w\mathbf{b} \cdot \nabla v + \int_\Omega \mathbf{c} \cdot \nabla wv + \int_\Omega awv \quad \forall w, v \in V$. Then, the **abstract weak formulation** reads as:

$$\begin{cases} \text{Find} \quad u \in V \quad \text{such that} \\ a(u,v) = F(v) \quad \forall v \in V. \end{cases} \tag{1.4}$$

According to Theorem 1, the abstract weak problem (1.4) and, therefore, the elliptic Dirichlet problem, are well posed.

Theorem 1.1. *Let $f \in L^2(\Omega)$. If $\mathbf{b}$ and $\mathbf{c}$ are Lipschitz continuous on Ω and $a - \frac{1}{2}\nabla \cdot (\mathbf{c} - \mathbf{b}) \geq 0$ ae in Ω, then $\exists!$ a weak solution $u \in H_0^1(\Omega)$ to the Dirichlet problem, satisfying the following stability estimate:*

$$||\nabla u||_{L^2(\Omega)} \leq \frac{c_p}{\alpha_0}||f||_{L^2(\Omega)},$$

where α_0 is the coercivity constant of $a(w,v)$ and $c_p > 0$ is the constant of the Poincaré inequality on Ω.

If the Dirichlet boundary condition is not homogeneous, i.e. $u = g_D \quad$ on $\partial\Omega$, $g_D \in H^{\frac{1}{2}}(\Omega)$ and Ω is at least Lipschitz continuous, so that there exists an extension $\tilde{g_D}$ of g_D in $H^1(\Omega)$, then we can define $w := u - \tilde{g_D} \in H_0^1(\Omega)$, satisfying the homogeneous Dirichlet equation with the following right hand side:

$$f - \nabla \cdot (\mathbf{A}(\mathbf{x})\nabla\tilde{g_D}) + \nabla \cdot (\mathbf{b}\tilde{g_D}) + \mathbf{c}\nabla\tilde{g_D} + a\tilde{g_D}.$$

Then, $\exists! w \in H_0^1(\Omega)$ satisfying the weak-sense equation and the following stability estimate:

$$||u||_{H^1(\Omega)} \leq \tilde{c}(||f||_{L^2(\Omega)} + ||g||_{H^{\frac{1}{2}}(\Omega)}),$$

where $||g||_{H^{\frac{1}{2}}(\Omega)} := \inf\{v \in H^1(\Omega) \text{ such that } v = g \text{ on } \partial\Omega\}$. Therefore, the non-homogeneous problem is well posed and $\exists!$ solution $u \in H^1(\Omega)$ given by $u = w + \tilde{g_D}$.

If we let $\mathbf{A} = \mathbf{I}$, $\mathbf{b} = \mathbf{c} = \mathbf{0}$ and $a = 0$, we obtain the first problem of our interest:

Definition 1.1.5 (Poisson-Dirichlet problem)**.**

$$\begin{cases} -\Delta u = f & \text{in } \Omega, \\ u = g_D & \text{on } \partial\Omega. \end{cases} \tag{1.5}$$

In the weak formulation, the bilinear form is defined as follows:

$$a(w,u) = \int_\Omega \nabla w \cdot \nabla v \quad \forall w, v \in V.$$

If we let instead $\mathbf{A} = \epsilon \mathbf{I},\ \mathbf{c} \neq 0,\ \mathbf{b} = \mathbf{0}$ and $a = 0$, with $f \in L^2(\Omega),\ \epsilon \in \mathbb{R},\ 0 < \epsilon_x < \epsilon < \epsilon^*,\ ||\mathbf{c}||_{L^\infty(\Omega)} \leq c$, then we obtain the advection-diffusion problem:

Definition 1.1.6 (Stationary advection-diffusion problem)**.**

$$\begin{cases} -\Delta u + \mathbf{c} \cdot \nabla u = f & \text{in } \Omega, \\ u = g_D & \text{on } \partial\Omega. \end{cases} \tag{1.6}$$

In the weak formulation, the bilinear form is defined as follows:

$$a(w,v) = \epsilon \int_\Omega \nabla w \cdot \nabla v + \mathbf{c} \int_\Omega \nabla w v \quad \forall w, v \in V.$$

1.2. Parabolic equations

Definition 1.2.1 (Parabolic equation)**.** *Let $\Omega \subset \mathbb{R}^n$ be a limited domain and consider the spatio-temporal cylinder $Q_T = \Omega \times (0,T)$, with $T > 0$. Let $\mathbf{A} = \mathbf{A}(\mathbf{x},t)$ be a square real matrix of order n, $\mathbf{b} = \mathbf{b}(\mathbf{x},t)$ and $\mathbf{c} = \mathbf{c}(\mathbf{x},t)$ vectors in $\mathbb{R}^n$, $a = a(\mathbf{x},t)$ and $f = f(\mathbf{x},t)$ real functions. Then, the following equation in divergence form:*

$$u_t - \nabla \cdot (\mathbf{A}\nabla u - \mathbf{b}u) + c \cdot \nabla u + au = f$$

*is called **parabolic** if*

$$A(\mathbf{x},t)\xi \cdot \xi > 0 \quad \forall (\mathbf{x},t) \in Q_T,\ \forall \xi \in \mathbb{R}^n,\ \xi \neq \mathbf{0}. \tag{1.7}$$

This equation is completed by assigning boundary conditions on $\partial\Omega$, which will be considered of Dirichlet type (i.e. $u = g_D$ on $\partial\Omega \times [0,T]$) in the following chapters, and an

initial condition

$$u(\mathbf{x},0) = g(\mathbf{x}) \qquad \forall \mathbf{x} \in \Omega.$$

1.2.1. The abstract parabolic problem

The most common initial-boundary value problems can be reformulated in terms of abstact parabolic problem (APP), so in this section we will analyze its weak formulation and an existence, uniqueness and stability result.

The functional setting where the APP is defined consists of an Hilbert triplet $< V, H, V^* >$, where H and V are separable, $< \cdot, \cdot >_*$ indicates the duality between V and V^* and $|| \cdot ||_*$ the norm in V^*. The choice of V depends on the boundary conditions and usually corresponds to $V = H_0^1(\Omega)$ when they are of Dirichlet type. The domain shall be considered limited and Lipschitz. Moreover, we set the initial datum as $g \in H$ and consider the distributed source

$$f \in L^2(0,T;V^*) := \{f : [0,T] \longrightarrow V \text{ measurable such that } t \mapsto ||f(t)||_V \text{ is in } L^2(0,T)\}.$$

Finally, a bilinear form $B(w,z;t) : V \times V \longrightarrow \mathbb{R}$ is defined for almos every $t \in (0,T)$, satisfying the following hypotheses:

- Continuity: $\exists M = M(T) > 0$ such that $|B(w,z;t)| \leq M||w||_V||z||_V \quad \forall w,z \in V,$ ae in $(0,T)$,
- $V - H$ weak coercivity: $\exists \lambda, \alpha > 0$ such that $B(w,w;t) + \lambda||w||_H^2 \geq \alpha||w||_V^2 \quad \forall w \in V,$ ae in $(0,T)$,
- Measurability with respect to t: $t \mapsto B(w,z;t)$ is measurable $\forall w,z \in V$.

Definition 1.2.2 (Abstract parabolic problem). *Find $u \in H^1(0,T;V,V^*)$ such that*

$$\begin{cases} \int_0^T < u_t(s), v(s) >_* ds + \int_0^T B(u(s)v(s);s)ds = \int_0^T < f(s), v(s) >_* ds \quad \forall v \in L^2(0,T;V), \\ u(0) = g. \end{cases}$$

The following theorem assures the well posedness of the problem:

Theorem 1.2. *The APP has a unique solution $u \in V = H^1(0,T;V,V^*) := \{u : u \in L^2(0,T;V),\ u_t \in L^2(0,T;V^*)\}$. Moreover, the following **energy estimates** hold:*

$$||u(t)||_V^2,\ \alpha \int_0^t ||u(s)||_V^2 ds \leq e^{2\lambda t} \left(||g||_H^2 + \frac{1}{\alpha} \int_0^t ||f(s)||_{V^*}^2 ds \right)$$

and
$$\int_0^t ||u_t(s)||_{V^*}^2 ds \leq \left(C_0 ||g||_H^2 + C_1 \int_0^t ||f(s)||_{V^*}^2 ds \right)$$
$\forall t \in [0,T]$, *with* $C_0 = 2\alpha^{-1} M^2 e^{2\lambda t}$, $C_1 = \alpha^{-2} M^2 e^{2\lambda t} + 2$.

In the case of parabolic problems, $B(u,v;t) = \int_\Omega (\mathbf{A}(\mathbf{x},t)\nabla u \cdot \nabla v + (\mathbf{c}(\mathbf{x},t) \cdot \nabla u)v + a(\mathbf{x},t)uv)\, d\mathbf{x}$ can be proved to satisfy the conditions discussed above. Therefore, the following result holds:

Theorem 1.3. *If $f \in L^2(0,T;V^*)$ and $g \in L^2(\Omega)$, then there exists a unique weak solution to the parabolic problem. Moreover, the following estimates hold:*
$$\max_{t\in[0,T]} ||u(t)||_0^2, \int_0^T ||u||_V^2 dt \leq C \left(\int_0^T ||f(t)||_*^2 dt + ||g||_0^2 \right)$$
and
$$\int_0^T ||u(t)||_*^2 dt \leq C \left(\int_0^T ||f(t)||_*^2 dt + ||g||_0^2 \right).$$

As well as in the elliptic case, the regularity of the solution increases with the regularity of the data.

The heat equation is a particular case of parabolic problem, describing the heat propagation by conduction in an homogeneous and isotrpic means.
In the following definition, $u(\mathbf{x},t)$ describes the temperature in the space location $\mathbf{x}$ at the time instant t of a metal object occupying the volume Ω. The temperature on the boundary is controlled by a function g_D, while f represents an external heat source. The temperature at time $t = 0$ is known as well and described by a function g.

Definition 1.2.3 (Heat equation)**.**
$$\begin{cases} u_t - \Delta u = f & \text{in } \Omega \times [0,T], \\ u = g_D & \text{on } \partial\Omega \times (0,T], \\ u(\mathbf{x},0) = g(\mathbf{x}) & \forall \mathbf{x} \in \Omega \end{cases} \tag{1.8}$$

If we add to the PDE of problem (1.8) a transport term $\mathbf{c} \cdot \nabla u$, then we obtain the evolutionary counterpart of the problem (1.6) introduced in Section 1.1.1:

Definition 1.2.4 (Advection-diffusion problem)**.**

$$\begin{cases} u_t - \Delta u + \mathbf{c} \cdot \nabla u = f & \text{in } \Omega, \\ u = g_D & \text{on } \partial\Omega, \\ u(\mathbf{x}, 0) = g(\mathbf{x}), & \forall \mathbf{x} \in \Omega. \end{cases}$$

1.3. Hyperbolic equations

Second order hyperbolic equations derive from a generalization of the wave equation $u_{tt} - c^2 \Delta u = f$ to

$$u_{tt} - \nabla \cdot (\mathbf{A}(\mathbf{x}, t) \nabla u) + \mathbf{b}(\mathbf{x}, t) \cdot \nabla u + c(\mathbf{x}, t) u = f(\mathbf{x}, t),$$

under the condition (1.7).

Typical problems require both boundary conditions on the lateral domain $\partial\Omega \times (0, T]$ and initial conditions

$$u(\mathbf{x}, 0) = g(\mathbf{x}), \quad u_t(\mathbf{x}, 0) = h(\mathbf{x}) \qquad \text{in } \Omega.$$

Since the general theory is very complex, we will limit the analysis on the problems of our interest. In partcular, we will talk about **conservation laws**:

$$u_t + \nabla \cdot \mathbf{F}(u) = 0 \qquad \text{in } \Omega \times (0, T]. \tag{1.9}$$

In general, u represents the concentration of a physical quantity and F its flux function, and the associated problems are initial value ones. Equation (1.9) often appears in 1-dimensional fluid-dynamics describing the formation and propoagation of shock and rarefaction waves.

If we take as flux of u the linear function $\mathbf{F}(u) = \beta u$, the first kind of conservation law that we will consider as test case is derived:

Definition 1.3.1 (Linear advection)**.**

$$\begin{cases} u_t + \beta \cdot \nabla u = f & \text{in } \Omega \times [0, T], \\ u = g_D & \text{on } \partial\Omega \times (0, T], \\ u(\mathbf{x}, 0) = g & \text{in } \Omega. \end{cases} \tag{1.10}$$

1.4. Galerkin discretization

A traditional method for the approximate solution of a PDE problem, based on the domain discretization and numeric integration is the Galerkin method.

Consider a sequence $\{V_h\}_h$ of finite-dimensional spaces $V_h \subseteq V$ such that $\dim(V_h) = N_h < \infty$, and restrict the abstract weak problems (1.4) to each V_h:

Definition 1.4.1 (Discrete weak formulation)**.**

$$\begin{cases} \text{Find} \quad u_h \in V_h \quad \text{such} \quad \text{that} \\ a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h \end{cases} \tag{1.11}$$

Theorem 1.4. *The solution of the discrete weak problem* (1.11) *is equivalent to the solution of a linear system of the form* $A\mathbf{u_h} = \mathbf{F}$*, where* $A \in \mathbb{R}^{N_h \times N_h}$*,* $\mathbf{F} \in \mathbb{R}^{N_h}$ *and* $\mathbf{u_h} \in \mathbb{R}^{N_h}$*.*

Let $\{\phi_1, ..., \phi_{N_h}\}$ be a basis for the finite-dimensional subspace V_h. Then, the matrix A has elements $a_{ij} = a(\phi_j, \phi_i) \quad \forall i, j = 1, ..., N_h$, $\mathbf{F} = [F(\phi_1), ..., F(\phi_{N_h})]^T$ and $\mathbf{u_h}$ is the vector of the expansion coefficients of u with respect to the given basis functions.

The algebraic problem is well-posed, and so is the discrete weak problem. Moreover, a quasi-optimal error bound holds:

Theorem 1.5. *Let* $u \in V$ *be the solution of the weak problem* (1.3) *and* $u_h \in V_h$ *be the solution of the discrete problem* (1.11)*. Then,*

$$||u - u_h||_V \leq \frac{M}{\alpha} \inf_{v_h \in V_h} ||u - v_h||_V,$$

where M *is the continuity constant of* $f(\cdot)$ *and* α *is the coercivity constant of* $a(\cdot, \cdot)$*.*

1.4.1. The Finite Element method

We are now left to construct the finite-dimensional spaces V_h with the finite element method.

The discretization is based on a triangular mesh $\tau_h = \{T\}_T$ of granularity h, such that $\Omega = \cup_{T \in \tau_h} T$, under the following assumptions:

- **Shape regularity**: $\exists c > 0 \quad st \quad \forall T \in \tau_h \quad \frac{\rho_T}{h_T} \geq c$, where ρ_T is the radius of the greatest ball contained in the triangle T and $h_T = \max_{x,y \in T} |x - y|$ is called the

diameter of the element T.

- **Mesh conformity**: the intersection of two triangles of the mesh is either $\emptyset$, or a vertex or a full edge.

Definition 1.4.2 (Finite element space). *Let* $V_h := \{v_h \in \mathcal{C}^0(\bar{\Omega}) \quad st \quad v_h\big|_T \in \mathbb{P}^n(T) \quad \forall T \in \tau_h, \quad v_h = 0 \quad on \quad \partial\Omega\}$*, where* $\mathbb{P}^n(T)$ *is the space of polynomials of degree at most* n *defined on* T.

In order to be uniquely identified on the mesh, a function must be defined on the set of the **degrees of freedom**, which varies according to the mesh shape and dimensionality. In the 2D conforming triangular case, the set of degrees of freedom is given by $\{V_i\}_{i=1,...,N_h}$, where V_i, $i = 1, 2, ..., N_h$ is an interior vertex of the mesh, since on the boundary nodes the Dirichlet condition is already imposed.

Finally, consider as **shape functions** the Lagrange functions $\phi_i(V_j) = \delta_{ij}\ \forall i, j = 1, 2, ..., N_h$, forming a basis for the finite-dimensional vector space V_h.

For the numerical solution of the proposed problem with Galerkin Finite Element method the following result concerning the L^2 norm of the approximation error holds:

Theorem 1.6. *Let* Ω *be a convex polygonal domain,* $u \in V$ *the solution of the weak problem* (1.3) *and* $u_h \in V_h$ *the solution of the discrete problem* (1.11)*. If* $u \in H^{r+1}(\Omega)$ *for some integer* $r \geq 1$*, then*

$$||u - u_h||_{L^2(\Omega)} \leq c(r)h^{r+1}|u|_{H^{r+1}(\Omega)}.$$

Moreover, we know that the exact solution to the weak problem (2.3) belongs to $H^1(\Omega)$, and the following error estimate holds:

Theorem 1.7. *Let* Ω *be a convex polynomial domain,* $u \in V$ *be the solutiion to the weak problem* (1.3) *and* $u_h \in V_h$ *be the solution to the discrete problem* (1.11)*. If* $u \in H^{r+1}(\Omega)$ *for some integer* $r \geq 1$*, then*

$$||u - u_h||_{H^1(\Omega)} \leq c(r)h^r|u|_{H^{r+1}}(\Omega)$$

In the advection-diffusion case some oscillations may arise. In order to make the method more stable, an artificial viscosity parameter is added, giving rise to particular schemes,

like the **upwind FEM**, where the L^2 error is controlled by h instead of h^2.

For further results and the application of the Galerking Finite Element method, refer to [74].

2 | An overview of Artificial Neural Networks

An artificial neural network (ANN) is a computational learning system, based on the mechanisms underlying the way neurons and synapses in the human brain recognize patterns. It consists in a network of functions able to understand and translate a data input into a numeric output. The task of actually mimicking the functioning of human brain for the creation of neurocomputers able to interact with the environment has not been achieved yet, even if it is not theoretically forbidden [40]. The idea of neurocomputing was first introduced in 1943 by McCulloch and Pitts [63], who suggested that even simple neural networks could compute arithmetic and logical functions, followed by the construction of the first functioning neurocomputer by Frank Rosenblatt and others in 1958. Up to now artificial intelligence is only capable of executing some given learning tasks, mainly applied to classification [72], pattern recognition [3, 95], and prediction in many disciplines, usually providing reasonable robustness. Some examples can be found in image processing [31, 48], handwriting [22, 61] and speech recognition [37], face identification [85], medical diagnosis [1, 75, 94] and biology [35]. Neural networks have also proved success in the field of physics [56] and mathematics [68, 96].

As shown in Figure 2.1, the building blocks of the networks structure are simple nodes, grouped in layers. We will only consider **feedforward dense neural networks**, where the flux of data processing follows the scheme:

$$\text{input layer} \longrightarrow \text{hidden layers} \longrightarrow \text{output layer}.$$

The left-most layer is called **input layer**, and consists of a given data set, while the right-most one is the **output layer** and is usually a single node containing the estimation result. Between input and output there are some **hidden layers**, whose nodes, called **neurons**, perform the actual classification tasks, each one taking several inputs $\mathbf{x} = [x_1, ..., x_j, ..., x_N]^T$ and giving a monodimensional output y. Both the inputs x_j, $j = 1, ..., N^T$, and the otuput y can be either scalars or vectors.

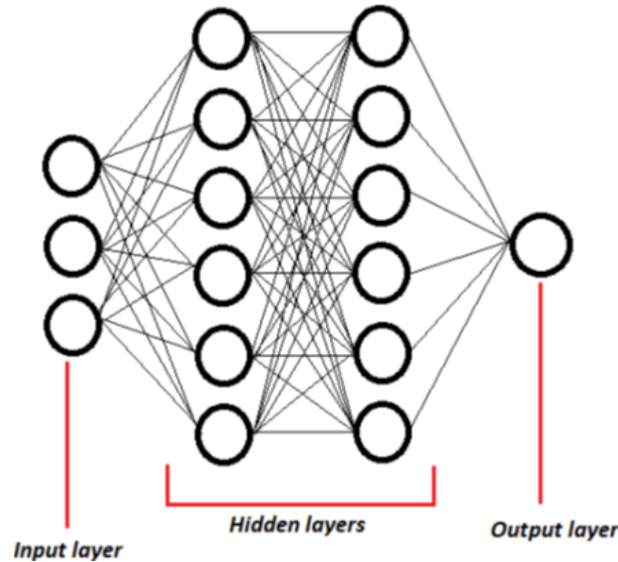


Figure 2.1: Feedforward dense neural network structure with 2 hidden layers composed by 6 neurons each.

Graphic connections between nodes belonging to different layers mean that the output of the one is taken as input from the other, forming a hierarchical structure that allows more and more complex decisions as the number of layers increase. A network with many hidden layers is called **deep neural network**, and it can be decomposed into smaller subnetworks. In feedforward neural networks no loops among layers are allowed and in dense networks each neuron takes as input the output from all the neurons in the previous layer only (see Figure 2.1).

There exist no proven "recipes" for setting the correct number of layers or neurons for specific problems, as well as many other features like the size of the training set, the weights initialization range, the stopping criterion, etc., called **hyperparameters** [88]. We will see at the end of this chapter that an ANN with a single layer can approximate continuous functions, thus it is generally sufficient for most problems, especially for classification tasks. However, if on the one hand a low number of layers helps preventing overfitting of the input data, on the other a deeper structure can learn complex relations among data in a more efficient way. The most trivial tuning technique is based on trial and error, but some more advanced heuristics were proposed, like the one based on Kolmogorov theorem for the approximation of continuous functions in $[0, 1]^n$ [36], or pruning and constructive algorithms as Optimal Brain Surgeon method [73]. Since the networks performance appears to be strongly dependent on the choice of the hyperparameters, and in order to make machine learning algorithms more autonomous, automatic tuning techniques were also introduced: some of the most promising are based on random search

[10] and Bayesian optimization [11, 64]. Finally, metalearning, i.e. learning based on prior experience on other tasks, can also improve the design of these architectures by the creation of recurrent neural networks [62].

As far as this thesis is concerned, we will tune the hyperparameters by trial and error, analyzing the effects of varying the number of hidden layers and the relative number of neurons contained in each layer, and finally adjusting the learning rate (a real parameter, whose specific definition will be introduced in Section 2.3.)

2.1. Activation function

The most simple type of neuron is the **perceptron**, introduced for the first time in 1958 by Rosenblatt [80], that transforms several input data $\mathbf{x} = [x_1, ..., x_j, ..., x_N]^T$ into a single output

$$y = \begin{cases} 0 & if\ \mathbf{w} \cdot \mathbf{x} \leq b, \\ 1 & if\ \mathbf{w} \cdot \mathbf{x} > b. \end{cases}$$

The parameters $w_j \in \mathbb{R}^N$ and $b \in \mathbb{R}$, are the **weights** and **threshold** (or **bias**), respectively, and are specific for each learning unit and updated at every iteration in order to minimize the classification error, usually in the least squares sense. In the perceptron case, the binary output is traduced in activation or non-activation of the associated neuron if the weighted sum of the inputs is sufficiently large. The parameters are optimized during the **training phase**, when differences between the exact and predicted outputs propagate backwards from the output through the hidden layers. After the optimization, the network is fed with a new set of data, called **test set**, for the validation. Finally, it can be applied to predictions corresponding to new inputs.

The main theoretical result concerning the perceptron is a convergence theorem, rigorously stated and proved in [69], affirming that the output of this simple algorithm, with any parameters initialization, is compatible with its training examples.
However, this conclusion only takes into account pattern recognition tasks, and the bad performance of this approach in other tasks were pointed out by Minsky and Papert [66]. In fact, we would expect small variations in the weights and biases to produce little changes in the output, but this is not true for networks made of perceptrons. Therefore, for better stability **sigmoid neurons** are prefered, since they produce smaller variations in the output with respect to variations in the weights. Their output is computed as follows:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b),$$

where $\sigma(\cdot)$ is given by:

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{2}\left(1+\tanh\left(\frac{z}{2}\right)\right).$$

As shown in Figure 2.3, the behaviour of $\sigma(\cdot)$ is asymptotically equivalent to the step function used for the computation of the output of the perceptron.

Moreover, in this case if we consider variations Δw_j and Δb in the parameters, then each neuron's output changes as follows:

$$\Delta out \leq \sum_j \frac{\partial out}{\partial w_j}\Delta w_j + \frac{\partial out}{\partial b}\Delta b.$$

More in general, many functions can be used for the output determination by normalizing the product $\mathbf{w} \cdot \mathbf{x}$ between 0 and 1 (or -1 and 1), and they are called **activation functions**. In Figure 2.2 we can observe the schematic functioning of a generic neuron that, given an input $\mathbf{x}$, computes the output y as the evaluation of the activation function at the weighted sum of the data $\{x_i\}_{i=1}^n$.

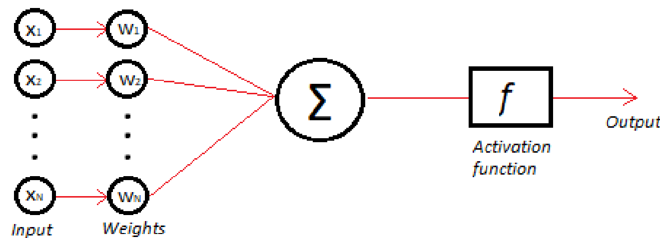


Figure 2.2: Scheme of neurons functioning: the output y is computed as the evaluation of the activation function f at the weighted sum of the inputs $\{x_i\}_{i=1}^n$ with respect to $\{w_i\}_{j=1}^n$.

The step function, signature of the perceptron, has zero gradient at every point, that implies saturation of the neurons and consequently is efficient for binary classfication only. Its most simple smoothing brings to the nonlinear sigmoid function, that presents no jumps but still provides an output in $[0, 1]$ and can make clear predictions. On the other hand, also the gradient of this activation is almost 0 almost everywhere, implying a slow learning as well. Moreover, the sigmoid is not centred around 0 and computationally expensive.

The centering around 0 can be easily fixed by passing to the **tanh** activation function:

$$\sigma(x) = \tanh x,$$

whose output is still bounded, but this time in $[-1, 1]$. This activation function provides better recognition accuracy, as proved in [45] and [46]. However, its shape is just the stretching of the sigmoid, so the issues about the gradient and expensive computational time are still present, due to the difficulties in evaluating the exponential function, overcome by the implementation of approximation techniques, such as [60].

It is also possible to use a linear activation function, even if it is not recommended for the hidden layers but only for the output one, since it is not able to represent complexity in practical applications, the **ReLU** activation function is defined by:

$$\sigma(x) = \max\{x, 0\}.$$

The ReLU is still not everywhere linear, so it can fit complex problems, but is very easy to compute, and has non-null gradient for every positive real value, so backpropagation of the error ensures no saturation of the neurons. Even better for backpropagation is the **leaky ReLU**, whose gradient is different from 0 also for negative real values:

$$\sigma(x) = \max\{\alpha x, x\},$$

with $\alpha > 0$. Finally, a smoothed version of the ReLU activation function consists of the so-called **softplus** activation, whose derivative is defined at every point as:

$$\sigma(x) = \ln(1 + e^x).$$

For further examples of activation functions and their comparison see [86].

As anticipated in the previous section, the choice of activation function is typically handcrafted and based on experience of some heuristics. However, as well as for the quantitative hyperparameters, also this feature can be automatically chosen by the ANN. The structures performing this additional task are called **Evolutionary ANNs** (EANNs), and combine evolution algorithms to learning [89, 93], allowing the adaptation of their topology to different tasks without the need for human intervention. Moreover, EANNs are less sensitive to weights initialization and network depth, do not require everywhere differentiability of the activation function and can avoid local minima. An example of such structure, based on learning a piecewise-linear activation function for every neuron

independently during training, is proposed by Agostinelli et al. [2].

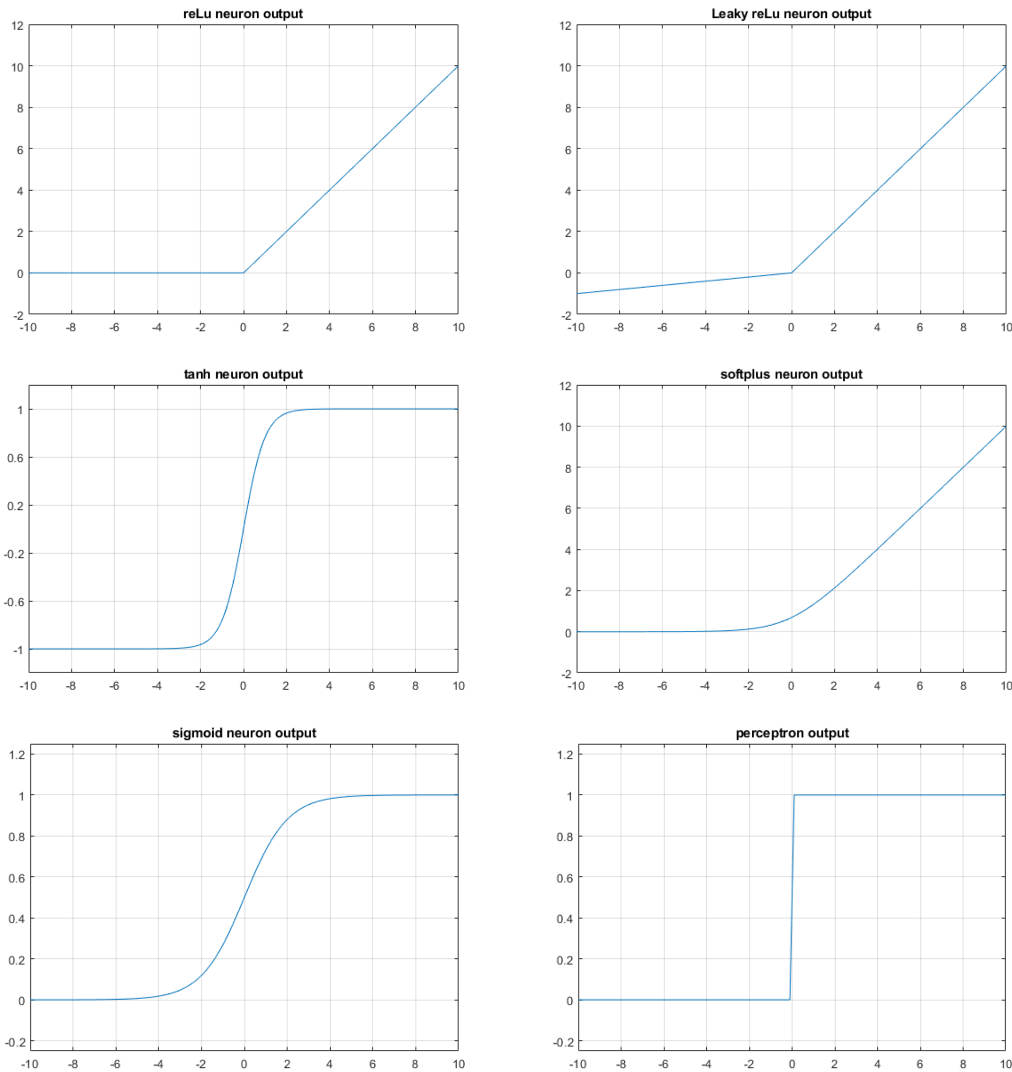


Figure 2.3: Plots of the most common activation functions

2.2. Backpropagation

The error produced by the network on the output depends both on the estimates of the parameters and on how many times the optimization is performed, i.e. on the hyperparameters introduced above.

Let $\{w^l_{jk}\}_{j,k,l}$, $j = 1, ..., N^l$, $k = 1, ..., N^{l-1}$, $l = 1, ..., L$, where N_l is the number of neurons in layer l and L the total number of layers, be the weight associated to the kth neuron in layer $l-1$ connected to the jth neuron in layer l, b^l_j the bias of neuron j in layer

l and a_j^l its output, called **activation** (see Figure 2.4 for a scheme). Then,

$$a_j^l = \sigma\left(\sum_j w_{jk}^l a_k^{l-1} + b_j^l\right). \tag{2.1}$$

Equation (2.1) can be rewritten as:

$$\sigma(\mathbf{z}^l) = \mathbf{a}^l = \sigma(W^l \mathbf{a}^{l-1} + \mathbf{b}^l), \tag{2.2}$$

by defining $\mathbf{z}^l := W^l \mathbf{a}^{l-1} + \mathbf{b}^l$, with $l = 1, ..., L$, and $W^l = \{w_{jk}^l\}_{j,k,l}$, $j = 1, ..., N^l$, $k = 1, ..., N^{l-1}$, $\mathbf{a}^l = \{a_j^l\}_{k=1}^{N^l}$ and $\mathbf{b}^l = \{b_j^l\}_{j=1}^{N^l}$, $\forall l = 1, ..., L$.

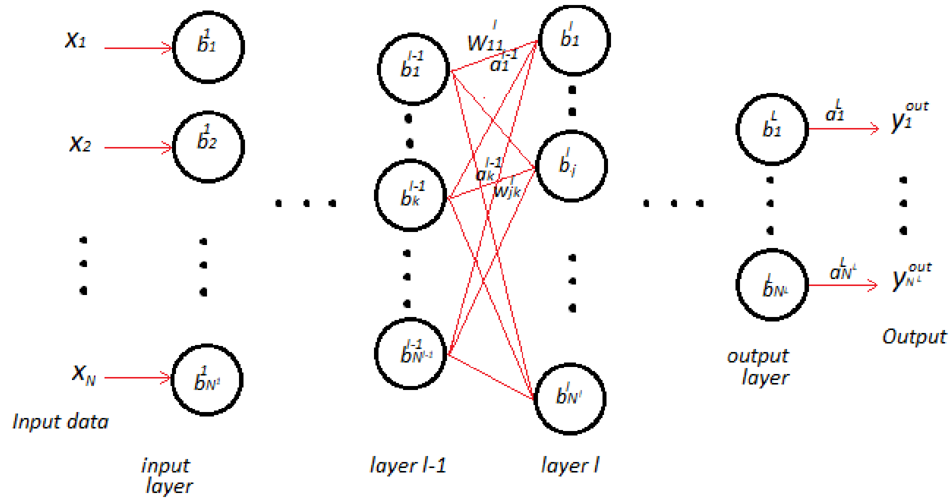


Figure 2.4: Scheme of backpropagation neurons.

Assume that the cost function to be minimized is given in terms of mean square errors (MSE):

$$J(\mathbf{x}; W, \mathbf{b}) = \frac{||\mathbf{y}^{out}(\mathbf{x}) - \mathbf{y}^{true}||^2}{2N}, \tag{2.3}$$

and denoted by $J_n = J_n(\mathbf{a}^L)$ the cost function for a single input x_n, with L = index of the output layer.

Define now the **error** related to neuron j in layer l as follows:

$$\delta_j^l := \frac{\partial J}{\partial z_j^l}, \qquad \forall j = 1, ..., N^l, \ \forall l = 1, ..., L.$$

Then, δ_j^l should be small in principle and we want it to get smaller.

In this framework, four equations describing the behaviour of the error depending on the network structure can be proved:

- *Equation for the error of the output layer L*:

$$\delta_j^L = \frac{\partial J}{\partial a_j^L}\sigma'(z_j^L), \qquad \forall j = 1, ..., N^L; \tag{2.4}$$

- *Equation for the error of layer l in terms of* δ^{l+1}:

$$\delta_j^l = \sum_k w_{kj}^{l+1}\sigma'(z_j^l)\delta_k^{l+1} \qquad \forall j = 1, ..., N^l, \; \forall l = 1, ..., L; \tag{2.5}$$

- *Equation of the variation of J with respect to* **b**:

$$\frac{\partial J}{\partial b_j^l} = \delta_j^l \qquad \forall j = 1, ..., N^l, \; \forall l = 1, ..., L; \tag{2.6}$$

- *Equation for the variation of J with respect to W*:

$$\frac{\partial J}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \qquad \forall j = 1, ..., N^l, \; \forall k = 1, ..., N^{l-1}, \; \forall l = 1, ..., L. \tag{2.7}$$

From (2.7) we notice that, if the output of one layer is small, i.e. $a_k^{l-1} \approx 0$, then the loss function does strongly depend on the weights in the following one, i.e. $\frac{\partial J}{\partial w_{jk}^l}$ is small. We say that the neuron learns slowly.

Moreover, since $\sigma' \approx 0$ for $z_j^l << 0$ or $z_j^l >> 0$, then in this cases $\frac{\partial J}{\partial b_j^l}$ is small as well, as a consequence of (2.5) and (2.6). Again the jth neuron of layer l learns slowly. We say that it is **saurated**.

Equations (2.4)-(2.7) can be of help when tuning the hyperparameters and chosing the activation function.

However, in feedforward ANNs the hidden neurons are not directly connected to the input data and the final output, thus the **backpropagation algorithm** shown in Algorithm 2.1 was introduced by Rumelhart et al. [82] for the iterative adjustment of the parameters in multilayered structures, based on the chain rule and the minimization of a function using the gradient descent algorithm, explained in the next section. The goal of this method is to build an internal representation that is suitable for performing the task.

Algorithm 2.1 Backpropagation Algorithm (input: data $\mathbf{x}$, target solution $\mathbf{y}^{true}$; output: approximate solution $\mathbf{y}^{out}$)

1: Initialize the weights $w^l_{jk} = w^l_{jk}(0)$, $\forall k = 1, ..., n_{l-1}$, $j = 1, ..., n_l$, $l = 1, ..., L$, set the learning rate $\eta \in (0, 1)$, a convergence threshold $\epsilon > 0$ and a maximum number of training iterations T

2: **while** $|J(\mathbf{x}; W, \mathbf{b})| < \epsilon \quad \& \quad t \leq T$ **do**

3: Compute the activation $\mathbf{a}^l$ of the hidden layers $l = 1, ..., L-1$, according to equation (2.2)

4: Compute the activation $\mathbf{a}^L$ of the output layer, according to equation (2.2)

5: Compute the error of each output unit (chain rule):

$$\delta^L_k = \left[(\mathbf{y}^{true})_k - a^L_k\right] \sigma'_L(z^L_k)$$

6: Compute the error of each output unit (chain rule):

$$\delta^l_k = \left(\sum_j \delta^{l+1}_j w_{kj}\right) \sigma'_l(z^l_k)$$

7: Update weights of the output layer:

$$w^L_{kj}(t+1) = w^L_{kj}(t) + \eta \delta^L_k z_j$$

8: Update weights of the hidden layers $l = 1, ..., L-1$:

$$w^l_{kj}(t+1) = w^l_{kj}(t) + \eta \delta^l_k a^{l-1}_j$$

9: Compute the MSE $J(\mathbf{x}; W(t), \mathbf{b})$, according to equation (2.3)

10: **end while**

11: The output value is $\mathbf{y}^{out} = \mathbf{a}^L$.

At the same time Rumelhart et al. [82], Parker [71] and Cun [23] have proposed similar ideas, based on the generalization of the single-layered perceptron network, and successively other derivations and generalizations of the algorithms were introduced. Indeed, it implements a highly efficient and simple neural network, but the convergence rate is rather slow. The generalizations are mainly divided into three categories: numerical-

based, heuristic-based and learning strategy-based algorithms. The first two rely on second-order optimization method instead of the gradient descent [8], the second kind on the systematic analysis of the learning process and the dynamic adaptation of learning rate [18, 42]. Finally, the process can be sped up by a selective presentation of the samples, for instance dividing them into groups according to the difficulty in learning [67] or preprocessing them in order to reduce redundancy. Cho and Kim [21] present some accelerated algorithms and conclude that the best performance is given by combinations of the three discussed approaches.

Finally, an important alternative derivation of the backpropagation was proposed by LeCun [51]. It is based on the Largangian formalism and consists in an alternative formulation as an optimization problem with nonlinear constraints, centred on the evaluation of a parametric function computed in several elementary steps. In next section we will analyze the network training phase from this perspective and introduce the most common algorithms for the optimization.

2.3. Gradient descent optimization algorithm

The learning phase consists of the estimation of the paramters $\mathbf{v} = (\mathbf{w}, b)$, $\mathbf{w} \in \mathbb{R}^N$, $b \in \mathbb{R}$, corresponding to each neuron, through the minimization of a given loss function. The loss function is usually expressed in terms of least square classification error, defined as:

$$C(\mathbf{x}, \mathbf{y}; \mathbf{v}) = \sum_{i=1}^{N} (\sigma(x_i; \mathbf{v}) - y_i)^2, \tag{2.8}$$

where $\mathbf{x} = [x_1, x_2, ..., x_N]^T$ are the input data and $\mathbf{y}$ the corresponding exact values.

A classical optimization method for the minimization of a general loss function $C(\mathbf{v})$ is the **gradient descent method**. It is based on the progressive improvement of an initial guess towards a local minimum moving along the gradient direction.
In Algorithm 2.2 we report the gradient descent method.

Since $\Delta C^{\mathbf{k}} = C(\mathbf{v}^{\mathbf{k+1}}) - C(\mathbf{v}^{\mathbf{k}}) \approx -\eta \left\|\nabla C(\mathbf{v}^{\mathbf{k}})\right\|^2 \leq 0$, then $C(\mathbf{v}^{\mathbf{k+1}}) = C(\mathbf{v}^{\mathbf{k}}) + \Delta C^{\mathbf{k}} \leq C(\mathbf{v}^{\mathbf{k}})$, and therefore the algorithm moves towards a local minimum. Here $|| \cdot ||$ is the Euclidean norm and η the fixed learning rate.
We oberse that, since C also depends on the data, they must be taken into account in the determination of $\boldsymbol{\Delta}\mathbf{v}^{\mathbf{k}}$. A basic method to do so is called **batch gradient descent**, that simply sums the gradient evaluated at all the training data, substituting the computation of the optimization delta in Algorithm 2.2, step 4, as follows:

$$\mathbf{\Delta v^k} = -\frac{\eta}{N}\sum_{i=1}^{N}\nabla_{\mathbf{v}}C(x_i, y_i, \mathbf{v^k}). \quad (2.9)$$

Algorithm 2.2 Gradient Descent Algorithm (input: loss function $C(\cdot)$; output: optimal parameters vector $\mathbf{v}$)

1: Set a learning rate $\eta \in (0, 1]$ and a convergence condition
2: Initialize the parameters $\mathbf{v}^0$ and compute the corresponding loss function $C(\mathbf{v}^0)$
3: **for** $\mathbf{k} = 0, 1, 2, ...$ up to convergence **do**
4: Compute the optimization step:

$$\mathbf{\Delta v^k} = -\eta\nabla C(\mathbf{v^k})$$

5: Update the parameters:

$$\mathbf{v^{k+1}} = \mathbf{v^k} + \mathbf{\Delta v^k}$$

6: Compute the corresponding loss function $C(\mathbf{v^{k+1}})$
7: **end for**

Du et al. [28] proved that the batch gradient descent can obtain 0 training loss in multilayered fully connected or convolutional Artificial Neural Networks with regular non-polynomial activation functions and objective function given by equation (2.8). The loss moreover decreases with geometric rate at every step and, under some balancing conditions on the weights initialization, also linear convergence to the global minimum can be achieved [5].

As we can easily observe from the expression of equation (2.9), every iteration of the batch gradient descent algorithm possibly requires the evaluation of the gradient of the loss function at each sample $\{x_i, y_i\}_{i=1}^{N}$, introducing redundancy that results into a slow method.

In order to overtake this limitation, the **stochastic gradient descent** method has been introduced by Bottou et al. [16], defined as Algorithm 2.3:

Algorithm 2.3 Stochastic Gradient descent Algorithm (input: loss function $C(\cdot)$; output: optimal parameters vector $\mathbf{v}$)

1: Set a learning rate $\eta \in (0, 1]$ and a stopping criterion
2: Initialize the parameters $\mathbf{v}^0$ and compute the corresponding loss function $C(\mathbf{v}^0)$
3: **for** $\mathbf{k} = 1, 2, ...$ up to convergence **do**
4: Randomly shuffle the data
5: Compute the optimization step

$$\boldsymbol{\Delta}\mathbf{v}^{\mathbf{k}} = -\eta \nabla C(x_i, y_i, \mathbf{v}^{\mathbf{k+1}})$$

6: Update the parameters

$$\mathbf{v}^{\mathbf{k+1}} = \mathbf{v}^{\mathbf{k}} + \boldsymbol{\Delta}\mathbf{v}^{\mathbf{k}} \tag{2.10}$$

7: Compute the corresponding loss function $C(\mathbf{v}^{k+1})$
8: **end for**

Stochastic gradient descent method provides faster convergence when the samples are redundant, is able to escape from local stationary points and achieves better minima than the batch gradient descent algorithm. However, it is not optimal. Some generalization are also proposed by Bottou et al. [16] for the application of this method even to functions that are non-differentiable on sets of zero measure, where the gradient for the computation of the increment in Algorithm 2.3, step 5, is modified as follows:

$$\boldsymbol{\Delta}\mathbf{v}^{\mathbf{k+1}}0 = -\eta H(\mathbf{v}^{\mathbf{k}}), \qquad \text{with} \qquad \mathbb{E}[H(\mathbf{v})] = \nabla_{\mathbf{v}}(\mathbf{v}) \ \forall \mathbf{v}.$$

In particular, this coincides with the stochastic gradient descent when J is differentiable everywhere.

A compromise between batch and stochastic gradient is the **mini batch gradient descent** method, that groups the data into N_b small batches of fixed size B, and performs a step of batch method on each group, as shown in Algorithm 2.4:

Algorithm 2.4 Mini Batch Gradient descent Algorithm (input: loss function $C(\cdot)$; output: optimal parameters vector $\mathbf{v}$)

1: Set a learning rate $\eta \in (0, 1]$ and a convergence condition
2: Set the number of batches B to divide the samples set into
3: Initialize the parameters $\mathbf{v}^0$ and compute the corresponding loss function $C(\mathbf{v}^0)$
4: **for** $\mathbf{k} = 1, 2, ...$ up to convergence **do**
5: Randomly shuffle the data
6: Divide the samples set into $N_b = n/B$ batches
7: Compute the optimization step

$$\boldsymbol{\Delta}\mathbf{v}^{\mathbf{k}} = -\eta \sum_{i=1+nB}^{B(n+1)} \nabla_{\mathbf{v}} C(x_i, y_i, \mathbf{v}^{\mathbf{k+1}}).$$

8: Update the parameters: $\mathbf{v}^{\mathbf{k+1}} = \mathbf{v}^{\mathbf{k}} + \boldsymbol{\Delta}\mathbf{v}^{\mathbf{k}}$.
9: **for** $n = 2, 3, ..., N_b$ **do**
10: Randomly shuffle the data
11: Compute the optimization step

$$\boldsymbol{\Delta}\mathbf{v}^{\mathbf{k}} = -\eta \sum_{i=1+nB}^{B(n+1)} \nabla_{\mathbf{v}} C(x_i, y_i, \mathbf{v}^{\mathbf{k+1}}).$$

12: Update the parameters: $\mathbf{v}^{\mathbf{k+1}} \mathrel{+}= \boldsymbol{\Delta}\mathbf{v}^{\mathbf{k}}$.
13: **end for**
14: Compute the corresponding loss function $C(\mathbf{v}^{k+1})$.
15: **end for**

This method provides variance reduction in the gradient estimation with noisy data and better scalability in the stochastic gradient descent by converting it into a parallel and distributed algorithm [25].

2.3.1. Adam optimizer

Unlike the algorithms listed above, the learning rate may be adapted over the iterations, and its periodic reduction can speed up the learning, in addition to reduce the probability of settling into a local minimum [81].

The method applied in this report, called Adam optimizer, is based on this idea and is widely used in machine learning [43, 90].

This algorithm relies on the Root Mean Square propagation (RMSprop) method, that works well in on-line and non-stationary settings, for problems where the loss function has saddle points, and it is coupled with a mini batch approach. The concept is applied in practice by dividing at each step η by the square root of the second moment of the gradient, i.e.

$$\mathbf{\Delta v^k} = -\frac{\eta}{\sqrt{E[\nabla C(\mathbf{v^k})^2]}}\nabla C(\mathbf{v^k}).$$

The Adam optimizer uses biased-corrected estimations of the first and second moment of the gradients to compute each optimization step $\mathbf{\Delta v^k}$.

For $k \geq 1$, define $\mathbf{g^k} = \nabla C(\mathbf{v^k})$, and let $\mathbf{m^k}$ and $\mathbf{u^k}$ be the (biased) estimates of the first and second order moments of $\mathbf{g^k}$, for every iteration $\mathbf{k}$. The algorithm updates exponential moving averages of the gradient and the squared gradient with the hyper-parameters $\beta_1, \beta_2 \in [0, 1)$ controlling their exponential decay rates.

The pseudo-code of the Adam method proposed by Kingma and Ba [47] is reported in Algorithm 2.5. The authors also prove that it has $O(\sqrt{K})$, with $K =$ number of iterations, convergence order. This estimate has been improved for convex loss functions with convex and bounded gradient as follows:

$$\frac{R(K)}{K} = O\left(\frac{1}{\sqrt{K}}\right),$$

where $R(K) = \sum_{k=1}^{K}\left[J_k(\mathbf{w}^k) - J_k(\mathbf{w}^*)\right]$ is the considered measure of error [15]. Furthermore Bock and Weiß [14] also proved local convergence with exponential rate.

Finally, some of the advantages of the Adam optimizer are that the magnitudes of parameter updates are invariant to rescaling of the gradient, the step sizes are approximately bounded by the parameter η, stationarity of the loss function is not required, and the method works with sparse gradients. The only drawback of this algorithm and of adaptive learning rate methods in general with respect to the stochastic gradient descent consists in their poor generalization capabilities. However, some extensions of the Adam optimizer seem to perform better and fill the gap [97].

Algorithm 2.5 Adam Optimization Algorithm (input: loss function $C(\cdot)$; output: optimal parameters vector $\mathbf{v}$)

1: Fix a learning rate η the exponential decay rates β_1 and β_2 and a convergence condition

2: Initialize the parameters $\mathbf{v^0}$ and the moments $\mathbf{m^k} = 0$, $\mathbf{u^k} = 0$, and compute the corresponding loss function $C(\mathbf{v^0})$

3: **for** $\mathbf{k} = 1, 2, ...,$ up to convergence **do**

4: Set $\mathbf{g^k} = \nabla_{\mathbf{v}} C(\mathbf{v^{k-1}})$

5: Set $\mathbf{m^k} = \beta_1 \mathbf{m^{k-1}} + (1-\beta_1)\mathbf{g^k}$

6: Update $\mathbf{u^k} = \beta_2 \mathbf{u^{k-1}} + (1-\beta_2)(\mathbf{g^k})^2$

7: Compute the unbiased first moment estimate: $\hat{\mathbf{m}}^{\mathbf{k}} = \frac{\mathbf{m^k}}{1-\beta_1^k}$

8: Compute the unbiased second moment estimate: $\hat{\mathbf{u}}^{\mathbf{k}} = \frac{\mathbf{u^k}}{1-\beta_2^k}$;

9: Compute the optimization step:

$$\mathbf{\Delta v^k} = -\eta \frac{\hat{\mathbf{m}}_{\mathbf{k}}}{\sqrt{\hat{u}_k} + \epsilon}$$

10: Update the parameters

$$\mathbf{v^{k+1}} = \mathbf{v^k} + \mathbf{\Delta v^k}$$

11: Compute the corresponding loss function $C(\mathbf{v^{k+1}})$.

12: **end for**

2.4. Universal approximation property

From the previous sections, we have understood that neural networks can be seen as rules for computing output values given inputs. Indeed, backpropagation allows multilayered feedforward ANNs to learn input-output mappings from training samples. Thus, we can think of approximating unknown functions given some observed values in points of their definition domains.

The choice of activation plays an important role in the approximation of a function via neural networks. In this section we will prove some results regarding the ability of the sigmoid function in doing so. The conclusions reported in this section rely on the dissertation [24] by Cybenko.

Let us first introduce some definitions:

Definition 2.4.1 (n-discriminatory function). *Consider the measurable space* (I_n, M) *and let* $\mu \in M$*, with* $I_n = [0,1]^n$*, be a measure. Then, a function* σ *is n-discriminatory using* μ *if*

$$\int_{I_n} \sigma(wx+\theta)d\mu(x) = 0 \qquad \forall w \in \mathbb{R} \qquad \Longrightarrow \qquad \mu = 0.$$

As a consequence, an n-discriminatory function does not lose any information about the input, since it does not map the affine space of type $wx - \theta$ into a 0-measure set.

Definition 2.4.2 (Discriminatory function). *A function* σ *is discriminatory with respect to a measure* μ *if*

$$\int \sigma(wx+\theta)d\mu = 0 \qquad \forall w \in \mathbb{R}^n \ \forall b \in \mathbb{R} \ \forall n \in \mathbb{N} \qquad \Longrightarrow \qquad \mu = 0.$$

Then, the following result holds:

Theorem 2.1. *Let* σ *be a sigmoidal continuous function. Then,* σ *is discriminatory for every measure.*

Consider now a generic functional space S with a metric d and let $f \in (S,d)$ be the target function to be approximated and $g \in U \subset S$ the approximate solution.

Definition 2.4.3 (Universal approximator). *A neural network is a universal approximator in a metric space* (S,d) *if*

$$\forall f \in S, \ \forall \epsilon > 0 \ \exists g \in U \text{ such that } d(f,g) < \epsilon,$$

i.e. U *is dense in* S.

The output of a generic single-layered neural network with N neurons can be expressed as the following sum:

$$G(\mathbf{x}) = \sum_{J=1}^{N} \alpha_j \sigma(\mathbf{w}_j \mathbf{x} + b_j). \tag{2.11}$$

Then,

$$U = \left\{ G : G(\mathbf{x}) = \sum_{J=1}^{N} \alpha_j \sigma(\mathbf{w}_j \mathbf{x} + b_j) \text{ for some } \mathbf{w}_j \in \mathbb{R}^n, \ \alpha_j, b_j \in \mathbb{R} \right\} \tag{2.12}$$

is the output space of such network, and the following result holds:

Theorem 2.2 (Cybenko)**.** *Let σ be a sigmoidal function and consider a network whose output can be expressed as in equation* (2.11)*. Then, the linear space of functions* (2.12) *is dense in $C(I_n)$.*

In conclusion, feedforward dense neural networks with sigmoid activation functions are universal approximators for continuous functions (see [38] for details), i.e.

$$\forall f \in C(I_n)\ \exists \hat{f} \in U \text{ such that } d(f, \hat{f}) < \epsilon \quad \text{for some } \epsilon > 0.$$

Hornik et al. [38] also proved that neural networks satisfying the hypotheses of Stone-Weierstrass theorem can even approximate any bounded measurable function, based on the proof of Funahashi [32] of the ability of neural networks with at least one hidden layer to approximate any measurable function on compact sets:

Theorem 2.3. *For any measurable function $f : \mathbb{R}^d \longrightarrow \mathbb{R}$ there exist a compact subset $K \subset \mathbb{R}^d$ and a feedforward neural network $\hat{f}(\cdot;\theta)$ such that $\forall \epsilon > 0 \quad \exists \mu(K) < 1 - \epsilon$ and $\forall \mathbf{x} \in K \quad |f(\mathbf{x}) - \hat{f}(\mathbf{x};\theta)| < \epsilon$.*

An error bound for neural network approximation was proved by Barron [6] as is summarized in the following theorem:

Theorem 2.4. *Let f be a function in $\mathbb{R}^d$ and $\tilde{f}(\omega)$ its Fourier transform. Define $C_f := \int_{\mathbb{R}^d} |\omega||\tilde{f}(\omega)|d\omega$.*
If C_f is finite, then there exists a linear combination of sigmoidal functions $\{f_n\}_{n\in\mathbb{N}}$ such that

$$\int_{B_r} [f(x) - f_n(x)]^2 \mu(dx) \leq \frac{(2rC_f)^2}{n} \qquad \forall n \in \mathbb{N}.$$

Another powerful application of ANNs is the possibility to approximate functions that are not differentiable everywhere but possess generalized derivative, such as piecewise differentiable ones. Moreover, Hornik et al. [39] and Gallant and White [33] proved that both functions belonging to Sobolev spaces and their generalized derivatives can be estimated with arbitrary accuracy by multiple-input-single-output neural network with at least one hidden layer. This has many applications, for instance in economics [30], and paves the way to approximation of the solution to differential equations [49], as we will se in next chapter.

3 | Numerical solution of PDEs via ANNs

In this section we present a method for the approximate solution of general PDEs, based on the one proposed by Xu et al. [91].

As anticipated in Chapter 1, it involves deep feedforward neural networks. In particular, it is made of two structures, one taking as input random points on the boundary and the another independent one taking as input random points from the inner domain. They both give as output the approximate solution at those locations, and then their results are summed to obtain the final lifted estimation, as explained in detail in the following.

Before entering in the details we provide a brief overview of the state of the art. Let us start with a brief overview of the main results in the application of ANNs for the approximate solution of differential equations. An extensive introduction to the subject can be found in [92].

As explained in the previous chapter, to each node of a neural network corresponds a set of parameters, updated at every learning iteration so that a given loss function is minimized, and then used for the computation of an output, according to a given activation function. According to Cybenko [24] and Hornik et al. [38][39], neural networks are universal approximators for a large class of functions and are also able to estimate their derivatives. Therefore, many deep learning approaches were proposed for the estimation of PDE solutions.

Consider a generic boundary value problem of the form:

$$\begin{cases} Lu(x) = f(x) & \text{in } \Omega, \\ u = g_D & \text{on } \partial\Omega. \end{cases}$$

The first approach to approximate the solution of a differential problem with Dirichlet boundary conditions with ANNs was made by Dissanayake and Phan-Thien [27]. The

approach was based on a multilayered perceptron, taking as input locations $\{x_i\}_{i=1}^N$ over the domain and giving as output the evaluation of u there, where the optimal weights and biases $p = \{W, \mathbf{b}\}$ are determined by minimization of the following loss function:

$$J(\mathbf{x}; p) = \sum_{i=1}^{n} [(L\Psi_t(x_i, p) - f(x_i))^2 + (\Psi_t(x, p) - g_D(x_i))^2],$$

involving a penalty term for the enforcement of the boundary constraint.

Later, Lagaris et al. [49] proposed to evaluate the parameters (weights and biases) $p = \{W, \mathbf{b}\}$ of the network $\hat{u}(\cdot; p)$ describing the solution u by an optimization least square error problem trying to fit the right-hand-side of the PDE, subject to the boundary conditions, i.e.

$$\begin{cases} p^* = \text{argmin}_p [L\hat{u}(x, p) - f(x)]^2 \\ \text{st boundary conditions.} \end{cases} \tag{3.1}$$

This can be solved either with constrained nonlinear programming techniques or by transforming it into an unconstrained problem and solve it with the corresponding (more efficient) algorithms.

In order to exploit the second option, the trial solution can be defined as follows:

$$\Psi_t(x; p) = \hat{\Psi}(x) + F(x)N(x; p), \tag{3.2}$$

where $\hat{\Psi}(x)$ is a function satisfying the bounday conditions, $N(x; p)$ an ANN and $F(x)$ a real function whose value on the bounday is 0. The feedforward neural network $N(x; p)$ of parameters p, approximates the solution at every point x of the domain, whithout taking into account the boundary conditions. It is then mutiplied by a function $F(x)$ such that $F = 0$ on the boundary and summed to a smooth extension of the boundary condition to the whole domain $\hat{\Psi}(x)$. Notice that, the boundary conditions are exactly satisfied by construction by Ψ_t, thanks to the additive term $\hat{\Psi}$.

The training set is given by a discretization of the domain $\hat{\Omega} = \{x^{(i)} \in \Omega \,:\, i = 1, 2, ..., N\}$ and the loss minimized during the learning process is the following:

$$J(p) = \sum_{i=1}^{N} \left(\nabla\Psi_t\left(x^{(i)}; p\right) - f\left(x^{(i)}\right)\right)^2.$$

Many proposals on the shape of $\hat{\Psi}$ were introduced, starting by Lagaris et al. [50], who

apply a multilayer perceptron for the basic PDE approximation and a radial basis function (RBF) network [70] for an initialization of $\hat{\Psi}$, followed by the exact imposition of the boundary condition. Malek and Beidokhti [58] derived general formulations for F and $\hat{\Psi}$ for different high order differential equations, while Chiaramonte et al. [20] takes $\hat{\Psi} = 0 \quad \textbf{in } \Omega$.

Finally, the optimal solution $\Psi_t(x; p^\star)$ is given by the trial solution with parameters $p^\star = \arg\min_p J(p)$.

A proof of the existence of an approximating neural network, whose error can be controlled, for the solution of elliptic PDEs with given boundary conditions can be found in [34]. The authors also point out that such machine learning-based methods are free from the curse of dimensionality. However, a drawback of these approaches is the arbitrary construction of the architecture, but it can be easily overcome by the introduction of **Finite Element NN**s, made of exactly M inputs and one hidden layer with N groups of N neurons each, where M is the number of elements in the FEM mesh and and N the number of nodes [9, 78].

An extension to parabolic problems was proposed by Sirignano and Spiliopoulos [87] with their Deep Galerkin Method (DGM). In DGM, N random points $(t_n, x_n) \in [0, T] \times \Omega$ and $(\tau_n, z_n) \in [0, T] \times \partial\Omega$, $n = 1, ..., N$, are sampled according to two probability densities ν_1, ν_2, then the loss function is computed as the sum of square errors with penalization terms related to the fitting of the boundary and initial condition, defined as follows:

$$J(\mathbf{t}, \mathbf{x}; \theta, \mathbf{s}) = \sum_{n=1}^{N} (\partial_t \Psi_t(t_n, x_n; \theta) + \nabla \Psi_t(t_n, x_n; \theta) - f)^2 + \\ + (\Psi_t(\tau_n, z_n; \mathbf{s}) - g_D(\tau_n, z_n))^2 + (\Psi_t(0, x_n; \theta) - u_0(x_n))^2,$$

where θ and $\mathbf{s}$ are network parameters learned in the training phase.
In this case two independent networks are used, one with parameters θ taking as input data $\{(t_n, x_n)\}_{n=1}^N \subset [0, T] \times \Omega$ and the other with parameters $\mathbf{s}$ taking as input $\{(\tau_n, z_n)\}_{n=1}^N \subset [0, T] \times \partial\Omega$, approximating the solution respectively in $[0, T] \times \Omega$ and $[0, T] \times \partial\Omega$. The authors in [87] have proved that the solution given by a neural network with one hidden layer and k neurons strongly converges to the exact solution as $k \to \infty$ in L^p $\forall p < 2$.
The DGM was also reformulated in order to avoid the computation of high order derivatives, that were proved to introduce instability in the estimation, by Lyu et al. [57],

rewriting the PDE as a first order system thanks to the introduction of auxiliary variables.

Another neural network approach for time-dependent problems was introduced by Chen et al. [19], focused on the solution of the linear transport equation. The authors propose as trial solution the output of a single network

$$\psi^{nn}(\mathbf{x}, t; p),$$

whose learning parameters are set by the minimization of the following loss function:

$$J(\mathbf{x}, t; p) = J_{GE}(\mathbf{x}, t; p) + J_{IC}(\mathbf{x}; p) + J_{BC}(\mathbf{x}, t; p),$$

made of three components, aiming at the minimization of the error corresponding to the general equation, the initial condition and the boundary condition, respectively.
As a consequence to the universal approximation property, it is possible to prove the convergence of this method in the L^∞ norm to the exact solution.

More complicated PDE problems have been also studied. A solution to the Stokes problem was proposed by Baymani et al. [7], and consists of the transformation of the Stokes system into three independent Poisson problems, each one solved by a feedforward neural network, while in [65] compressible Euler equations is analyzed.

All the previously discussed methods require physical knowledge of the system, given by the differential equation. Raissi [76] and Long et al. [55] have instead applied a deep learning algorithm to a set of scattered data, aiming at discovering the underlying PDE space-time model, starting from the following representation of a generic nonlinear PDE: $u_t = F(x, u, \nabla u, \nabla^2 u, ...), \ \forall x \in \Omega \subset \mathbb{R}^2, \ t \in [0, T]$. In the first paper [76], both u and F are represented by feedforward neural networks, respectively made of 5 hidden layers with 50 neurons each and 2 hidden layers with 100 neurons each, both trained by minimizing the sum of square errors $\sum_{n=1}^{N}(|u(t_n, x_n) - u_n|^2 + |u_t - F(t_n, x_n, \nabla u_n, \nabla^2 u_n, ...)|^2)$. The second approach [55] proposes a much more complex blocks structure where each block corresponds to a feedforward neural network, whose parameters are shared among all the blocks. Moreover, Raissi et al. [77] propose a machine learning method based on backpropagation for the automatic differentiation of the network output with respect to its training variables, introducing more stability.

3.1. Network structure

The method proposed in this project is based on the work of K. Xu, B. Shi and S. Yin [91] and implements coupled feedforward deep neural networks, one approximating the solution on the boundary and the other in the interior of the domain. In this section it is introduced for the solution of the Poisson problem, while at the end of the chapter I discuss the extension to general elliptic equations (Section 3.4.1, in particular, the advection-diffusion problem) and time-dependent problems (Section 3.4.2).

The expression of the trial solution is based on the equation (3.2) of Lagaris et al. [49]. As anticipated in the previous section, the shape of the boundary term $\hat{\Psi}$ can be adjusted for improving specific performance of the deep learning method. When look for the minimum error possible, it can be directly substituted with the boundary value if it is set by Dirichlet conditions or by some functions constructed in order to exactly satisfy any other type of boundary condition. These methods are however often very expensive from the computational point of view, thus in this thesis the investigated method involves an expression of $\hat{\Psi}$ that guarantees only approximate imposition of the boundary condition. This choice makes the algorithm less expensive and more intuitive.

In two dimensions, the trial solution in is expressed as follows:

$$u_h(x, y; w_1, w_2) = A(x, y; w_1) + B(x, y)N(x, y; w_2), \tag{3.3}$$

where $A(x, y; w_1)$ is the **boundary network** and $N(x, y; w_2)$ is the **PDE network**. Here w_1 ans w_2 are the parameters learned by the neural networks A and N during the training phase, respectively. Both A and N are feedforward dense neural networks made of sigmoid neurons. The choice of the activation function for the hidden layers is based on the discussion made in Section 2.1, and in particular the conclusion about the high recognition accuracy of the hyperbolic tangent (tanh) proved in [45, 46], while for the output layer linear activation is chosen. Both structures are trained using the backpropagation algorithm with Adam optimizer, chosen for its fast convergence and the error estimate discussed in Section 2.3.1. The number of layers and neurons and the learning rate of the optimization algorithm will be set by trial and error, as we will see in Chapter 5, concerning stationary numerical experiments, and none of the autotuning techniques introduced in Chapter 2 are implemented. These hyperparameters will be fixed for both architectures with the same values, in order to maintain symmetry and reduce the human intervention in the tuning to the minimum. Moreover, since neural networks with only one hidden layer are theoretically able to approximate any continuous functions

[24], we will observe the behaviour of the coupling made of structures with a number of hidden layers that varies among 1, 2 and 3, in order to understand if growing depth can actually detect additional features of the solution.

In (3.3) $N(\cdot,\cdot;w_2)$ plays the same role as the neural network $N(\cdot,p)$ in the trial solution (3.2), giving as output the approximate solution of the PDE without taking into account the boundary condition. It is then multiplied by a function B that is 0 on the boundary, and the lifting operator in this case is substituted by another independent neural network $A(\cdot,\cdot;w_2)$, that approximates the solution on $\partial\Omega$ by exploiting the information given by the boundary condition.

Figure 3.1 shows the structure of the coupling between A and N.

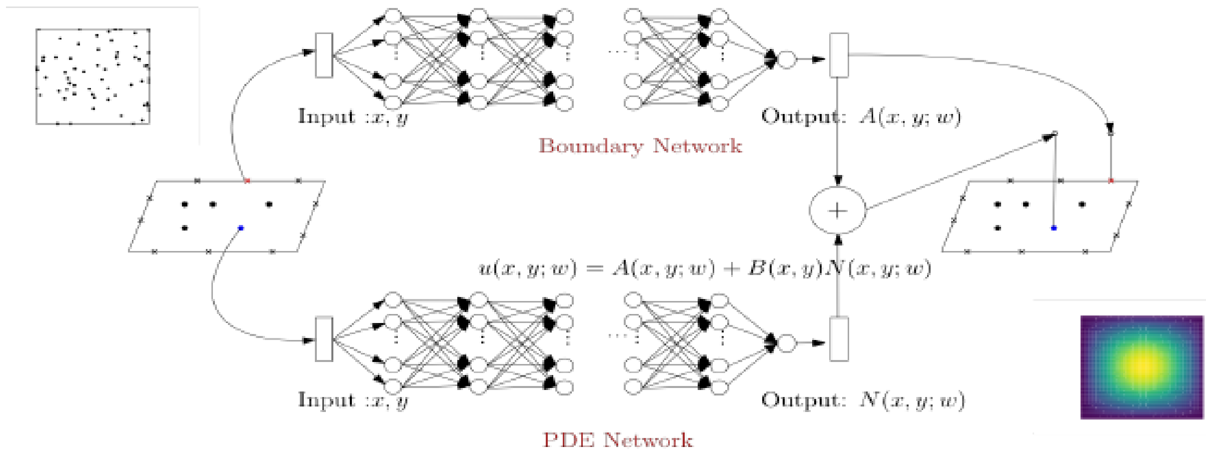


Figure 3.1: Coupled neural network structure. The value of the solution on the boundary is given by the network A taking as input some points on the boundary, while in the inner domain it is the sum between the outputs of A and the PDE network N, multiplied by a function B whose value on the boundary is 0.
(Taken from [91])

The inputs of the two networks are given by coordinate points in the domain and the corresponding evaluation of the problem data, i.e. f for the PDE network and g_D for the boundary one. In particular, a finite number n_b of points $\{\hat{\mathbf{x}}_i\}_{i=1}^{n_b}$ is sampled on the boundary and a finite number n_p of points $\{\mathbf{x}_i\}_{i=1}^{n_p}$ in the interior of Ω (see Figure 3.2), then $\{f(\mathbf{x}_i)\}_{i=1}^{n_p}$ and $\{g_D(\hat{\mathbf{x}}_i)\}_{i=1}^{n_b}$ are computed.

The difference between this method and the ones presented in the previous chapter is that the Dirichlet condition is not directly imposed by a deterministic function, but it is approximated by an independent network, so that $A(x,y;w_1)\big|_{\partial\Omega} \approx g_D \ \forall(x,y) \in \partial\Omega$. At every learning iteration for $N(x,y;w_2)$ a new set of boundary points is sampled and

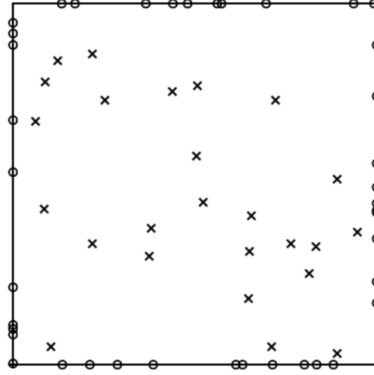


Figure 3.2: Data sampling: boundary points (°) and interior points (×)

$A(x, y; w_1)$ is trained on them, updating its parameters w_1 by minimizing the boundary loss function

$$J_b(\hat{\mathbf{x}}, \hat{\mathbf{y}}) = \sum_{i=1}^{n_b} [(g_D)_i - A(\hat{x}_i, \hat{y}_i; w_1)]^2. \tag{3.4}$$

The network $N(x, y; w_2)$ approximates the solution to the partial differential equation, minimizing the error with respect to the external source f, through the following loss function:

$$J_p(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n_p} [f(x_i, y_i) - \Delta N(x_i, y_i; w_2)]^2. \tag{3.5}$$

The PDE output is then multiplied by the function B that is zero on the boundary. In the case $\Omega = (0, 1)^2$, we consider:

$$B(x, y) = x(x-1)y(y-1),$$

and add to the output of $A(x, y; w_1)$, so that at every learning iteration the approximate solution minimizes the overall error. Notice that, beacuse of the non-exact term A, the boundary condition is not satisfied by construction but only approximately.

Refer to Algorithm 3.1 for details.

Algorithm 3.1 Networks coupling Algorithm (input: a coordinate point $(x, y) \in \Omega$, expression of B such that $B\big|_{\partial\Omega} = 0$; output: approximate solution of the PDE u)

1: Obtain the optimal parameters w_1^* and w_2^* by training the boundary network A and the PDE network N according to Algorithm 3.2
2: Predict the value u_b as the output of the network $A(x, y; w_1^*)$ corresponding to (x, y)
3: Predict the value u_p as the output of the network $N(x, y; w_2^*)$ corresponding to (x, y)
4: Compute the approximate solution of the PDE at the point (x, y):

$$u = u_b + B(x, y)u_p.$$

3.2. Training algorithm

In this section the main steps of the coupled training algorithm are presented. For a more detailed explanation of the Python code, see Appendix A.

The number of training iterations of the PDE network is manually set for every experiment. Each training iteration of $N(\cdot, \cdot; w_2)$ actually starts with a training loop for the boundary network, followed by a random sampling of points over Ω, used as input, and the minimization with the Adam optimizer of the loss function (3.5). The number of training iterations of $A(\cdot, \cdot; w_2)$ at every step of the loops we have just discussed cannot be manually set, instead. Indeed, we are guaranteed at least the same number of iterations as the ones chosen for N, and in order to maintain the symmetry discussed above it could be enough. However, we know that the boundary condition is much more easy to estimate than the PDE solution and less computationally expensive, since it does not require the computation of high order derivatives (or none at all in the Dirichlet case) and because its domain has dimension equal to the PDE input domain minus 1. As a consequence, we expect the boundary loss to achieve small values in less steps, thus we train A once and stop if the loss order is lower than a threshold, that we will fix to 10^{-5} or otherwise perform more iterations.

At each step the sum of square errors is minimized:

$$\begin{aligned} &SSE(\hat{x}_i, \hat{y}_i, x_i, y_i; w_1, w_2) = \\ &\quad = \sum_{i=1}^{n_b} \left[g_D(\hat{x}_i, \hat{y}_i) - u_h(\hat{x}_i, \hat{y}_i, w_1, w_2)) \right]^2 + \sum_{i=1}^{n_p} \left[f(x_i, y_i) - \Delta u_h(x_i, y_i; w_1, w_2)) \right]^2 . \quad (3.6) \end{aligned}$$

Algorithm 3.2 Training Algorithm (input: expression of the PDE data g_D and f; output: optimal networks parameters w_1 and w_2)

1: Initialize the boundary network parameters w_1
2: Initialize the PDE network parameters w_2
3: Set a convergence threshold for the training of the boundary network $\epsilon = 10^{-5}$ and a number of training iterations for the PDE network
4: **for** $i = 1, 2, ..., M$ **do**
5:     **for** $j = 1, 2, .., N$ **do**
6:         Sample n_b random points on the boundary $\{(\hat{x}_i, \hat{y}_i)\}_{i=1}^{n_b}$
7:         Evaluate the boundary condition on the training set $g_D(\hat{x}_i, \hat{y}_i)$ $\forall i = 1, 2, ..., n_b$
8:         Train the boundary network A and update the parameters w_1 by minimizing the loss function

$$J_b(\mathbf{x}, \mathbf{y}; w_1) = \sum_{i=1}^{n_b} [g_D(\hat{x}_i) - A(\hat{x}_i, \hat{y}_i; w_1)]^2$$

9:         **if** $j == 1$ and $J_b(\mathbf{x}, \mathbf{y}; w_1) < \epsilon$ **then**
10:             stop
11:         **end if**
12:     **end for**
13:     Sample n_p random points in the interior of the domain $\{(x_i, y_i)\}_{i=1}^{n_b}$
14:     Evaluate the external source on the training set $f(x_i, y_i)$ $\forall i = 1, 2, ..., n_p$
15:     Compute the numerical Laplacian $\Delta N(x_i, y_i; w_2)$ of the output of the PDE network, $\forall i = 1, ..., n_p$
16:     Train the PDE network N and update the parameters w_2 by minimizing the loss function

$$J_p(\mathbf{x}, \mathbf{y}; w_2) = \sum_{i=1}^{n_p} [f(x_i, y_i) - \Delta N(x_i, y_i; w_2)]^2$$

17: **end for**

3.3. Error metrics

In order to compare the performance of the method, three different error metrics are computed.

Definition 3.3.1 (Bounding loss).

$$L_b = \sum_{i=1}^{n_b} \left[A(x_i, y_i; w_1) - g_D(x_i, y_i) \right]^2 . \tag{3.7}$$

Definition 3.3.2 (PDE loss).

$$L_p = \sum_{i=1}^{n_p} \left[\Delta u_h(x_i, y_i; w_1, w_2) - f(x_i, y_i) \right]^2 . \tag{3.8}$$

Definition 3.3.3 (Approximation error).

$$err_2 = \sqrt{\frac{1}{m} \sum_{i=1}^{m} |u_h(x_i, y_i; w_1, w_2) - u(x_i, y_i)|^2}. \tag{3.9}$$

The definitions appearing in (3.7) and (3.8) correspond to the sum of square errors minimized by every hidden neuron of the respective network during the training process at each learning iteration using the Adam optimizer.
The error defined by (3.9) measures the difference between the exact analytic solution and the output of the neural network in terms of sum of square errors, and allows us to compare the results obtained with the Finite Element method, introduced in Chapter 1. In two dimensions, it is evaluated on a **test set** taken as a bidimensional regular 50×50 grid on $(0, 1)^2$ (and for the N-dimesnional problems, as the extension of such grid generated by 64 equispaced locations along each coordinate).

3.4. Extensions

In the following section I propose a way for applying the method to the other types of differential equations introduced in Chapter 1. The boundary conditions will always be considered of Dirichlet type, but the extension other to types of bounday conditions is

straightforward.

3.4.1. Steady advection-diffusion equation

Consider a generic stationary PDE, characterised by the differential operator $L : V \longrightarrow \mathbb{R}$:

$$\begin{cases} Lu = f & \text{in } \Omega, \\ u = g_D & \text{on } \partial\Omega. \end{cases}$$

Then, u can be approximated by the same algorithm introduced in the previous sections, made of a boundary and a PDE network, trained alike.
The boundary network remains unchanged: $A(x, y; w_1)\big|_{\partial\Omega} \approx g_D \ \forall (x, y) \in \partial\Omega$, with loss function given by (3.4). The only adjustment is introduced on the PDE network $N(x, y; w_2)$, and specifically in its loss function: in equation (3.5) the term $\Delta N(x_i, y_i; w_2)$ will be substituted by the evaluation of the differential operator L at the output of the network. The PDE loss becomes:

$$\sum_{i=1}^{n_p} \left[f(x_i, y_i) - L(N(x_i, y_i; w_2)) \right]^2 .$$

In particular, in the advection-diffusion case,

$$Lu = -\Delta u + \beta \cdot \nabla u,$$

where $\beta \in \mathbb{R}^2$ is given, and therefore the loss function (3.5) is substituted by the following functional:

$$J_p(\mathbf{x}, \mathbf{y}; w_2) = \sum_{i=1}^{n_p} \left[f(x_i, y_i) + \Delta N(x_i, y_i; w_2) - \beta \cdot \nabla N(x_i, y_i; w_2) \right]^2 .$$

Finally, the approximate solution of the problem is still given by:

$$u_h(x, y; w_1, w_2) = A(x, y; w_1) + B(x, y)N(x, y; w_2),$$

where $B(x, y)$ is a function in Ω, whose restriction to the boundary is zero, i.e. $B : \Omega \to \mathbb{R}$, such that $B\big|_{\partial\Omega} = 0$.

The training algorithm is not modified, except for the expression of the PDE loss function in step 15, as discussed above.

3.4.2. Evolutionary problems

A time-dependent PDE problem can be expressed in generic terms as follows:

$$\begin{cases} u_t - Lu = f & \text{in } \Omega \times (0,T], \\ u = g_D & \text{on } \partial\Omega \times (0,T], \\ u(\mathbf{x},0) = g(\mathbf{x}) & \text{in } \Omega. \end{cases}$$

In addition to the (Dirichlet) boundary condition $u = g_D$ also an initial condition is given, fixing the value of u at time 0.

The corresponding optimization problem, analogous to (3.1) presented in Section 3.1, has an additional constraint due to the initial condition and is consequently modified as follows:

$$\begin{cases} p^* = \text{argmin}_p \; [\hat{u}_t(\mathbf{x},t;p) - L\hat{u}(\mathbf{x},t;p) - f(\mathbf{x},t)]^2 \text{ st} \\ \hat{u} = g_D \quad \text{on } \partial\Omega \times [0,T] \quad \text{and} \\ \hat{u}(\mathbf{x},0;p) = g(\mathbf{x}) \quad \text{in } \Omega. \end{cases}$$

The first constraint can be eliminated by the introduction of an independent neural network for the estimation of the solution on the boundary, while the second one can be incorporated as a penalization in the objective functional. In this way we obtain an unconstrained problem, whose minimization can be performed by the Gradient Descent-based algorithms.

If we simply extend the concept like we did for the steady advection-diffusion problem, we consider the time just as a third input variable for both networks.

The input data for the PDE and boundary networks will be given respectively by $\{x_i, y_i, t_i\}_{i=1}^{n_p}$ and $\{\hat{x}_i, \hat{y}_i, \hat{t}_i\}_{i=1}^{n_b}$, random samples in the domain and on $\partial\Omega$ along the time interval $[0,T]$. As well as the elimination of a discrete triangulation as space discretization of Ω, also no time discretization is introduced on $[0,T]$, like in [87].

We obtain a coupling between the extension of usual boundary network $A(x,y,t;w_1)_{|\partial\Omega} \approx g_D \; \forall (x,y,t) \in \partial\Omega \times [0,T]$, with loss function given by a slight modification of equation (3.4), where also the time variable is taken into consideration:

$$J_b(\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{t}}; w_2) = \sum_{i=1}^{n_b} [(g_D)_i - A(\hat{x}_i, \hat{y}_i, \hat{t}; w_1)]^2, \tag{3.10}$$

and a new PDE network $\hat{N}(x, y, t; w_2)$, whose corresponding loss function is expressed as follows:

$$J_{\hat{N}}(\mathbf{x}, \mathbf{y}, \mathbf{t}; w_2) = \sum_{i=1}^{n_p} \left[g(x_i, y_i) - \hat{N}(x_i, y_i, t_i; w_2) \right]^2 + \\ + \sum_{i=1}^{n_p} \left[f(x_i, y_i, t_i) - \frac{\partial \hat{N}(x_i, y_i, t_i; w_2)}{\partial t} + L(\hat{N}(x_i, y_i, t_i; w_2)) \right]^2. \tag{3.11}$$

The output of this coupling, as an extension of (3.3), is given by:

$$u_h(x, y, t; w_1, w_2) = A(x, y, t; w_1) + B(x, y)\hat{N}(x, y, t; w_2).$$

Notice that the coupling structure has remained unchanged, and so has the training algorithm, except the steps 6 and 13, where the training sets are defined. Indeed, also a sequence of time points $\{\hat{t}_i\}_{i=1}^{n_b}$ must be randomly sampled as input of the boundary network and another one $\{t_i\}_{i=1}^{n_p}$ for the PDE network. Therefore, the training sets become $\{(\hat{x}_i, \hat{y}_i, \hat{t}_i)\}_{i=1}^{n_b}$ on the boundary and $\{(x_i, y_i, t_i)\}_{i=1}^{n_p}$ in $\Omega \times [0, T]$. Moreover, at steps 8 and 15 the functionals to be minimized are respectively given by (3.10) and (3.11).

As a consequence, the overall sum of square errors (3.6), minimized at each training step, is modified as follows:

$$S\hat{S}E(\hat{x}_i, \hat{y}_i, x_i, y_i; w_1, w_2) = \\ = \sum_{i=1}^{n_b} \left[g_D(\hat{x}_i, \hat{y}_i, \hat{t}_i) - u_h(\hat{x}_i, \hat{y}_i, \hat{t}_i; w_1, w_2) \right]^2 + \sum_{i=1}^{n_p} [g(x_i, y_i) - u_h(x_i, y_i, t_i; w_1, w_2)]^2 + \\ + \sum_{i=1}^{n_p} \left[f(x_i, y_i, t_i) - \frac{\partial u_h(x_i, y_i, t_i; w_1, w_2)}{\partial t} + Lu_h(x_i, y_i, t_i; w_1, w_2) \right]^2.$$

Observe that the penalty term related to the initial condition in equation (3.11) is not only considered for $t = 0$ but also for samples corresponding to any time. This allows the definition of a smaller dataset, without the necessity of a pre-training on $\Omega \times \{t = 0\}$ and, as we will see in the experiments in Chapter 6, it does not produce overfitting of the initial condition. A smaller set of input data reduces the computational cost and makes

the method faster.

Finally, the measures of error analyzed for the evolutionary problems are the same as introduced in the Definitions 3.3.1 3.3.2 and 3.3.3, but modified in order to take into account also the variable t and a measure of fitting of the initial condition g. their precise definition is given by:

- **Bounding loss**:

$$L_b = \sum_{i=1}^{n_b} \left[A(x_i, y_i, t_i; w_1) - g_D(x_i, y_i, t_i)\right]^2; \tag{3.12}$$

- **PDE loss**:

$$L_p = \sum_{i=1}^{n_p} \left[\frac{\partial u_h}{\partial t}(x_i, y_i, t_i; w_1, w_2) - \Delta u_h(x_i, y_i, t_i; w_1, w_2) - f(x_i, y_i, t_i)\right]^2 + \\ + \sum_{i=1}^{n_p} \left[u_h(x_i, y_i, t_i; w_1) - g(x_i, y_i)\right]^2; \tag{3.13}$$

- **Approximation error**:

$$err_2 = \sqrt{\frac{1}{m} \sum_{i=1}^{m} |u_h(x_i, y_i, t_i; w_1, w_2) - u(x_i, y_i, t_i)|^2}. \tag{3.14}$$

The first two measures correspond to the loss functions of the boundary and PDE network, respectively, while the last one is comparable with the error in terms of sum of square errors. They will be evaluated again on regular grids, this time of dimension $20 \times 20 \times 20$, except for the test cases involving an exact solution with a particular feature in a restricted region of the domain (test cases TC.P2, TC.P5 and TC.H2 of Chapter 6), where we will consider a $50 \times 50 \times 50$ grid in order to better fit that characteristic.

4 | Numerical results: set up of the test cases

The numerical results are presented first in the simple case of the Poisson equation (1.5), applied to solutions with different levels of regularity and defined on domains with different dimensions. In this framework, we will analyze the effect of variations in some hyperparameters of the network model and its behaviour when the dimension of the domain increases.

Subsequently, the deep learning approach will be extended to the stationary advection-diffusion problem (1.6) in Section 5.2 and finally also evolutionary problems will be considered, both parabolic and hyperbolic (Chapter 6). The heat equation of the form (1.8) and advection-diffusion problem defined in equation (1.6) are presented (see Section 4.2), while the hyperbolic test cases consist of linear transport equations (see Section 4.3).

This chapter introduces the problems that will be analyzed in Chapters 5 and 6. They are built using the method of manufactured solutions [79], that is a way of defining examples of equations having solutions with given characteristics, without any physical meaning, with the only purpose of method verification. We will start from the analytical expression of the desired exact solutions u and the general form of the PDE we want to test, and then deduce the problem data by evaluating the equation at u in order to compute the external source f, the boundary and initial conditions.

The examined solutions will be of three types: a very smooth one (u^{smooth}), a regular solution with a steep peak (u^{peak}) in the centre of the domain and a solution with low regularity ($u^{irregular}$). Their expressions are given by:

$$\begin{cases} u^{smooth}(x,y) = \sin(\pi x)\sin(\pi y) & \text{in } (0,1)^2, \\ u^{peak}(x,y) = e^{-1000(x-0.5)^2-1000(y-0.5)^2} & \text{in } (0,1)^2, \\ u^{irregular}(\rho,\theta) = \rho^{\frac{2}{3}}\sin\left(\frac{2}{3}\theta\right) & \forall \rho \in (0,1) \quad \forall \theta \in \left(0, \frac{\pi}{2}\right). \end{cases}$$

Notice that $u^{irregular}$ is defined in polar coordinates and does not belong to the Sobolev space H^2, since it is analytical in the closure of the domain $\Omega = (0,1)^2$ but its gradient is singular in $(x,y) = (0,0)$.

The time-dependent tests will instead be constructed by starting from solutions that correspond to the exponential evolution in time of the above ones: i.e.

$$\begin{cases} u^{smooth,t}(x,y,t) = u^{smooth}e^{-t} & \text{in } (0,1)^2 \times [0,1], \quad (4.1a) \\ u^{peak,t}(x,y,t) = u^{peak}e^{-t} & \text{in } (0,1)^2 \times [0,1], \quad (4.1b) \\ u^{irregular,t}(x,y,t) = u^{irregular}e^{-t} & \text{in } (0,1)^2 \times [0,1]. \quad (4.1c) \end{cases}$$

In the following sections the different test cases are introduced, while the detailed derivation of the problems data will be reported in the following chapters, together with the experiments results.

We point out that all the functions presented above are at least in H^1. According to the results presented in Chapter 1, we expect the approxiation error in the L^2 norm produced by the Finite Element Galerkin method with a mesh of granularity h to be controlled by h^2 in the Poisson and parabolic case and h when also the transport term is included for all the problems with exact solution also in H^2.

4.1. Elliptic problems

The first type of PDE considered is the elliptic one. In particular, the Poisson equation, only involving second order derivatives, that will be deeply investigated in Section 5.1, and the advection-diffusion equation, derived by adding a transport term to the Poisson problem.

4.1.1. Poisson problem

Three Poisson problems on the domain $\Omega = (0,1)^2$ are analyzed and solved with a deep learning approach in 2 and higher dimensions, each one constructed in such a way that the corresponding solutions have different behaviours. Here we rapidly introduce them in the 2D case.

The first one is

$$\begin{cases} \Delta u = -2\pi^2 \sin(\pi x)\sin(\pi y) & \forall (x,y) \in \ \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases} \tag{TC.E1}$$

and has a very smooth solution.

The second solution presents a peak in the middle of Ω, and the corresponding problem is

$$\begin{cases} \Delta u = [-4000 + 2000^2((x-0.5)^2 + (y-0.5)^2)]e^{-1000(x-0.5)^2 - 1000(y-0.5)^2} & \text{in } \Omega, \\ u(x,y) = e^{-250-1000(y-0.5)^2} & \forall y \in [0,1], \ x = 0 \ \vee \ x = 1, \\ u(x,y) = e^{-250-1000(x-0.5)^2} & \forall x \in [0,1], \ y = 0 \ \vee \ y = 1. \end{cases} \tag{TC.E2}$$

The last example consists of a problem whose solution presents a singularity at $(x,y) = (0,0)$:

$$\begin{cases} \Delta u = f & \text{in } \Omega, \\ u = g_D & \text{on } \partial\Omega, \end{cases} \tag{TC.E3}$$

where f and g_D are chosen so that the exact solution is given, in polar coordinates, by $u(\rho,\theta) = \rho^{2/3}\sin\left(\frac{2}{3}\theta\right)$.

4.1.2. Steady state advection-diffusion problem

The first extension to a more complex problem consists of the steady advection-diffusion defined in equation (1.6) with $\beta = [1,1]^T$. The test cases are defined again on the domain $\Omega = (0,1)^2$ so that their exact solutions are equivalent to the ones of the previously introduced Poisson problems.

In particular, the first example presenting a smooth exact solution reads as:

$$\begin{cases} -\Delta u + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = \\ \qquad = 2\pi^2 \sin(\pi x)\sin(\pi y) - \pi \sin(\pi x)\cos(\pi y) - \pi \cos(\pi x)\sin(\pi y) & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega, \end{cases} \tag{TC.E4}$$

the second one with a peak in the middle of the domain:

$$\begin{cases} -\Delta u(x,y) + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = \\ \qquad = [6000 - 2000^2((x-0.5)^2 + (y-0.5)^2) - 2000(x+y)]\cdot \\ \qquad\qquad \cdot e^{-1000(x-0.5)^2 - 1000(y-0.5)^2} & \text{in } \Omega, \\ u(x,y) = e^{-250-1000(y-0.5)^2} & \forall y \in [0,1],\ x = 0 \ \vee \ x = 1, \\ u(x,y) = e^{-250-1000(x-0.5)^2} & \forall x \in [0,1],\ y = 0 \ \vee \ y = 1, \end{cases} \tag{TC.E5}$$

and the third one involving an irregular behaviour of the gradient is expressed as follows:

$$\begin{cases} -\Delta u + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = f(x,y) & \text{in } \Omega, \\ u = g_D(x,y) & \text{on } \partial\Omega, \end{cases} \tag{TC.E6}$$

where f and g_D are chosen so that the exact solution is given, in polar coordinates, by $u(\rho,\theta) = \rho^{2/3}\sin\left(\frac{2}{3}\theta\right)$.

The goal of this analysis is to find out whether the regularity of the solution affects the accuracy more than the complexity of the equation or viceversa. These tests will only be performed on a bidimensional domain.

4.2. Parabolic problems

As parabolic problems, we will study three heat equations and two time-dependent advection-diffusion equations, with initial solutions with different regularity, given by the solutions of the stationary problems introduced in Sections 2.1 and 2.2. All the following examples are defined in the spatial domain $\Omega = (0,1)^2$ and within the time interval $[0,1]$.

4.2.1. Heat equation

The first parabolic problem is characterized by a smooth initial condition:

$$\begin{cases} u_t - \Delta u = (2\pi^2 - 1)\sin(\pi x)\sin(\pi y)e^{-t} & \text{in } (0,1)^2 \times (0,1], \\ u = 0 & \text{if } x = 0 \ \vee \ x = 1 \ \vee \ y = 0 \ \vee \ y = 1 \text{ in } (0,1], \\ u(x,y,0) = \sin(\pi x)\sin(\pi y) & \forall (x,y) \in (0,1)^2. \end{cases} \tag{TC.P1}$$

The second one presents a peak in the middle of the domain at time 0:

$$\begin{cases} u_t - \Delta u = [3999 - 4 \cdot 10^4((x-0.5)^2 + (y-0.5)^2)] \cdot \\ \qquad \cdot e^{-1000\left[(x-0.5)^2+(y-0.5)^2\right]} e^{-t} & \text{in } (0,1)^2 \times (0,1], \\ u = e^{-[t+250+1000(y-0.5)^2]} & \text{if } x = 0 \ \vee \ x = 1, \ \forall t \in (0,1], \\ u = e^{-[t+250+1000(x-0.5)^2]} & \text{if } y = 0 \ \vee \ y = 1, \ \forall t \in (0,1], \\ u(x,y,0) = e^{-1000\left[(x-0.5)^2+(y-0.5)^2\right]} & \forall (x,y) \in (0,1)^2. \end{cases} \tag{TC.P2}$$

The last initial condition considered for the heat equation examples has a singularity on the boundary. The corresponding problem reads as follows:

$$\begin{cases} u_t - \Delta u = f & \text{in } (0,1)^2 \times [0,1], \\ u = g_D(x,y,t) & \forall t \in (0,1], \ x = 0 \ \vee \ x = 1 \ \vee \ y = 0 \ \vee \ y = 1 \\ u(x,y,0) = g(x,y) & \text{in } \Omega. \end{cases} \tag{TC.P3}$$

where f, g_D and g are chosen so that the exact solution is given, in polar coordinates, by $u(\rho,\theta,t) = \rho^{2/3}\sin(\frac{2}{3}\theta)e^{-t}$.

4.2.2. Time-dependent advection-diffusion equation

The advection-diffusion problem with smooth initial solution is given by the following expression:

$$\begin{cases} u_t - \Delta u + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = (2\pi^2 - 1)\sin(\pi x)\sin(\pi y)e^{-t} + \\ \qquad +\pi \sin(\pi x)\cos(\pi y) + \pi\cos(\pi x)\sin(\pi y)]e^{-t} \qquad \text{in } (0,1)^2 \times (0,1], \\ u = 0 \qquad \text{if } x = 0 \ \vee \ x = 1 \ \vee \ y = 0 \ \vee \ y = 1, \ \forall t \in (0,1], \\ u(x,y,0) = \sin(\pi x)\sin(\pi y) \qquad \forall (x,y) \in (0,1)^2. \end{cases} \tag{TC.P4}$$

The second one presents a peak in the middle of the domain at time 0:

$$\begin{cases} u_t - \Delta u + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = e^{-1000\left[(x-0.5)^2+(y-0.5)^2\right]}e^{-t} \cdot \\ \qquad \cdot [5999 - 2000^2((x-0.5)^2 + (y-0.5)^2) - 2000(x+y)] \\ \qquad \qquad \text{in } (0,1)^2 \times (0,1], \\ u = e^{-(t+250+1000(y-0.5)^2)} \qquad \text{if } x = 0 \ \vee \ x = 1, \ \forall t \in (0,1], \\ u = e^{-(t+250+1000(x-0.5)^2)} \qquad \text{if } y = 0 \ \vee \ y = 1, \ \forall t \in (0,1], \\ u(x,y,0) = e^{-1000\left[(x-0.5)^2+(y-0.5)^2\right]} \qquad \forall (x,y) \in (0,1)^2. \end{cases} \tag{TC.P5}$$

The last initial condition considered for the heat equation examples has a singularity on the boundary. The corresponding problem reads as follows:

4.3. Hyperbolic problems

We will analyze two examples, corresponding to the two continuous initial conditions. Linear advection should not produce discontinuities in the solutions when the initial data are smooth, while it may happen for irregular ones.
Also in this section the domain is $Q_T = \Omega \times [0,T]$ with $\Omega = (0,1)^2$ and $T = 1$.

Two bidimensional linear transport problems are analyzed with $\beta = [1,1]^T$.
We start by analyzing a case with smooth initial condition, expressed as follows:

$$
\begin{cases}
u_t + u_x + u_y = \pi[\cos(\pi x)\sin(\pi y) + \sin(\pi x)\cos(\pi y)]e^{-t} + \\
\qquad\qquad\qquad - \sin(\pi x)\sin(\pi y)e^{-t} \qquad \text{in } (0,1)^2 \times (0,1], \\
u(x,y,t) = 0 \qquad \text{if } x = 0 \ \vee \ x = 1 \ \vee \ y = 0 \ \vee \ y = 1, \ \forall t \in (0,1], \\
u(x,y,0) = \sin(\pi x)\sin(\pi y) \qquad \forall (x,y) \in (0,1)^2.
\end{cases}
\tag{TC.H1}
$$

Then, we move on to an initial condition presenting a steep peak in the middle of Ω:

$$
\begin{cases}
u_t + u_x + u_y = -[2000(x+y) + 2001]e^{-1000[(x-0.5)^2+(y-0.5)^2]}e^{-t} \qquad \text{in } (0,1)^2 \times (0,1], \\
u(x,y,t) = e^{[-t+250+1000(y-0.5)^2]} \qquad \text{if } x = 0 \ \vee \ x = 1, \ \forall y \in [0,1], \ \forall t \in (0,1], \\
u(x,y,t) = e^{[-t+250+1000(x-0.5)^2]} \qquad \text{if } y = 0 \ \vee \ y = 1, \ \forall x \in [0,1], \ \forall t \in (0,1], \\
u(x,y,0) = e^{-1000[(x-0.5)^2+(y-0.5)^2]} \qquad \forall (x,y) \in (0,1)^2.
\end{cases}
\tag{TC.H2}
$$

5 | Numerical results: elliptic problems

The choice of the specific problems to be solved in this section is based on the **method of manufactured solutions**: the equation and boundary data are constructed starting from a corresponding known analytic solution, with different regularity features in the three examples proposed.

The considered solutions are the same for both the diffusion and advection-diffusion, so that the efficiency of the network on the two types of problems can be compared.

The bidimensional problems are defined on the domain $\Omega = (0,1)^2$, and the data given as input to the neural network consist of a 50×50 grid on Ω, constructed by starting from a sampling on the boundary at every iteration. In the most simple case (Poisson problem, Section 6.1) also the extension to generic N-dimensional domains $\Omega = (0,1)^N$, $N > 1$, are presented, since the method is completely mesh free and therefore theoretically apt to higher-dimensional settings.

We first discuss the results obtained by fixing the number of hidden layers to 3, with 256 neurons each, 0.001 learning rate and 1000 training iterations, as suggested by [91], and then in the last parts of Sections 5.1 and 5.2 we try to tune these hyperparameters in order to achieve the best performance. Such final network structures will then be applied in Chapter 6 to the corresponding problems evolving in time.

In the 2-dimensional Poisson cases (TC.E1-TC.E3) also a comparison with the performance of the Finite Element Galerkin method introduced in Section 1.4 is discussed in terms of approximation error and execution time.

5.1. Poisson-Dirichlet problem

In the first part of this section we will discuss the numerical results obtained by fixing the number of layers, neurons and the value of the learning rate as suggested in [91]. Then

we will try to tune these hyperparameters according to the specific problems, so that when the solution is unknown the most suitable model can be applied, according to the boundary data, type of problem, and regularity of the potential initial condition.
In this section we also report the results obtained for different number of data in the training set with the relative approximation errors and execution times. The computational times are measured using the Python package **time** and all the tests are performed on a PC with Windows 10 and 8 GB RAM.

5.1.1. TC.E1: Smooth solution

We solve TC.E1, cf. Chapter 4.
In Figures 5.2a, 5.2b and 5.2c the red surface represents the exact solution, while the green one is the plot of the approximated solution at the first iteration and after 500 and 1000 steps. After 500 iterations we can see that the two graphs are indistinguishable. Indeed, the overall loss behaviour (Figure 5.1a) decreases and reaches 10^{-3} in almost 300 training iterations. Then, the networks coupling starts fluctuating around values of its convergence order 10^{-4}.
The boundary loss defined in (3.7) has the fastest decreasing among the considered error measures (Figure 5.1c), since the condition to be approximated there, i.e. $u = 0$, is extremely simple. Its value reaches almost immediately 10^{-5} and, after a small peak around the 100th iteration, it stabilizes on 3×10^{-6}.
The obstacle that makes the overall function difficult be approximated by the coupling resides in the PDE loss defined in (3.8), that needs the numerical computation of the second order partial derivatives in every direction. Indeed, as we can observe from the plot in Figure 5.1b, it reaches order 10^{-1} in less than 400 training steps and and then presents a lower decrease. Moreover, if on the one hand the bounding loss has fluctuations with amplitude smaller than 10^{-5}, on the other hand, they grow wider in the global output error because of the instability introduced by the PDE network.

Finally, in Table 5.2 containing the performance of the Galerkin Finite Element Method applied to the problem (TC.E1), we can observe that the minimum order reached with the ANN-based method 10^{-4} is attained after 6 refinements of the domain, corresponding to 4225 degrees of freedom. Moreover, the FEM L^2 error strictly depends on the mesh granularity h, as anticipated in Section 1.4, and in particular it is controlled by h^2. On the contrary, the error produced by the ANN-based method does not seem to depend on the number of training points, as we can read in Table 5.1. I have decided to stop the networks training after 400 iterations, that is almost the point where the error plot starts to stabilize around a convergence value in Figure 5.1a, and I found out that with only

60 coordinate points the machine-learning-based method can attain the same accuracy as the FEM with a mesh of 4225 points in a comparable time.
The number of degrees of freedom in the Finite Element Galerkin method considered is equivalent to the number of internal mesh nodes, since on the boundary ones the Dirichlet condition is imposed exactly, while in the machine-learning-based method it is equivalent to the total number of training points of both networks. As a reference, the number of degrees of freedom corresponding to 64 training points for the PDE network is 256.

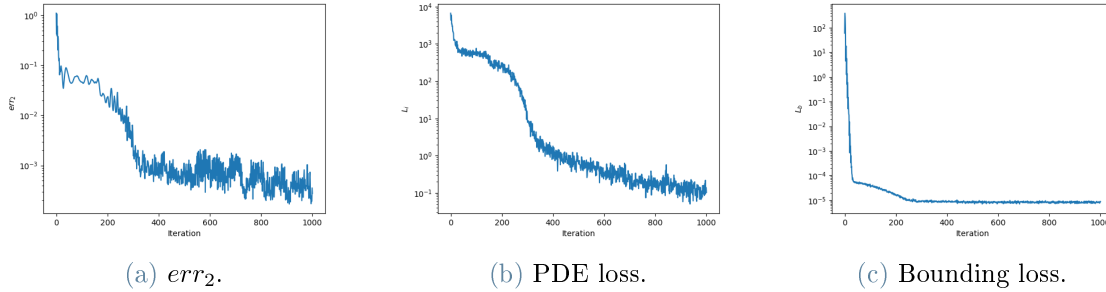
On the other hand, the execution time of the Finite Element Method is much less dependent on the mesh granularity than the ANN-based one and, since the error is controlled by h, it can be decreased up to much lower values. For intance, in the same time needed by the ANN-based method for processing 4225 data and reaching an error of order 10^{-4}, several mesh refinements can be performed in order to make the FEM error gain much lower values.

Degrees of freedom	**Number of training points**	err_2	**Execution time**
80	16	1.6250e-03	1.5883+01 seconds
300	60	5.5904e-04	2.9499e+01 seconds
1100	220	3.2004e-04	8.6556e+01 seconds
4225	845	3.9255e-04	2.5783e+02 seconds

Table 5.1: TC.E1: Performance of the ANN-based method stopped after 400 training iterations.

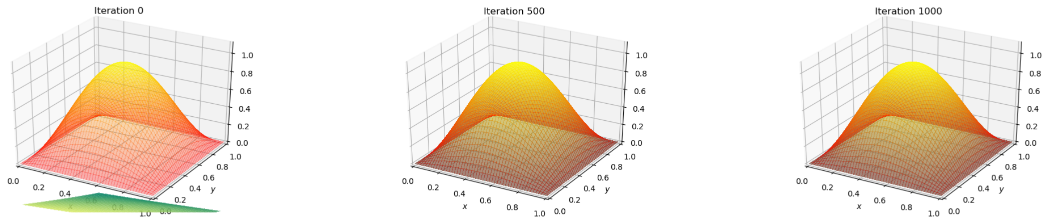
Degrees of freedom	**Mesh granularity**	L^2 **error**	**Execution time**
81	8.8388e-02	2.2564e-02	5.0286e+00 seconds
289	4.4194e-02	5.7270e-03	9.2943e+00 second
1089	2.2097e-02	1.4373e-03	3.1812e+01 seconds
4225	1.1048e-02	3.5967e-04	3.7323e+01 seconds

Table 5.2: TC.E1: Performance the of Finite Element Galerkin method.



(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.1: TC.E1: Measures of error as a function of the iteration counts; networks with 3 layers, 256 neurons and $\eta = 0.001$. The approximation error (5.1a) has an initially fast decrease and then keeps oscillating around 10^{-3} - 10^{-4}. The PDE loss (5.1b) first decreases very rapidly and then slows down, reaching 10^{-1} within 1000 iterations. The bounding loss (5.1c) converges very fast to 10^{-5}.



(a) Computed and exact solutions at iteration 0. (b) Computed and exact solutions after 500 iterations. (c) Computed ans exact solutions after 1000 iterations.

Figure 5.2: TC.E1: Computed (green) and exact (orange) solutions after 0, 500 and 1000 training iterations; networks with 3 layers, 256 neurons and $\eta = 0.001$. The initial guess (5.2a) is almost flat and very different from the exact solution, but the two almost coincide after 400 iterations (5.2b) and at the end of training (5.2c).

5.1.2. TC.E2: Solution with a peak

We solve TC.E2, cf. Chapter 4.

In this case convergence is not attained in 1000 iterations: after a fast initial decrease towards almost 10^{-2}, the error increases and starts oscillating around 10^{-1} until the end of training (see Figure 5.3). The reason for this behaviour is that, even if the approximate solution converges on the boundary, the same does not happen in the inner domain, where the PDE loss does not even present an overall decreasing trend, aside from the instability. This is due to the dramatic change of the exact solution near the peak, where it suddenly

becomes very large, compared to its value in the other points, making the function too complex for the approximation with this relatively simple network configuration. Indeed, we can observe in Figure 5.4 that not only does the approximate curve not fit the peak, but it also presents an opposite curvature to the expected one. This problem is not overcome by increasing the number of input data, as we can read from Table 5.3: indeed, the approximation error does not seem to decrease with respect to the number of samples given as input, that only noticeably increases the exectution time.
In Table 5.4 I have reported the value of err_2 reached after 400 iterations when the data are not randomly sampled allover Ω but instead gathered around the peak region. We can notice that this does not really improve the performance of the method, even when increasing the number of coordinate points given as input. The Galerkin Finite Element method gains better accuracy than the ANN-based one in this case for every mesh with granularity of order at least 10^{-2}, requiring a very low computational cost.

Degrees of freedom	**Number of training points**	err_2	**Execution time**
1100	220	1.5225e-01	9.5856e+01 seconds
4225	845	1.5136e-01	3.2192e+02 seconds
16640	3328	1.5850e-01	1.1151e+03 seconds

Table 5.3: TC.E2: Performance of ANN-based method on a regular grid, stopped after 400 training iterations.

Degrees of freedom	**Number of training points**	err_2	**Execution time**
1100	220	8.6977e-02	7.9462e+01 seconds
4225	845	1.5895e-01	3.2724e+02 seconds
16640	3328	1.8727e-01	1.0621e+03 seconds

Table 5.4: TC.E2: Performance of the ANN-based method stopped after 400 training iterations, data gathered around the peak.

Degrees of freedom	Mesh granularity	L^2 error	Execution time
1089	2.2097e-02	1.0648e-02	3.0334e+01 seconds
4225	1.1048e-02	3.4479e-03	2.2489e+01 seconds
16641	5.5242e-03	9.1166e-04	8.5010e+01 seconds

Table 5.5: TC.E2: Performance of the Galerkin Finite Element method.

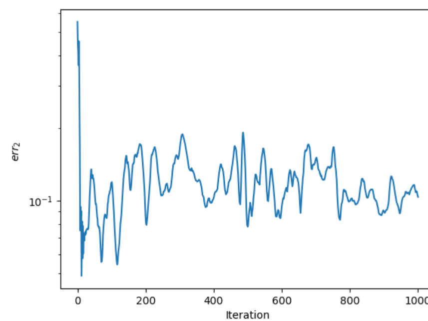
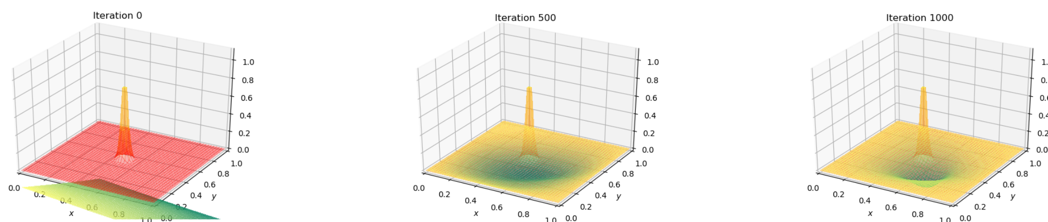


Figure 5.3: TC.E2: Measure of error as a function of the iteration counts; networks with 3 layers, 256 neurons and $\eta = 0.001$. After few iterations the minimum value is achieved by the error err_2, but then the curve starts to show an increasing trend and then settles around 10^{-1} for the first 1000 training iterations.



(a) Computed and exact solutions after 0 iterations.

(b) Computed and exact solutions after 500 iterations.

(c) Computed and exact solutions after 1000 iterations.

Figure 5.4: TC.E2: Computed (green) and exact (orange) solutions after 0, 500 and 1000 training iterations; networks with 3 layers, 256 neurons and $\eta = 0.001$. The two plots after 500 iterations (5.4b) almost coincide on the boundary but in the interior of the domain and especially near the peak they have very different shapes and the approximation is opposite to the exact plot.

If we sum a smooth function to the considered solution as follows:

$$\tilde{u}(x,y) = u(x,y) + v(x,y), \qquad \text{in } \Omega \tag{5.1}$$

with

$$v(x,y) = \sin(\pi x),$$

then the peak is softened, so that the variation in the Laplacian is less steep, and therefore we expect the region around it to be better approximated.
We expect the method to perform better since the peak is no more as steep as before. However, this does not happen. Indeed, the approximate and exact solutions are still very different even after 1000 iterations (Figure 5.4) and the overall sum of square errors (Figure 5.5) is increasing towards 10^1.

The issue presented by TC.E2 is not overcome by the smooth correction in the exact solution because it relies on the dimension of the neighbourhood of the peak, where the derivatives start to vary. The critical region is indeed very small and, having chosen as test set a random group of coordinate points, not enough points are sampled in there for this peak to be correctly learned. However, selecting too many input positions gathered in this narrow area may lead to an overfitting of the corresponding high values taken by the solution, that results in a worse estimation of the function over the remaining part of the domain. As we have previously observed, this does not produce an improvement in the approximation accuracy.

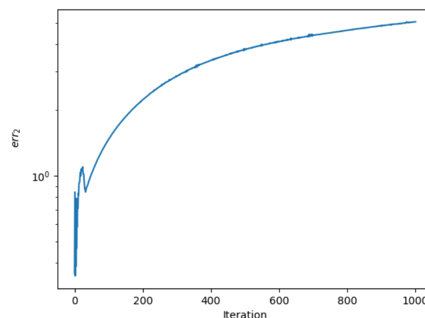
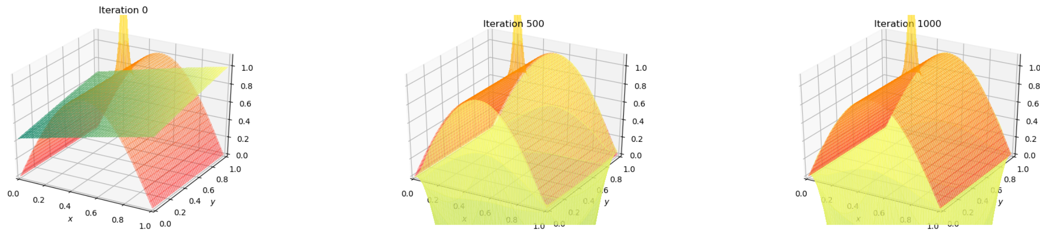


Figure 5.5: TC.E2 with smooth correction: Approximation error as a function of the iteration counts; networks with 3 layers, 256 neurons and $\eta = 0.001$. An initial decreasing trend of the approximation error is followed by an increase towards the value 10^1.



(a) Computed and exact solutions after 0 iterations.

(b) Computed and exact solutions after 500 iterations.

(c) Computed and exact solutions after 1000 iterations.

Figure 5.6: TC.E2 with smooth correction: Computed (green) and exact (orange) solutions after 0, 500 and 1000 iterations; networks with 3 layers, 256 neurons and $\eta = 0.001$. The initial guess (5.6a) is almost flat, then after 500 (5.6b) and 1000 (5.6c) iterations the plots have opposite curvature.

5.1.3. TC.E3: Solution with low regularity

As final and more delicate example, we choose the setting of TC.E3, cf. Chapter 4. We recall that in this case the gradient $\nabla u(\cdot, \cdot)$ presents a singularity on the boundary point $(x, y) = (0, 0)$.
As a consequence, we can observe in Figure 5.8, where the red surface represents the exact solution and the green one is the plot of its approximation after 0, 500 and 1000 learning steps, that the method has difficulties in estimating the solution for $x \approx 0$ and $y \approx 0$ and in general on the boundary since this singularity complicates the Dirichlet condition and slows down the convergence of the boundary network. The error err_2 presents an overall decreasing trend and attains values of order 10^{-3} in less than 200 iterations, but presents very high fluctuations during all the training process (Figure 5.7). Finally, the low regularity introduces an obstacle in the learning process of the PDE network as well, since the function is not in $H^2(\Omega)$ and thus its derivatives, computed for the optimization of the PDE loss, are not smooth.

All in all, the global convergence rate obtained by the application of the proposed deep learning method to a problem whose analytical solution presents a singularity on the boundary is not very distant from the one achieved for smooth solution (TC.E1, Section 5.1.1) for the Poisson problem.

Finally, in this case, as we can observe from the results in Table 5.6, the error err_2 is of order 10^{-3} for every consider number of input data, and does not seem to depend on the input dimensionality, while the L^2 error given by the Galerkin Finite Element method

is decreasing with the mesh granularity and attains order 10^{-4} with only 289 degrees of freedom in very little time.

Degrees of freedom	Number of training points	err_2	Execution time
300	60	3.7942e-03	2.5882e+01 seconds
1100	220	4.2510e-03	8.8374e+01 seconds
4225	845	7.0128e-03	3.0840e+02 seconds
16640	3328	1.9899e-03	1.2557e+03 seconds

Table 5.6: TC.E3: Performance of the ANN-based method stopped after 400 training iterations.

Degrees of freedom	Mesh granularity	L^2 error	Execution time
289	4.4194e-02	6.4676e-04	5.3279e+00 second
1089	2.2097e-02	2.0725e-04	1.0514e+01 seconds
4225	1.1048e-02	6.5980e-05	4.1542e+01 seconds
16641	5.5242e-03	2.0921e-05	2.2249e+02 seconds

Table 5.7: TC.E3: Performance the of Finite Element Galerkin method.

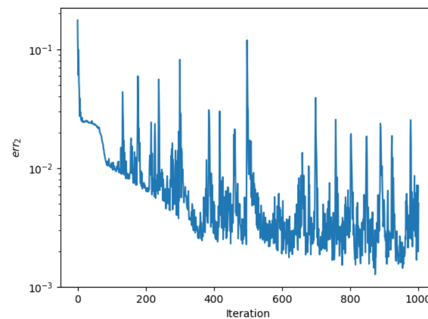
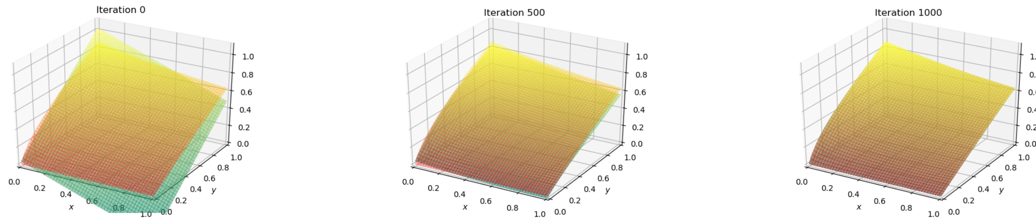


Figure 5.7: TC.E3: Measure of error as a function of the iteration counts; networks with 3 layers, 256 neurons and $\eta = 0.001$. The approximation error err_2 presents an overall decreasing trend towards values of order 10^{-3} but also very high oscillations.



(a) Computed and exact solutions after 0 iterations.

(b) Computed and exact solutions after 500 iterations.

(c) Computed and exact solutions after 1000 iterations.

Figure 5.8: TC.E3: Computed (green) and exact (orange) after 0, 500 and 1000 iterations; networks with 3 layers, 256 neurons and $\eta = 0.001$. The plots almost coincide after 1000 iterations (5.8c).

5.1.4. Tuning of the hyperparameters

In this section we will analyze the relation between the performance of the method proposed in Section 3 and the hyperparameters of the networks. I have decided to assign to both structures the same value for every hyperparameter firstly because in general, as we will see later on, the behaviour of the overall coupling and the of the single learning system is affected in a similar way by the choice. Moreover, even if in some cases only one of the two independent loss functions seems to benefit of a modification in the hyperparameters while the other is indifferent or even slightly disadvantaged by it, the global stability seems to be negatively influenced by an asymmetry in the networks structure.

We will start by varying the number of hidden layers from 1 to 3, maintaining the quantity of neurons they contain and the learning rate of the optimization algorithm to 256 and 0.001 respectively, as in the previously considered tests.
Afterwards, keeping the quantity of layers fixed to the optimal according to our empirical analysis, we will study the impact of the number of neurons per layer and finally we will tune the best value for the learning rate of each network.
All the following observations are based on 20000 training iterations per networks, so that we are able to assess the possible asymptotic trend of the method.

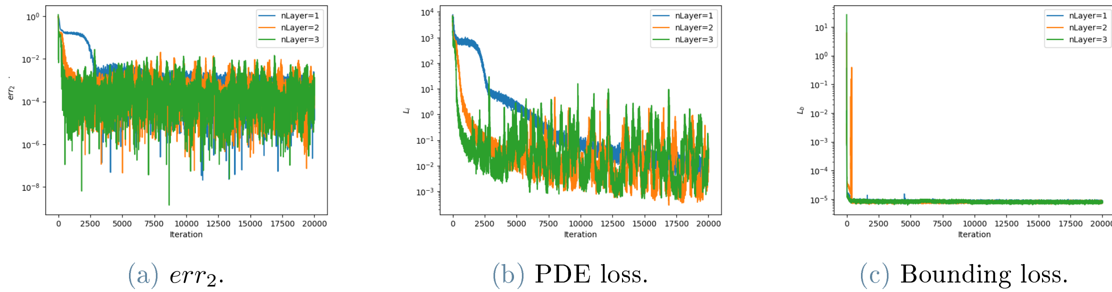
Figures 5.9 - 5.12 show the computed errors of the four experiments TC.E1-TC.E3 (including the test case TC.E2 with smooth correction) when varying the number of hidden layers. The blue lines correspond to the coupling with 1, the orange ones with to 2 and the green ones to 3 hidden layers. The most evident trend we notice from these images is that increasing the value of this hyperparameter leads to a faster method: both loss

functions reach their minimum in less iterations. However, the wide oscillations presented by the interior loss of test TC.E2 with smooth correction and TC.E3 (reported in Figures 5.11 and 5.12, respectively) is not reduced. In both cases, the error err_2 does not present a decreasing trend and the networks loss functions oscillate a lot, even if the bounding ones rapidly achieve order 10^{-4} (see Figures 5.11c and 5.12c). The fluctuations seem to be reduced by an increasing number of hidden layers, even if the PDE loss functions (Figures 5.11b and 5.12b) increase after a few thousands of iterations. This does not allow, as a consequence, convergence of the method for TC.E2 and leads to almost constant approximation error of order 10^0 (Figure 5.11a). The results in terms of minimization of the objective functions are better in the problem with smooth exact solution (Figures 5.9b and 5.9c), even if in the interior of Ω the lowest error is of order 10^{-3}. An improvement is also registered in the performance of both networks in TC.E2, where the bounding loss (Figure 5.10c) reaches almost immediately convergence to a value of order 10^{-6} when the number of layers is increased, and the PDE network, although it does settle on values of order 10^{-5} instead of 10^{-6} like when only one hidden layer is used, is much more stable with 3 layers, except for a single isolated peak around the 12000th training step (see Figure 5.10b). As to the approximation error, the best result is achieved by the problem with smooth analytical solution (Figure 5.9a), whose approximation converges in very few iterations, with oscillations always between small values of order 10^{-3} and 10^{-6}. Similarly, the problem with singularity presents an approximation error that fluctuates between 10^{-2} and 10^{-5} (Figure 5.9a).

All in all, Figure 5.9b shows a noticeable improvement in the learning speed of the coupled model applied to TC.E1 when the number of layers increases, and a subsequent lower order of the PDE loss. When passing from 1 to 2 layers some oscillations arise, and they become wider when using 3 layers. This problem might be however overcome by tuning correctly the other hyperparameters. On the other hand, in Figure 5.9c we can observe that not only the boundary network learns faster but also it becomes more stable when the number of layers increase. Indeed, up to 2 layers some initial oscillations are present, while they disappear adding one more layer.
TC.E2 never reaches convergence to the desired order, but both losses reach lower values when the number of layers is set to 3 (see Figures 5.10b and 5.10c). Moreover, the PDE loss shows a significant decrease when passing from 2 to 3 layers.
Finally, the 2-dimensional problem with irregular solution (TC.E3) both networks are maximally stable when composed of 3 layers.

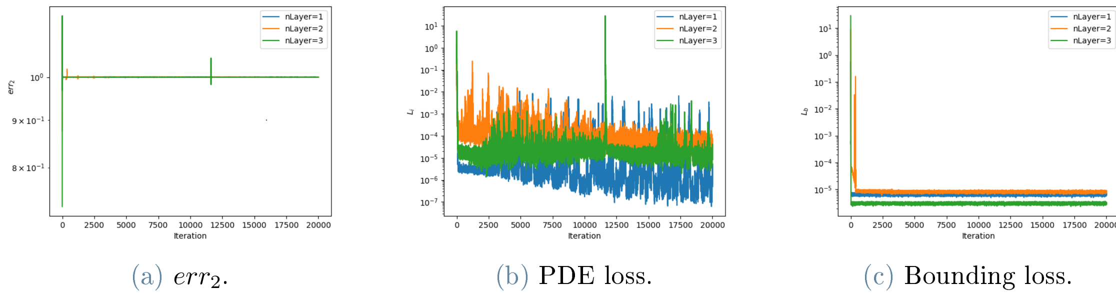
I have observed that the deeper the networks become, the higher the training execution time is, but here we are looking for a fast and intuitive black-box approach, so the

computational time has to be taken into account.



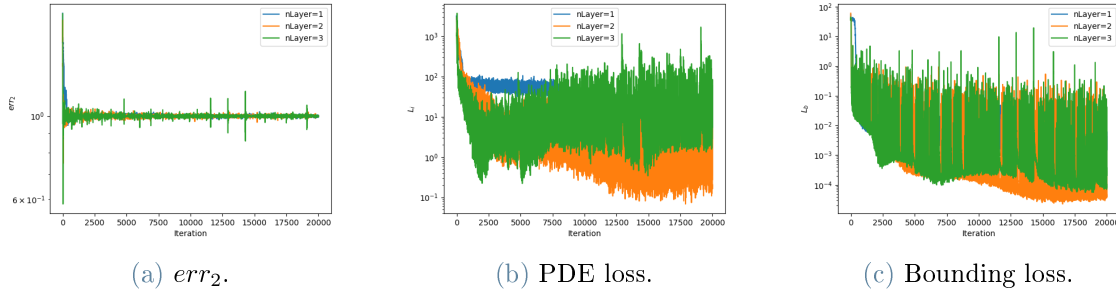
(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.9: TC.E1: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3; networks with 256 neurons per layer and $\eta = 0.001$. Convergence is attained by the error err_2 (5.9a) with any number of layers, oscillating around values of order 10^{-5}. The interior loss (5.9b) always reaches values of order 10^{-3} but with 2 and 3 layers it happens faster; the fluctuations increase with the number of layers. The bounding loss (5.9c) rapidly converges to values of order 10^{-6} in every case.



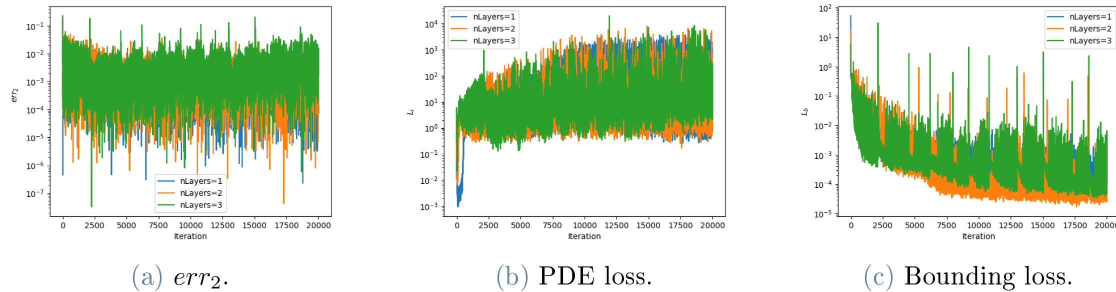
(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.10: TC.E2: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3; networks with 256 neurons per layer and $\eta = 0.001$. The approximation error (5.10a) is constant and of order 10^0 with some oscillations that get wider with the number of layers. The behaviour of the interior (5.10b) and bounding (5.10c) is not generally affected by the number of layers but with 3 layers they both reach values of lower order.



(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.11: TC.E2 with smooth correction: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3; networks with 256 neurons per layer and $\eta = 0.001$. The approximation error (5.11a) is almost constant of value 10^0. The interior loss (5.11b) has high oscillations that get worse with the number of layers and presents an increasing trend with 3 layers. Wide oscillations arise almost immediately for any number of layers in the bounding loss (5.11c).



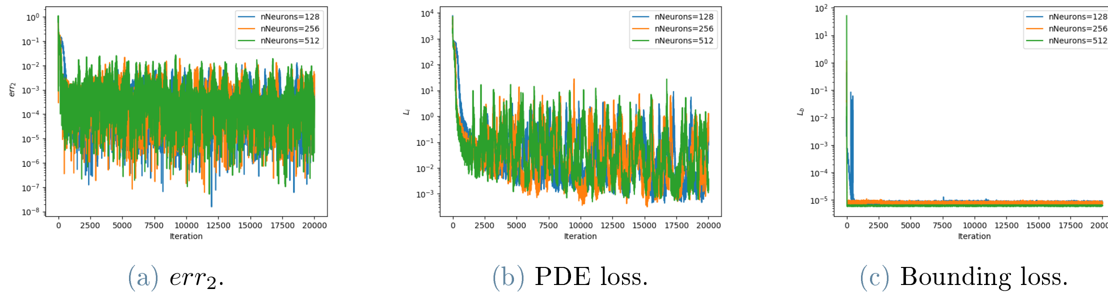
(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.12: TC.E3: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3; networks with 256 neurons per layer and $\eta = 0.001$. All the graphs present very high oscillations for any number of layers and the PDE loss 5.12b increases and fluctuates widely after almost 1000 iterations.

Let us now analyze the impact of the number of neurons per layer. Since we have noticed that no significant improvement is introduced in the test case with peak and smooth correction with respect to the original one, we will focus only on TC.E1, TC.E2 and TC.E3.

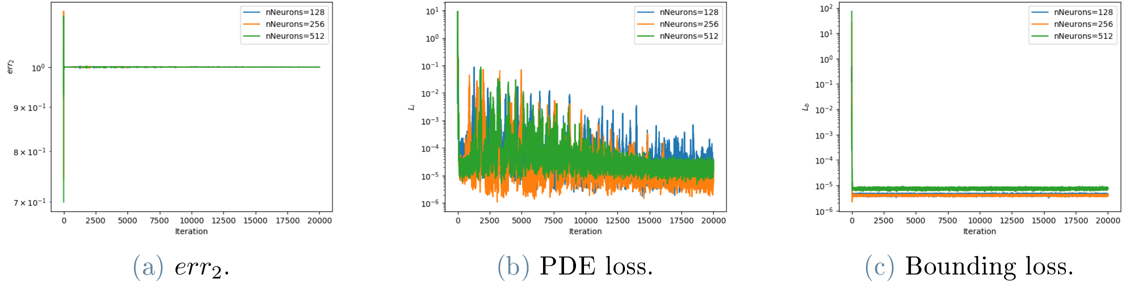
The networks learn faster and the oscillations presented by the plot of the boundary loss disappear when increasing the number of neurons. However, the results are not as evident as for the other hyperparameters, so it seems sufficient to keep the number of neurons per layer of both networks fixed to 256.

As for TC.E2, Figure 5.14a shows once again a constant behaviour of the approximation error corresponding to any choice of the hyperparameters. However, the PDE network becomes more stable when the number of neurons per layer increases, even if the loss order remains between 10^{-4} and 10^{-5} and does not decrease, as displayed in Figure 5.14b. Figure 5.14c shows instead that the convergence value of the boundary loss is higher in the case of 512 neurons, and no oscillations are presented even with 128. Therefore, in this case this hyperparameter could be reduced to 128. On the other hand, the PDE network can become a bit more stable when passing to 512 neurons per layer, but implying a significant increase in the computational time, thus also in this case it seems that 256 is a good compromise.
The graphs related to TC.E3 (Figure 5.15) show the same dependence of the error on the number of neurons per layer and on the number of layers. However, this hyperparamter shall be set to the intermediate value 256, that keeps a decreasing trend of the bounding loss and also avoids wide oscillations in the approximation error err_2.



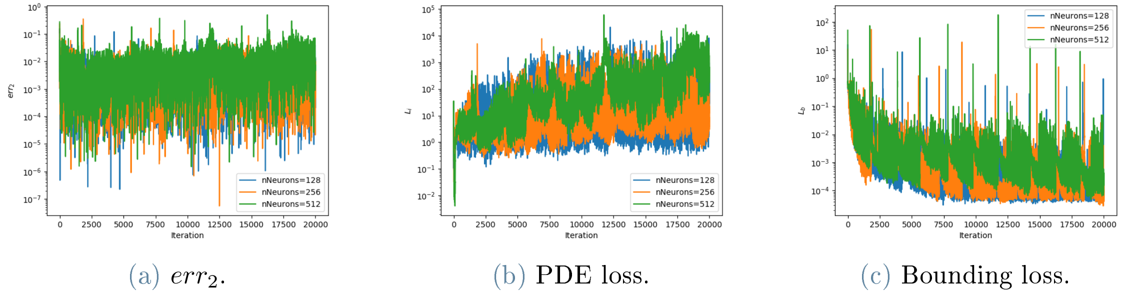
(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.13: TC.E1: Measures of error as a function of the iteration counts when varying the number of neurons; networks with 256 neurons per layer and $\eta = 0.001$. The approximation error (5.13a) oscillates around values of order 10^{-4} and the fluctuations do not change their amplitude with the number of neurons per layer. The interior loss (5.13b) decreases faster when increasing the number of neurons and reaches 10^{-2} in less than 2500 iterations in any case with the same amplitude of the oscillations. The bounding loss converges to 10^{-5} in very few iterations.



(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.14: TC.E2: Measures of error as a function of the iteration counts when varying the number of neurons; networks with 256 neurons per layer and $\eta = 0.001$. The approximation error (5.14a) is constant and of order 10^0. The oscillations of the interior loss (5.14b) become less wide when increasing the number of neurons, especially in the last iterations where the loss fluctuate between 10^{-4} and 10^{-6}.



(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.15: TC.E3: Measures of error as a function of the iteration counts when varying the number of neurons; networks with 256 neurons per layer and $\eta = 0.001$. All the plots present very huge oscillations for any number of neurons per layer, the PDE loss (5.12b) never presents a decreasing trend and the error (5.12a) oscillaes between 10^{-1} and 10^{-5} for every number of neurons per layer.

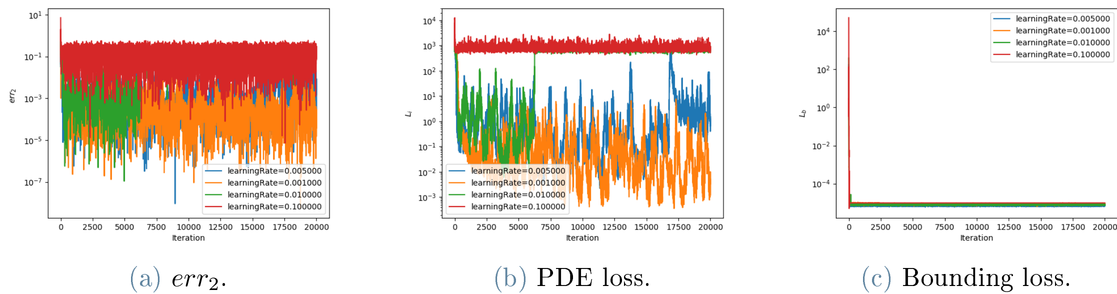
Figures 5.16-5.18 show the effect of modifying the learning rate of the two networks on the three usual error metrics introduced in Section 3.4.

We can immediately observe that an excessively high learning rate does in general produce instability and may even increase the loss order.
However, simple boundary conditions as the ones imposed for TC.E1 and TC.E2 are well approximated and highlight no additional instabilities when the learning rate increases. Indeed, the loss value in the peak case (Figure 5.17c) is lower with $\eta = 0.1$ than all the other cases and its plot converges to a value between 10^{-5} and 10^{-6} without any

oscillation. In the smooth case (TC.E1), the loss function value is almost equivalent for any chosen learning rate, but again no fluctuations arise even with a high value of η, as shown in Figure 5.16c.
On the other hand, when the boundary condition to be approximated is as in TC.E3, an increase in the learning rate creates wider oscillations and a marked increase in the loss order. Indeed, Figure 5.18c reports a $10^1 - 10^3$ boundary loss corresponding to $\eta = 0.1$ (blue line) and, after an initial decrease un to 10^{-2}, fluctuations between 10^{-1} and 10^1 when $\eta = 0.01$ (orange line) for the first 2500 iterations, before stabilizing on 10^1. For $\eta = 0.01$ and $\eta = 0.1$, moreover, the PDE network is visibly untable and the corresponding loss function presents values of order $10^{-10} - 10^5$ and $10^{-16} - 10^2$, respectively. However, when η varies from $\eta = 0.001$ to $\eta = 0.0001$ the learning process does not become slower, but never presents a decreasing trend and settles around 10^0 after few iterations.

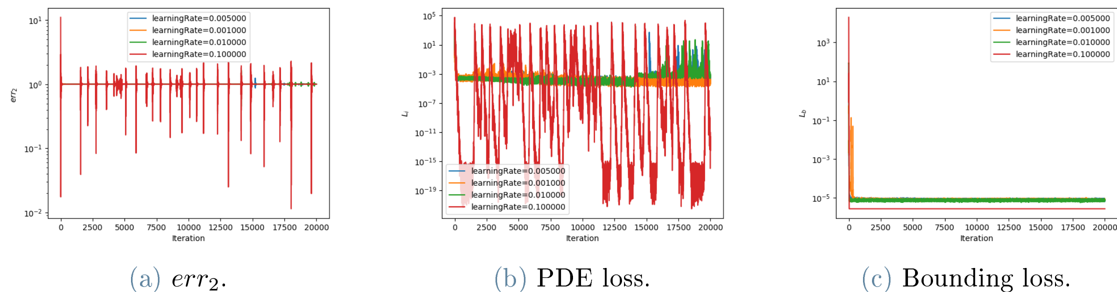
Ideally, we would expect a faster learning every-time we increase the value of η, since we do not change the optimization direction in our algorithm but simply accelerate the procedure. However, it can happen that it misses the local minimum and kind of initializes a new application of the method that brings to another stationary point, that may correspond to higher value of the loss functions. This is likely the case of the PDE networks of TC.E1 and TC.E2, whose bounding loss is represented in Figures 5.16b and 5.17b, respectively. In these pictures the green line, corresponding to $\eta = 0.01$, initially oscillates around 10^{-4} and suddenly jumps to 10^{-2} after 15000 iterations. In the first case, it goes from $10^{-1} - 10^1$ to 10^3, but gaining stability, while in the second one it is stable around 10^{-3} in the beginning and starts oscillating up to 10^1 after 12500 iterations.
Figure 5.18b presents an increasing trend corresponding to $\eta = 0.01$. Even if in this example the plot obtained with $\eta = 0.1$ seems to present a better behaviour, the loss value is much higher than the one around which the orange plot corresponding to $\eta = 0.001$ oscillates.
Another issue that can arise from a too large learning rate is instability of the network, as shown in TC.E2, cf. Figure 5.17b, with very large oscillations between 10^{-19} and 10^5. In this case the excessive learning rate makes the algorithm jump around a local minimum, without the possibility of reaching it. Indeed, the direction of the gradient is the fastest optimization direction, but if the steps are too large, from an iteration to the other, it may point towards a maximization of the loss function, and then again a minimization, missing the stationary points. For this PDE network the best learning rate seems to be $\eta = 0.001$ (corresponding to the orange line in Figure 5.17).
In TC.E1, as we have already pointed out above, a too high learning rate worsens the performance of the network. Therefore, even if wide oscillations are sill present, according

to Figure 5.16b, the best learning rate seems to be $\eta = 0.001$.
Finally, since in the first two examples (TC.E1 and TC.E2) the PDE network is the most complex and therefore the interior loss is the predominant component of the overall error, also the graphs of the the error err_2 follow the same trend. In particular, we can observe a higher order of the global error for TC.E1 in Figure 5.16a and wide oscillations for TC.E2 in Figure 5.17a. As for TC.E3, also the approximation error seems to present a more stable behaviour and lower order of the global error when both networks have the chosen learning rates (green plot in Figure 5.18a). In this last case, an improvement in stability is also given by a further reduction of the hyperparameter to $\eta = 0.0001$.



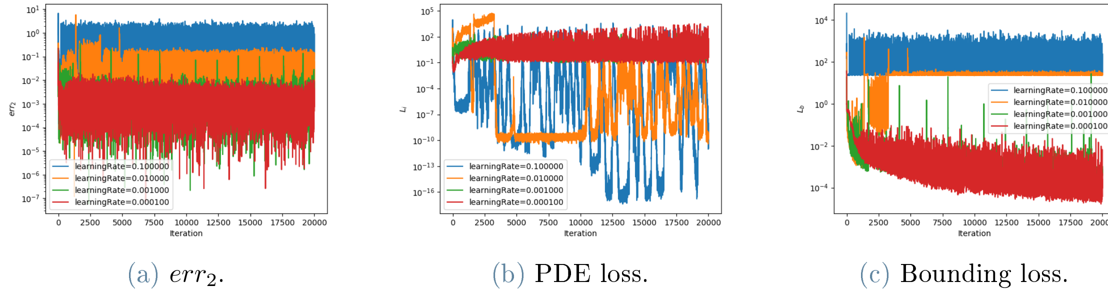
(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.16: TC.E1: Measures of error as a function of the iteration counts when varying the learning rate; networks with 3 layers and 256 neurons each. Convergence is not attained by the error err_2 (5.16a) for $\eta = 0.1$ and the interior loss (5.16b) never decreases below 10^3. For $\eta = 0.01$ both the err_2 and PDE loss plots present a sudden increase to a much higher value. For $\eta \leq 0.005$ the interior loss oscillates around 10^{-3}



(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.17: TC.E2: Measures of error as a function of the iteration counts when varying the learning rate; networks with 3 layers and 256 neurons each. Convergence is never attained by the approximation error (5.17a) and presents wide oscillations around this value between 10^{-2} and 10^2 for $\eta = 0.1$. For every learning rate the interior loss (5.17b) is around 10^{-3}, except for $\eta = 0.1$, that causes oscillations between 10^{-19} and 10^5.



(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.18: TC.E3: Measures of error as a function of the iteration counts when varying the learning rate; networks with 3 layers and 256 neurons each. For $\eta = 0.1$ the oscillations in the interior loss (5.18b) are the most narrow but it has an increasing behaviour and reaches the order 10^7. The bounding loss (5.18c) is greater than 10^2 for $\eta = 0.1$, and presents very wide oscillations between 10^{-1} and 10^1 for $\eta = 0.01$. For $\eta = 0.001$ and $\eta = 0.0001$ the approximation error 5.18a is very stable but does not present a decreasing trend anyway.

In conclusion, we have found out that the best values for the hyperparameters seem to be the ones applied in the previous section, thus in the following examples we will start the analysis by setting them as above. Only the test case with singularity requires a lower learning rate. Indeed, as we can observe in Figure 5.19, the method is more stable and the approximation error is still decreasing and reaches order 10^{-3} in 1000 iterations.

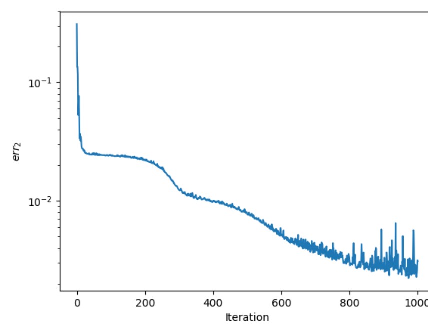


Figure 5.19: TC.E3 with tuned hyperparameters: Measures of error as a function of the iteration counts; networks with 3 layers, 256 neurons and $\eta = 0.0001$.

5.1.5. Extension to N-dimensional elliptic PDEs

In this section we will repeat the experiments for analogue test cases but in higher-dimensional spaces. Here we vary the number of hidden layers between 1, 2 and 3.

For the definition of the test cases the method of manufactured solutions is again used, applied to the N-dimensional version of the first two examples presented in the previous sections and different test case with irregular solution substituting TC.E3, that cannot be extended to higher-dimensional spaces since it is defined in polar coordinates:

- **TC.E1b (Smooth solution)**:

$$u(\mathbf{x}) = \prod_{k=1}^{N} \sin(\pi x_k) \tag{TC.E1b}$$

- **TC.E2b (Solution with a peak)**:

$$u(\mathbf{x}) = e^{-1000 \sum_{K=1}^{N} (x_k - 0.5)^2} \tag{TC.E2b}$$

- **TC.E3b (Solution with low regularity)**:

$$u(\mathbf{x}) = {x_2}^{0.6} \tag{TC.E3b}$$

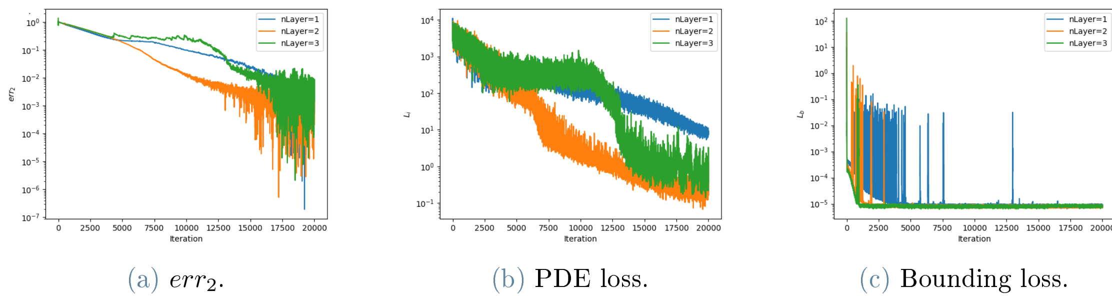
Since this method is mesh-free, it can virtually be applied to spaces of any dimension, higher than 2 and 3. Figures 5.20 - 5.22 represent all the errors measured in the numerical examples discussed above, obtained by applying neural networks with 1, 2 and 3 hidden layers and 256 neurons per layer in the 5-dimensional domain $\Omega = (0,1)^5$.

The convergence speed always increases with the number of layers, and the behaviour of the bounding and PDE loss is almost the same for the peak and singularity cases.

The solution with peak (TC.E2b) still reaches the threshold 10^{-5} in less iterations when 2 or 3 layers are considered, and the oscillations in the boundary loss are flattened by the usage of 3 layers. The overall approximation error err_2 does not converge and is still constant and of order 10^0 (see Figures 5.21a, 5.21c and 5.21b).

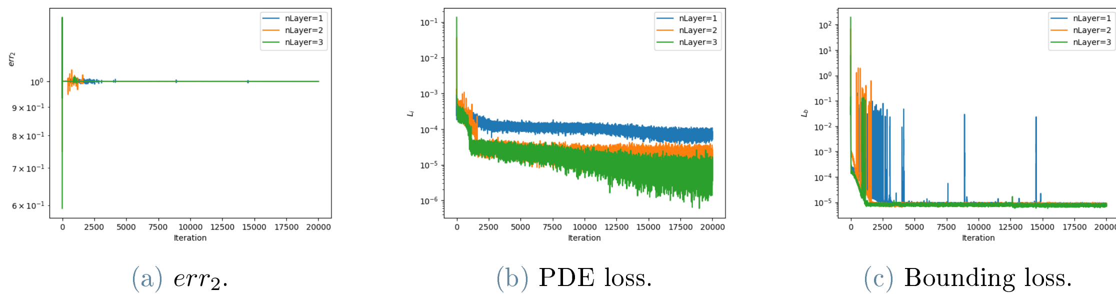
The oscillations in the PDE loss with solution with a singularity near y=0 (TC.E3b) become too wide and vary between 10^0 and 10^4, so the approximation error has an irregular behaviour and is never stably reduced under 10^{-2}. The best results are still obtained in the smooth case (TC.E1b), where three hidden layers minimize the boundary loss function

to the order 10^{-5} with no oscillations in less than 2000 iterations and the PDE loss has a decreasing trend, even if the minimum order reached within 20000 iterations is 10^{-1}, much higher than the 2D case (see Figures 5.20b and 5.20c). This leads to a decreasing approximation error, reaching the value 10^{-2}, in 10000 and 15000 iterations with 2 and 3 hidden layers respectively (plot in Figure 5.20a). After getting to an error of order 10^{-2} in a 5-dimensional space also the approximation of the smooth solution begins presenting oscillations without stabilizing around the threshold 10^{-5}.



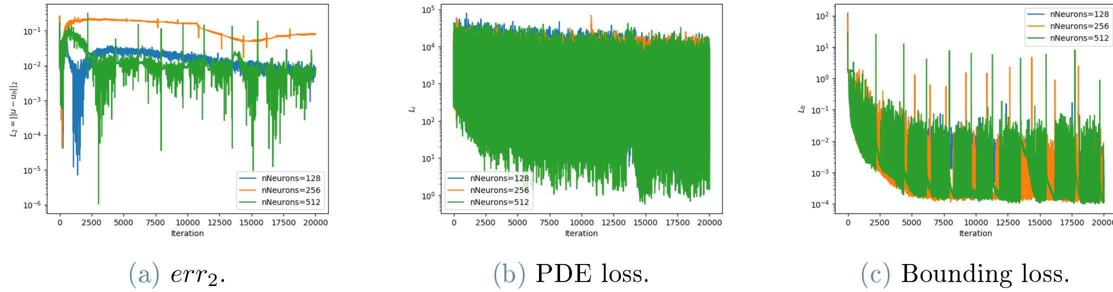
(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.20: TC.E1b: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3; networks with 256 neurons per layer and $\eta = 0.001$. After reaching values of order 10^{-2} the approximation error starts to oscillate, but unlike the 2D case fluctuations are tighter when the number of layers increase.



(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.21: TC.E2b: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3; networks with 256 neurons per layer and $\eta = 0.001$. The approximation error (5.21a) is constant and of order 10^0 with some oscillations that get wider with the number of layers. The behaviour of the interior (5.21b) and bounding (5.21c) is not generally affected by the number of layers, except that they both reach values of lower order when it increases.



(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.22: TC.E3b: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3; networks with 256 neurons per layer and $\eta = 0.001$. All the graphs present very huge oscillations for any number of layers. The approximation error (5.22a) becomes more regular with a higher number of layers and its value tends to stabilize around 10^{-2}.

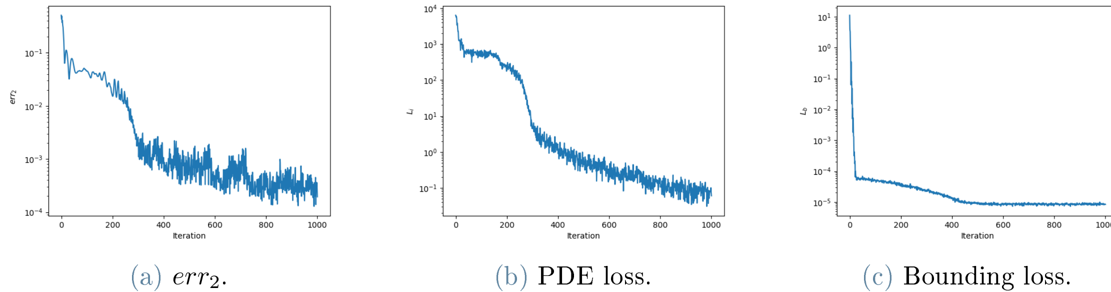
5.2. Advection-diffusion problem

In this section we discuss the results obtained with TC.E4-TC.E6, cf. Chapter 4.

Figures 5.24a, 5.24b and 5.24c show that after 500 iterations the exact and approximate solutions of test case TC.E4 are almost indistinguishable. Indeed, the plot of the error err_2 reaches 10^{-3} in less than 300 iterations (Figure 5.23a) and settles on values of order 10^{-4} until the end of the training process, always preserving a decreasing trend, even if slower and sided by arising oscillations.
The results are almost equivalent to the ones obtained in TC.E1 (Section 5.1.1), but here the boundary network learns less quickly and achieves a higher convergence value. Actually, the decrease towards order 10^{-5} is almost immediate but the plot in Figure 5.23c shows that the bounding loss settles around its equilibrium no sooner than 400 iterations. Finally, the plot of the interior loss in Figure 5.23b is equivalent to its reciprocal in the Poisson example (Figure 5.1b), rapidly decreasing towards 10^0 in about 200 iterations and then slowing down and reaching lowest order 10^{-2} within 1000 iterations.

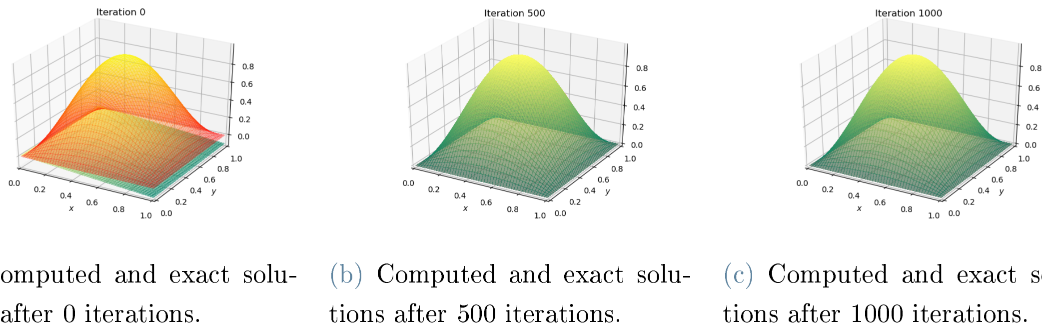
The late convergence with higher order of the boundary network does not produce meaningful variation in overall error with respect to the Poisson case analyzed in Section 5.1.1 (compare the graphs in Figures 5.23a and 5.1a for reference). This is due to the significant difference between the values obtained by the bounding and PDE loss, that is not much reduced even with this performance decline. As a consequence, we can deduce that the separation of the two networks is useful for a stronger imposition of the boundary condition, that helps attenuate the overall error, but does not influence in a subtle way

the final result. Hence, we should focus on the performance of the PDE approximation as well and try to make the corresponding structure as stable as possible.



(a) err_2. (b) PDE loss. (c) Bounding loss.

Figure 5.23: TC.E4: Measures of error as a function of the iteration counts; networks with 3 layers, 256 neurons and $\eta = 0.001$. The approximation error (5.23a) has an initially fast decrease and the keeps oscillating around 10^{-3} - 10^{-4}. The PDE loss (5.1b) reaches 10^0 in 200 iterations and then slows down its decrease, reaching 10^{-1} within 1000 iterations. The bounding loss (5.1c) converges in almost 400 iterations to 10^{-5}



(a) Computed and exact solutions after 0 iterations. (b) Computed and exact solutions after 500 iterations. (c) Computed and exact solutions after 1000 iterations.

Figure 5.24: TC.E4: Computed (green) and exact (orange) solutions after 0, 500 and 1000 iterations; networks with 3 layers, 256 neurons and $\eta = 0.001$. The initial guess (5.24a) is almost flat but the two curves almost coincide after 500 (5.2b) and 1000 5.24c iterations.

We next address TC.E5, cf. Chapter 4.
As observed in Section 5.1.2, in the region surrounding the peak the partial derivatives of both first and second order present a rapid increase, that strongly affects the approximation efficiency of the PDE network.

The evaluation of the first derivatives allows a better approximation of the solution inside the domain, as shown in Figures 5.26b and 5.26c. Indeed, in the corresponding Poisson problem (TC.E2), the interior network used to produce a solution that was opposite in sign

with respect to the expected one (Figures 5.6b, 5.6c), while in this case the approximation is closer to real value because not only the curvature but also the slope of the correct plot are learned. This result is reflected in the approximation error, that seems to stabilize at 4×10^{-2} after about 400 iterations.

The fluctuations in the plot of the overall error, although isolated and only relatively wide, are caused by the instability of the PDE network, due to the difficulty in evaluating the output corresponding to the small critical area where little data are provided. In this case not only the Laplacian but also the gradient change very rapidly around that zone, introducing an additional obstacle to learning.

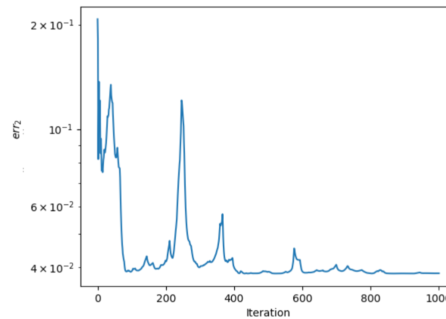
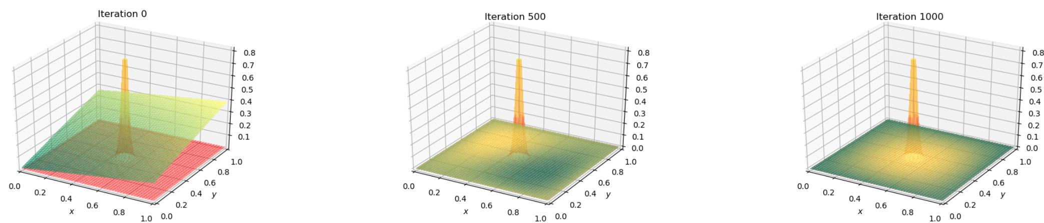


Figure 5.25: TC.E5: Approximation error as a function of the iteration counts; networks with 3 layers, 256 neurons and $\eta = 0.001$. The error tends to 4×10^{-2} in very few itetrations.



(a) Computed and exact solutions after 0 iterations.

(b) Computed and exact solutions after 500 iterations.

(c) Computed and exact solutions after 1000 iterations.

Figure 5.26: TC.E5: Computed (green) and exact (orange) solutions after 0, 500 and 1000 iterations; networks with 3 layers, 256 neurons and $\eta = 0.001$. After 500 iterations (5.26b) the two plots are very close everywhere but around the peak region, where they even have opposite curvature. At the final iteration (5.26c) the two graphs have same curvature but the peak is still not fitted.

Finally, we present the results obtained with TC.E6.
After 500 iterations the networks output is already close to the expected solution, as we can observe in Figure 5.28b, but the boundary condition is not correctly approximated even after 1000 iterations (Figure 5.28c). The plot of the computed solution is visibly less similar to the exact one in this case than in the corresponding Poisson problem TC.E3 (Figure 5.12).

The overall approximation error err_2 has an irregular behaviour, reported in Figure 5.27, and after few iterations starts oscillating around 2×10^{-2}. It does not present an overall decreasing trend and the method in this case is very unstable. This behaviour is due to the singularity presented by the first order partial derivatives of u, that have to be numerically computed for the evaluation of the PDE loss function.

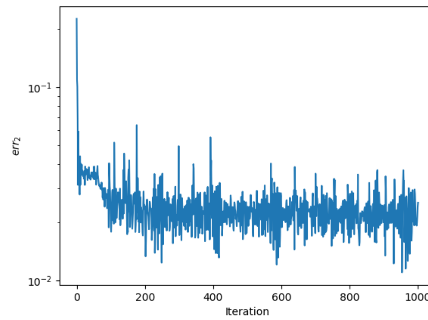
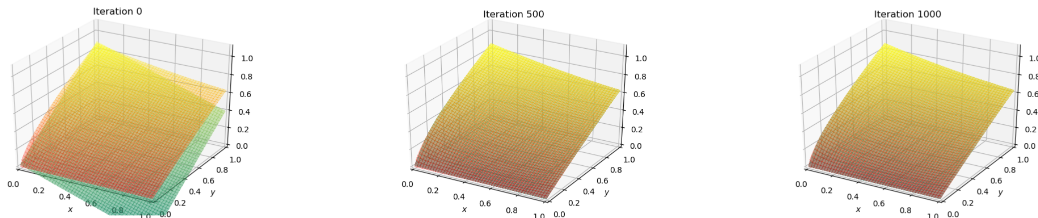


Figure 5.27: TC.E6: Approximation error as a function of the iteration counts; networks with 3 layers, 256 neurons and $\eta = 0.001$. The error starts fluctuating around 2×10^{-2} after almost 100 iterations and does not present an overall decreasing trend.



(a) Computed and exact solutions after 0 iterations.

(b) Computed and exact solutions after 500 iterations.

(c) Computed and exact solutions after 1000 iterations.

Figure 5.28: TC.E6: Computed (green) and exact (orange) solutions after 0, 500 and 1000 iterations; networks with 3 layers, 256 neurons and $\eta = 0.001$. The estimation is not very accurate on the boundary, especially around $(0, 0)$.

5.2.1. Solution with the tuned hyperparameters

The hyperparameters were tuned by trial and error for the problems TC.E4-TC.E6 in the same way as TC.E1-TC.E3 in Section 5.1.4.

The analysis of the incidence of the adjusted hyperparameters on the error produced by the method validate again the proposal of [91] when the exact solution is smooth (TC.E4). In general we expect better approximation when the structure of the networks is more complex, since every hidden layer allows a deeper learning level and more neurons improve the capacity of processing many data. This is actually true in the most simple examples and asymptotically in any case, and we have actually recognized it on the boundary networks of almost all the examples, where the loss function is easy to evaluate. Conversely, in TC.E5 the coupling has difficulties in the computation of the numerical derivatives since both the first and second derivative rapidly change around $\mathbf{x} = (0.5, 0.5)$ and this issue is not overcome by a deeper architecture.

In this section we will observe how the coupling with 1 hidden layer and 128 neurons behaves in the example with peak (TC.E5) and how the coupling with 3 hidden layers made of 256 neurons each and learning rate $\eta = 0.0001$ behaves in the least regular example (TC.E6), and compare the performance to the errors obtained at the beginning of Section 5.2.

Figure 5.30, representing the comparison between the plot of the exact solution, corresponding to the orange curve, and of its estimation, coloured in green, shows that the peak is not exactly approximated even with the tuned hyperparameters. On the other hand, the method is overall more stable, since the global approximation error, displayed in Figure 5.29, remains equal to 4×10^{-2} up to 1000 iterations, while it previously used to oscillate up to 10^{-1}. We can stop the training after a few hundreds iterations and obtain the optimal result.

The most evident improvement is however introduced in the test TC.E6. We can observe in Figures 5.32b and 5.32c that the approximation of the solution inside the domain follows the plot of the exact one, even though on the boundary edges adjacent to $(x, y) = (0, 0)$ it does not show an accurate estimation. Nevertheless, the approximation error is no more as unstable as before, as shown in Figure 5.25: after an immediate initial descent to 3×10^{-2}, it converges to 2×10^{-2} with very little oscillations within the first 1000 iterations (Figure 5.29).
The irregularity in the gradient of the exact solution does not allow the approximation error to decrease below order 10^{-2}.

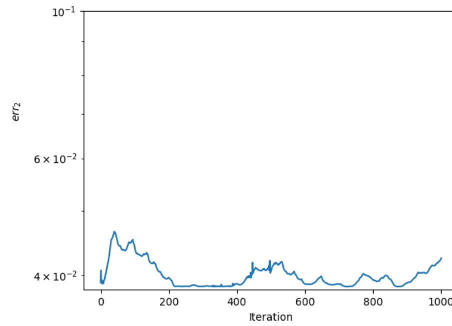
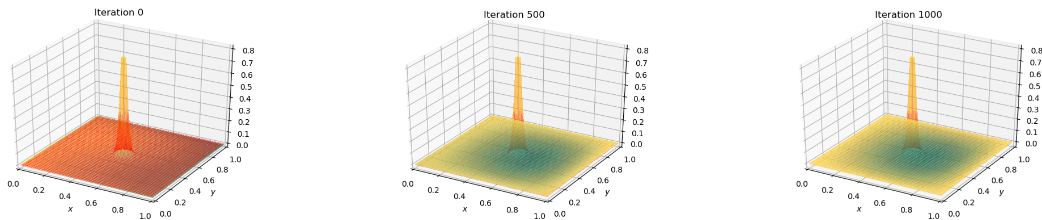


Figure 5.29: TC.E5 solved with tuned hyperparameters: Approximation error as a function of the iteration counts. The error is stably around 4×10^{-2}. Networks with 1 layer, 128 neurons and $\eta = 0.001$.



(a) Computed and exact solutions after 0 iterations.

(b) Computed and exact solutions after 500 iterations.

(c) Computed and exact solutions after 1000 iterations.

Figure 5.30: TC.E5 solved with tuned hyperparameters: Computed (green) and exact (orange) solutions after 0, 500 and 1000 iterations; networks with 1 layer, 128 neurons and $\eta = 0.001$. The orange surface represents the exact solution an the green one its numerical approximation. After 500 iterations (5.30b) the two graphs are very close everywhere but around the peak region, where they even have opposite curvature. The same happens at the final iteration (5.30c).

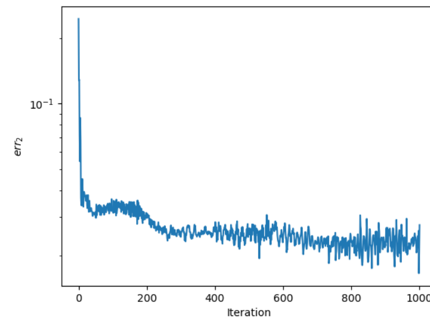
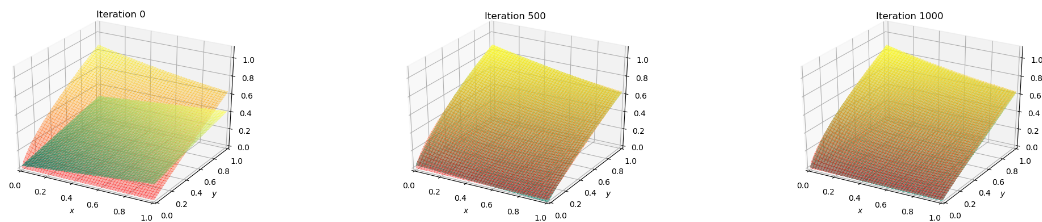


Figure 5.31: TC.E6 solved with tuned hyperparameters: Approximation error as a function of the iteration counts; networks with 3 layers, 256 neurons and $\eta = 0.0001$. The error converges after very few iterations to 2×10^{-2}.



(a) Computed and exact solutions after 0 iterations.

(b) Computed and exact solutions after 500 iterations.

(c) Computed and exact solutions after 1000 iterations.

Figure 5.32: TC.E6 solved with tuned hyperparameters: Computed (green) and exact (orange) solutions after 0, 500 and 1000 iterations; networks with 3 layers, 256 neurons and $\eta = 0.0001$. After 500 iterations (5.32b) the two plots are very close but especially when approaching the vertex $(0, 0)$ the estimation becomes more difficult. At the final iteration (5.32c) the graphs are almost identical, but for $(x, y) \approx (0, 0)$.

6 | Numerical results: evolutionary problems

In this chapter we analyze the time dependent problems introduced in Sections 4.2 and 4.3, constructed from the solutions defined in equations (4.1a)-(4.1c).

6.1. Parabolic problems

We start by presenting the numerical results on TC.P1-TC.P5, cf. Chapter 4.

Here the hyperparameters of the networks are set according to the results from Section 5.1.5 for TC.P1-TC.P3, namely 3 hidden layers with 256 neurons each and learning rate $\eta = 0.001$ for the smooth problem and the problem with peak, while for the case with singularity the chose learning rate is $\eta = 0.0001$, and from Section 5.2 for TC.P4 and TC.P5. As in the previous chapter, we observe the behavior of the coupling within 1000 iterations, examining as evaluation benchmark only the approximation error err_2 defined by equation (3.14).
Finally, as input data for the PDE network a set of 64 random points in $\Omega \times [0, T]$ is employed at each training iteration, while on the boundary 64 points per edge are sampled on $\partial\Omega$.

6.1.1. TC.P1: smooth initial condition

We present the results obtained for TC.P1, cf. Section 4.2.1.
The overall learning is slower than the corresponding stationary example and, as we can see in Figure 6.2, within 1000 iterations the approximation error reaches at most the order 10^{-2}, even if it shows a continuously decreasing trend. Moreover, the error plot presents very little oscillations, proving that the hyperparameters choice is still valid. Finally, Figure 6.1 shows the plot of the approximate solution compared to the expected one at three time snapshots. The first row shows the computed initial condition after 0, 400 and 1000 training steps. We would expect an almost exact interpolation of g, or at most a

very small loss, similar to the boundary one, since the condition is very similar. However, this does not happen, and in fact the corresponding PDE loss, as pointed out before, has high order. This is due to the fact that the initial condition, unlike the boundary one, is not learned by an independent network, so the parameters are mostly influenced by the difference between the approximation error of the proper PDE with respect to the right-hand-side functional f. Indeed, for $t = 0$ the approximate curve is lower than expected. The time evolution of the field u is simply characterized by a rescaling of the initial solution, that gets smaller and smaller values as t increases. The approximation is not able to perfectly fit the exact solution at any time within 1000 iterations, but the difference between the two plots decreases according to the range. The fact that the estimated values of u are slightly greater than the exact ones is caused by the additive term concerning the fitting of the initial condition in the PDE loss, that gives a small (as we notice in the imperfect fitting of $g(\cdot, \cdot)$ when $t = 0$) but significative effect.

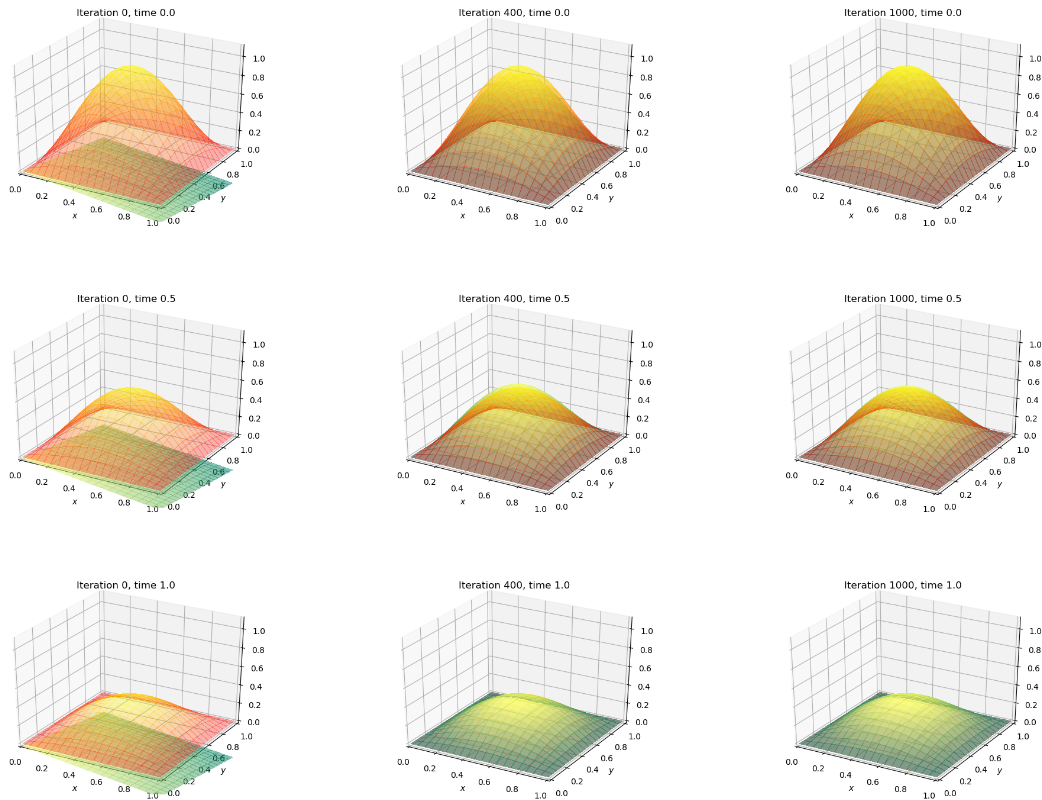


Figure 6.1: TC.P1: Computed (green) and exact (orange) solutions after 0, 400 and 1000 iterations (from left to right); networks with 3 hidden layers, 256 neurons and $\eta = 0.001$. Each row shows a different time snapshot: $t = 0$ (top), $t = 0.5$ (middle) and $t = 1$ (bottom).

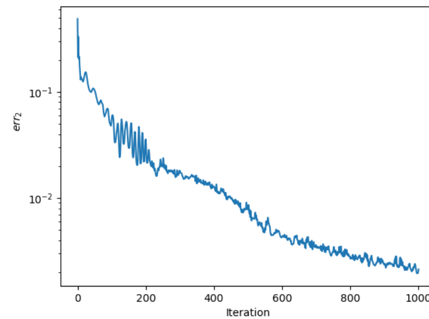


Figure 6.2: TC.P1: Computed error as a function of the iteration counts; networks with 3 hidden layers, 256 neurons and $\eta = 0.001$.

6.1.2. TC.P2: initial condition with a peak

The second test case refers to TC.P2, see Chapter 4.
We can notice a significant improvement in the approximation error shown in Figure 6.4, that does not present and increasing trend anymore, differently from what observed in TC.E2 (Figure 5.5), and is stably of order 10^{-2}, suggesting convergence to 3×10^{-2}, despite some oscillations. The better global approximation error may be due to evaluations of the solution for $t \approx 1$, when the peak is substantially reduced and, even if not exactly fitted, the numerical and exact graphs are much closer than in correspondence of $t = 0$.

Indeed, the better approximation error is reflected in the plots shown in Figure 6.3, where the approximate curves are closer but at the expense of a worse estimation allover the remaining domain area.

I recall that I do not to treat the initial condition as an additional network because it would lead to an overfitting of $u(x, y, 0)$ that spreads along all the time interval $[0, T]$. Moreover, in this case the peak could still remain undetected for the usual reason, i.e. lack of data in its neighbourhood, or on the contrary add a further risk of not being able to estimate the solution corresponding to the remaining domain area.

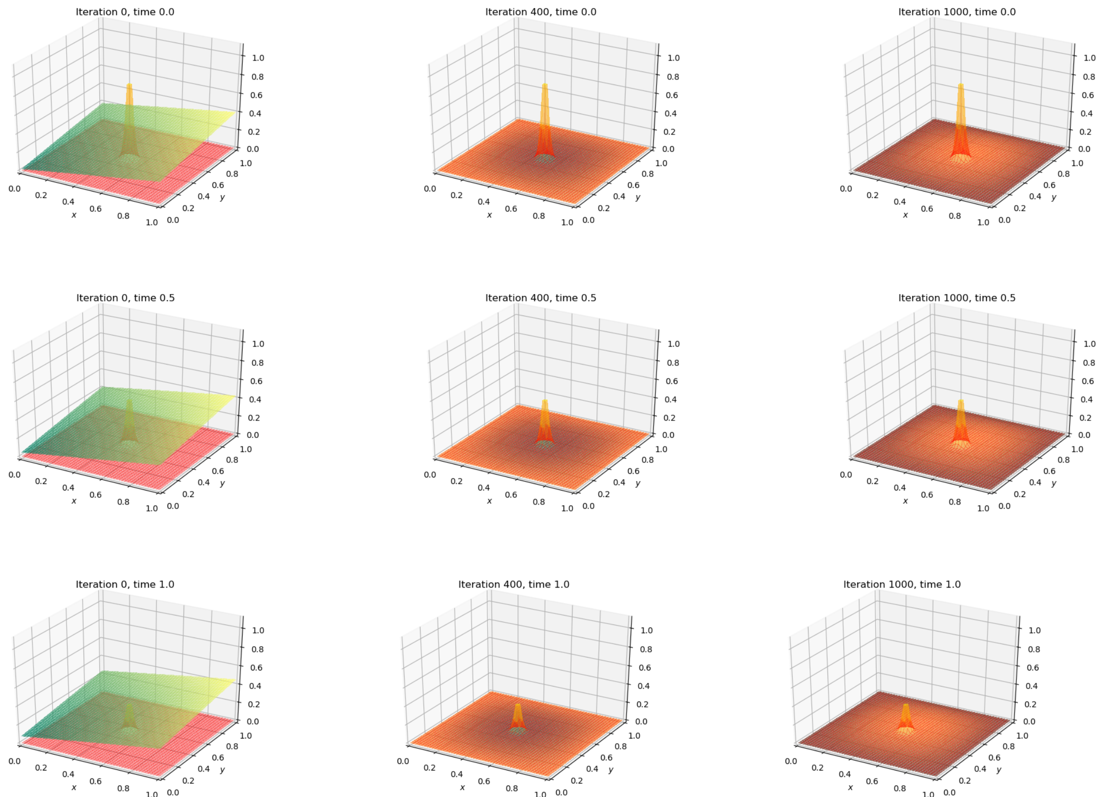


Figure 6.3: TC.P2: Computed (green) and exact (orange) solutions after 0, 400 and 1000 iterations (from left to right); networks with 3 hidden layers, 256 neurons and $\eta = 0.001$. Each row shows a different time snapshot: $t = 0$ (top), $t = 0.5$ (middle) and $t = 1$ (bottom).

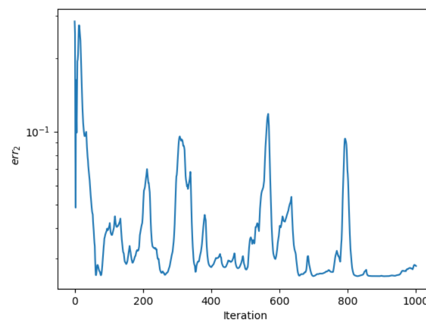


Figure 6.4: TC.P2: Computed error as a function of the iteration counts; networks with 3 hidden layers, 256 neurons and $\eta = 0.001$.

6.1.3. TC.P3: Solution with low regularity

We now present the numerical results obtained with TC.P3, see Chapter 4.

Because of the difficulty in learning form the PDE, we would expect that the minimization of the global error (Figure 6.6) is driven by the component related to the initial condition g, and a consequential overfitting of this information. Apparently, it does not really affect the approximation, in fact the the plot corresponding to $t = 0$ is not exactly equal to the expected one after 1000 iterations, as we can observe in Figure 6.5, and in particular the two curves are different for $(x, y) \approx (0, 0)$. This results however in a good estimations also for $t > 0$, where we can observe that at the final iteration the approximation is very accurate everywhere but around $(0, 0)$.
Moreover, the approximation error has a decreasing trend and attains values of order 1.5×10^{-3} within the first 1000 iterations, with small oscillations during the whole training process, except for the last 300 steps.

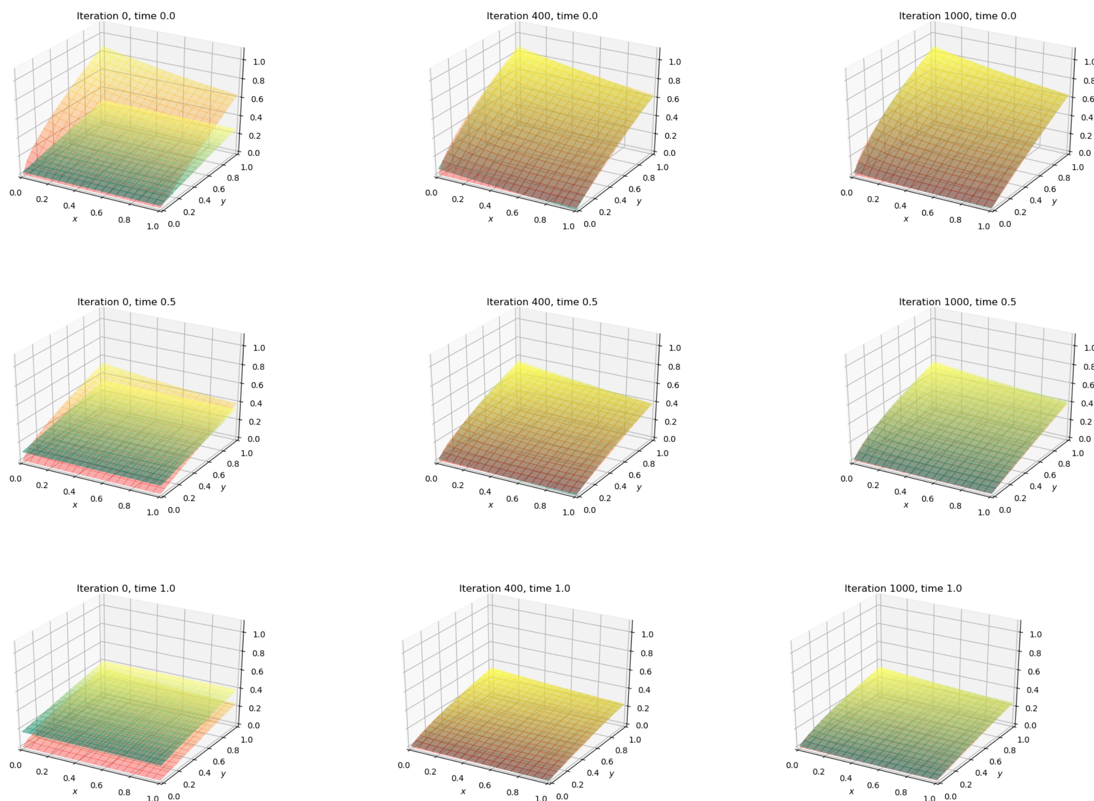


Figure 6.5: TC.P3: Computed (green) and exact (orange) solutions after 0, 400 and 1000 iterations (from left to right); networks with 3 hidden layers, 256 neurons and $\eta = 0.0001$. Each row shows a different time snapshot: $t = 0$ (top), $t = 0.5$ (middle) and $t = 1$ (bottom)

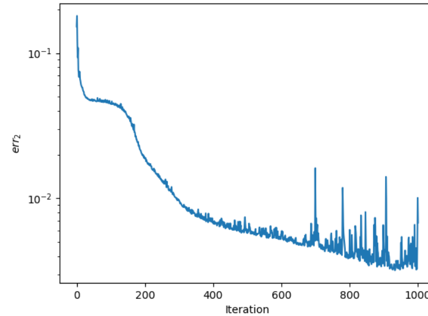


Figure 6.6: TC.P3: Computed error as a function of the iteration counts; networks with 3 hidden layers, 256 neurons and $\eta = 0.0001$.

6.1.4. Time-dependent advection-diffusion equation

In this section we present the results obtained for TC.P4 and TC.P5. We refer to Chapter 4 for the details of the test cases. The time-dependent extension of TC.E6, whose exact solution corresponds to $u^{irregular,t}$ (4.1c), is not considered since, as we have observed in Section 5.2.1, the method is not efficient when the transport term is introduced.

The performance of the method applied to TC.P4 with a smooth initial condition are almost the same as for the corresponding heat equation (TC.E4, Section 6.1.1): the approximation error is decreasing and converges to 3×10^{-2} in 200 iterations (Figure 6.8). The very narrow oscillations presented by the error plot suggest that the choice of parameters is still valid also in this case.
Finally, Figure 6.7 shows a better approximation of the solution at every time instant than in TC.P1 (Figure 6.1), since the exact and approximate plot are very close for every t and in particular the maximum value is almost perfectly estimated.

The exact solution of TC.P5 presents a peak in the middle of the spatial domain Ω that becomes less severe as t increases. The data for this test case are those given in Section 4.3.2.
The networks applied to this example have one hidden layer with 128 neurons and learning rate $\eta = 0.001$, as discussed in Section 5.2.4 for the stationary case. The coupling is very stable, as we can observe in Figure 6.10 representing the plot of the approximation error. Indeed, not only it always remains of order 10^{-2} within the first 1000 iterations, but it also converges almost immediately at the value 3×10^{-2} with negligible oscillations. This means that in the case of regular solution with sudden variation of both the derivatives and the corresponding value of the function the method with just one layer and 128 neurons learns very fast and can be stopped after a few hundreds of iterations.

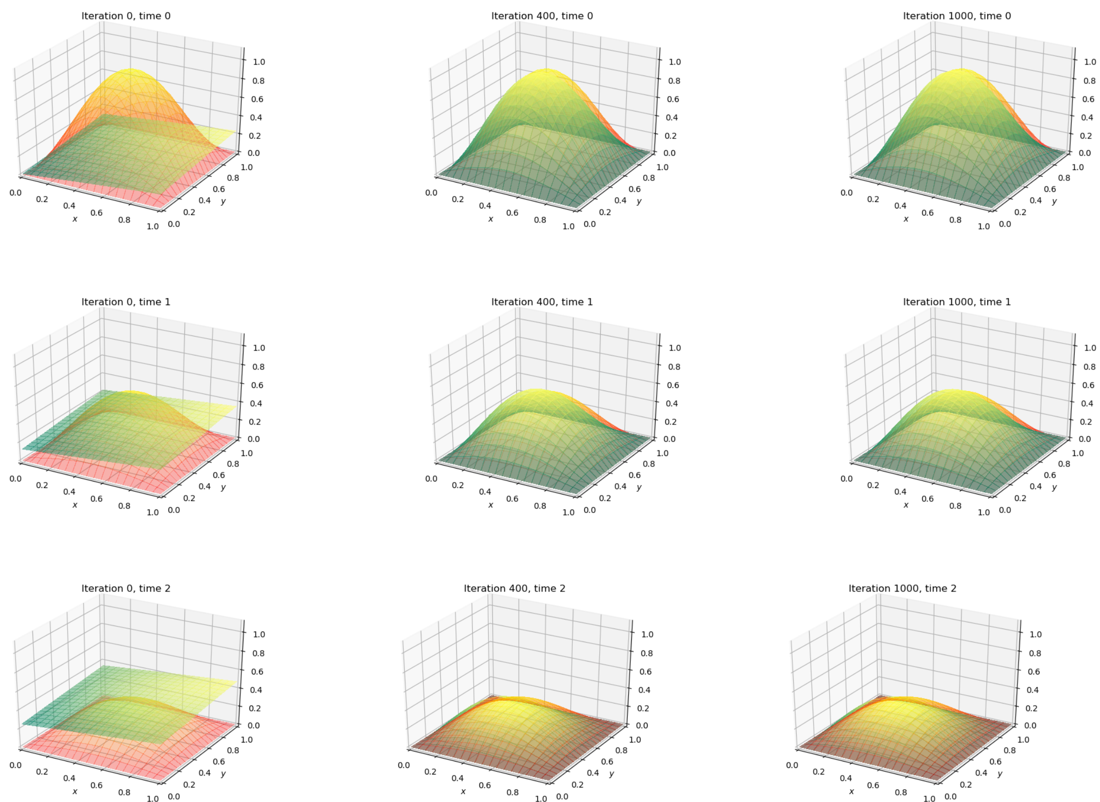


Figure 6.7: TC.P4: Computed (green) and exact (orange) solutions after 0, 400 and 1000 iterations (from left to right); networks with 3 hidden layers, 256 neurons and $\eta = 0.001$. Each row shows a different time snapshot: $t = 0$ (top), $t = 0.5$ (middle) and $t = 1$ (bottom).

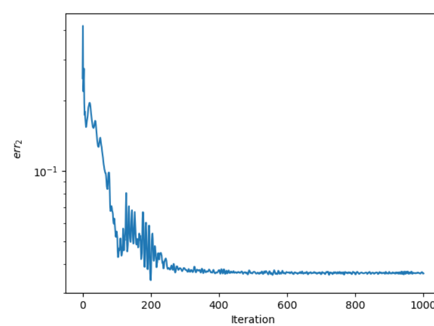


Figure 6.8: TC.P4: Computed error as a function of the iteration counts; networks with 3 hidden layers, 256 neurons and $\eta = 0.001$.

In fact, we see in Figure 6.9 that the difference among the three columns, representing the approximate solution after 0, 400 and 1000 training iterations, is not very evident. The initial guess is already very close to the exact plot for all the three time instants considered, and waiting for the training process to reach the 1000th repetition is not apparently useful. This observation is in line with the approximation error that does never change its order of magnitude and has fluctuations of amplitude no greater than 0.001.

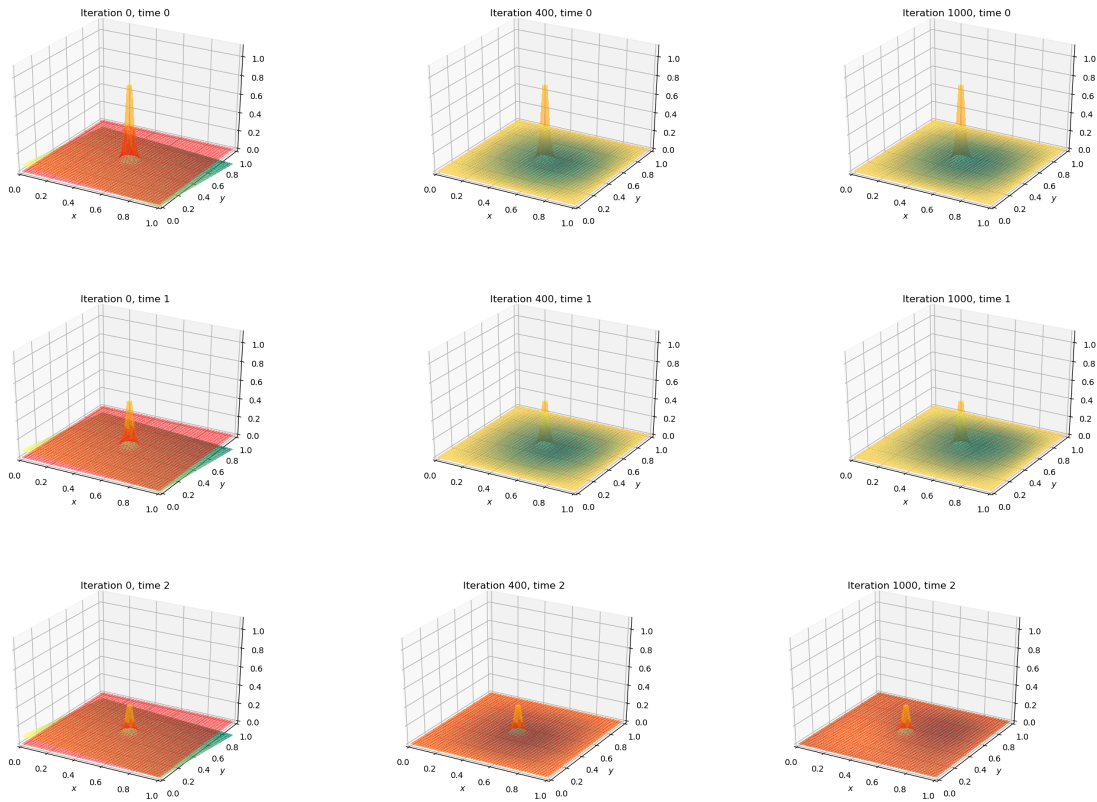


Figure 6.9: TC.P5: Computed (green) and exact (orange) solutions after 0, 400 and 1000 iterations (from left to right); networks with 3 hidden layers, 256 neurons and $\eta = 0.001$. Each row shows a different time snapshot: $t = 0$ (top), $t = 0.5$ (middle) and $t = 1$ (bottom).

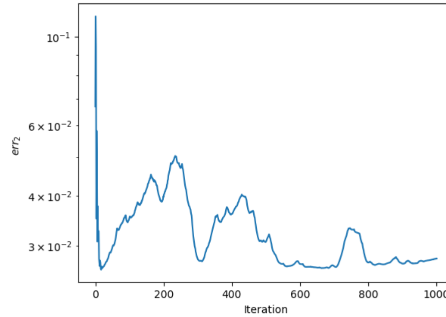


Figure 6.10: TC.P5: Computed error as a function of the iteration counts; networks with 1 hidden layer, 128 neurons and $\eta = 0.001$.

6.2. Hyperbolic problems

The last test cases concern hyperbolic problems, specifically linear transport equations, as specified in Chapter 4. On the boundary, Dirichlet constraints are given as in the previous examples. The specific problems are built again by applying the method of manufactured solutions on the same fields $u(x, y, t)$ considered for the parabolic advection-diffusion tests in Section 6.1.4.

The training set is defined in the same way as for the other evolutionary problems, by sampling 64 space-time locations for the PDE network and 64 points per edge for the boundary one.

The hyperparameters for the examples discussed in this chapter are set according to the choices made in Section 5.1.4, i.e. 3 hidden layers with 256 neurons each and learning rate $\eta = 0.001$.

The tests TC.H1 and TC.H2 represent the evolution in time of the exact solutions of the problems studied in Chapter 5, according to the conservation law defined by the linear transport equation (1.10).

6.2.1. TC.H1: smooth initial condition

The first problem has a smooth solution $u(x, y, t)$ at every time $t \in [0, 1]$, defined by the function $u^{smooth,t}$ (4.1a) defined in Chapter 4.
Although the simple shape of g_D makes the boundary condition very easy to be learned, the overall approximation error converges to a value around 2×10^{-2}. This is due to the performance of the PDE network, whose loss function, involving numerical derivatives in

every direction, is more difficult to be evaluated. One proposal for faster reduction of the PDE approximation error could be to increase the learning rate of the corresponding network but, as we can observe in Figure 6.11, after 800 iterations the fluctuations around the convergence value already start to become wider, and they would be made even worse by a greater η.

However, the estimation obtained for the solution of this example with smooth data is good and the convergence to the optimal result very fast.

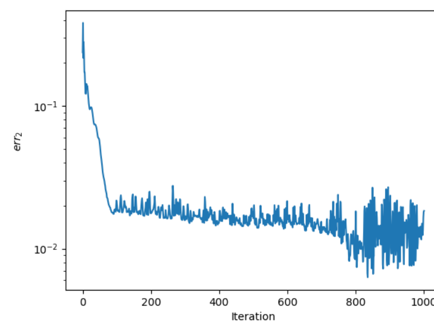


Figure 6.11: TC.H1: Computed error as a function of the iteration counts.
Networks with 3 hidden layers, 256 neurons and $\eta = 0.001$.

6.2.2. TC.H2: initial condition with a peak

Here we present the results relative to (TC.H2). We have repeated the same numerical experiments as before and the results are shown in Figure 6.12.
The plot of the error err_2 stably attains its minimum after almost 300 iterations, and then it essentially becomes constant.

In this case the fast convergence entails that the process may be stopped after just a couple hundreds steps for reaching the best possible approximation. This suggests that the peak, not recognized in 1000 iterations, will never be learned. The reason behind the inability of the method to detect that sudden variation of the solution is maybe due to the lack of data, in terms of space-time locations gathered in the peak region, given as input to the PDE network, but a downside of this solution is a possible overfitting of the high values taken by the function in that area.

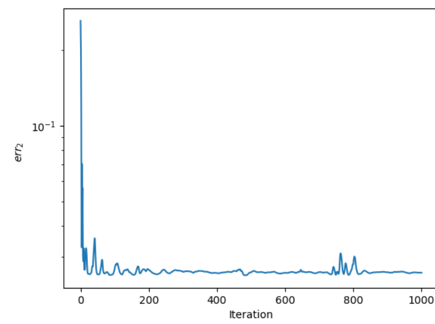


Figure 6.12: TC.H2: Computed error as a function of the iteration counts. Networks with 3 hidden layers, 256 neurons and $\eta = 0.001$.

7 | Conclusions and future developments

We have extensively investigated an ANN-based approach to numerically solve elliptic, parabolic and hyperbolic problems. The extensive testing is justified by a lack of strong theoretical results. We observed the behaviour of the machine-learning-based approach on different types of problems and deduced some general conclusions about its performance.

1. From the hyperparameters tuning performed in Section 5.1.4, where the training algorithm was repeated up to 20000 times, we can notice that the approximation error stops decreasing after the very few hundreds of initial iterations, then it seems to stagnate and starts presenting oscillations with increasing amplitude. In particular, most examples end up attaining the minimum global error within 300-400 training iterations at least for the considered configurations. As expected, both the type of PDE (i.e. elliptic, parabolic, hyperbolic) and the regularity of the solution have a central role in the choice of the ideal number of training iterations. Indeed, they are maximally reduced in all the test cases with smooth solution, and we can also observe a shrink in the hyperbolic examples with solutions having singularity on the boundary when the diffusion term is dropped and only first order partial derivatives need to be approximated. In the test cases with peak the approximation properties of the network seem to be less accurate because the chosen input locations do not allow a better learning. On the other hand, this issue could only be overcome by sampling many data points in the neighbourhood of the peak, but this decision cannot be made apriori, without knowing the shape of the solutions.

2. In the approximation of the solution with peak no significant improvement is introduced by increasing the number of layers or neurons, and in fact in the diffusion-transport cases we have observed that the best choice was to actually reduce them. For this reason, we can conclude that the proposed method, at least in our test cases, is not able to detect particular local features such as steep peaks.

3. Although no theoretical result was proven, we can observe that all the examples show that the overall error is not better than 10^{-3}. This appears to depend on the solution regularity in the interior of the domain, since the examples corresponding to the smooth one and to the one with singularity on the boundary in some cases reach order 10^{-4}. In particular, in the test cases with smooth solution the best results are achieved for second order elliptic operators.

Moreover, this method seems to be computationally efficient in the time-dependent cases, since it does not require a discretization in time of the problem and allows us to treat t as an additional input variable that does not imply a high increase in the training execution time.
In conclusion, the proposed method is suitable for fast black-box approximations, also because any extension to other types of PDE or boundary conditions is very intuitive, but cannot be substituted to the traditional grid-based methods. The main technical drawback consists in the minimum errors achieved. A huge concern also regard the lack of physical knowledge, since the system law is only taken into account for the loss definition. Future developments of this idea could overcome these issues following two different paths:

a) Exploiting more the structure of the neural networks involved;

b) Changing the focus and trying to combine the machine learning approach and Galerkin-type schemes.

A possible extension also consists in applying autotuning techniques, as mentioned in Chapter 2, in order to set the optimal values of the hyperparameters and improve the efficiency. Another further development could be to apply physics-informed neural networks to the calibration of the values assigned to some parameters in the traditional methods.
Moreover, the extension to other types of boundary conditions and nonlinear problems, such as the Burgers equation, should be investigated in order to fully exploit the potentiality of the proposed fully ANN-based approach.

Bibliography

[1] O. Abiodun, A. Jantan, M. Singh, M. Anbar, Z. Zaaba, and O. Oludare Abiodun. Forensic dna profiling for identifying an individual crime. *International Journal of Civil Engineering and Technology*, 9(7):755–765, 2018.

[2] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi. Learning activation functions to improve deep neural networks. In *3rd International Conference on Learning Representations, ICLR 2015 - Workshop Track Proceedings*, 2015.

[3] S. Amari. A Theory of Adaptive Pattern Classifiers. *IEEE Transactions on Electronic Computers*, EC-16(3):299–307, 1967.

[4] P. Antonietti, A. Cangiani, J. Collis, Z. Dong, E. Georgoulis, S. Giani, and P. Houston. Review of discontinuous Galerkin Finite Element methods for Partial Differential Equations on complicated domains. *Lecture Notes in Computational Science and Engineering*, 114:279–308, 2016.

[5] S. Arora, N. Golowich, N. Cohen, and W. Hu. A convergence analysis of gradient descent for deep linear neural networks. In *7th International Conference on Learning Representations, ICLR 2019*, 2019.

[6] A. R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, 1993.

[7] M. Baymani, S. Effati, and A. Kerayechian. A feed-forward neural network for solving Stokes problem. *Acta Applicandae Mathematicae*, 116(1):55–64, 2011.

[8] S. Becker and Y. Lecun. Improving the convergence of back-propagation learning with second-order methods. *Proceedings of the 1988 Connectionist Models Summer School, San Mateo*, pages 29–37, 1989.

[9] A. I. Beltzer and T. Sato. Neural classification of finite elements. *Computers & Structures*, 81(24-25):2331–2335, 2003.

[10] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research (JMLR)*, 13:281–305, 2012.

[11] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011, NIPS 2011*, 2011.

[12] J. Berner, P. Grohs, and A. Jentzen. Analysis of the generalization error: empirical risk minimization over deep artificial neural networks overcomes the curse of dimensionality in the numerical approximation of Black-Scholes partial differential equations. *SIAM Journal on Mathematics of Data Science*, 2(3):631–657, 2020.

[13] P. Bochev and M. Gunzburger. *Least-squares finite element methods*, volume 166. 2009.

[14] S. Bock and M. Weiß. A proof of local convergence for the adam optimizer. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2019-July, 2019.

[15] S. Bock, J. Goppold, and M. Weiß. An improvement of the convergence proof of the Adam-optimizer. *arXiv e-prints*, 1804.10587, 2018.

[16] L. Bottou et al. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8):12, 1991.

[17] S. C. Brenner and L. R. Scott. *The mathematical theory of finite element methods*, volume 15 of *Texts in Applied Mathematics*. Springer, New York, third edition, 2008.

[18] L.-W. Chan and F. Fallside. An adaptive training algorithm for back propagation networks. *Computer Speech and Language*, 2(3-4):205–218, 1987.

[19] Z. Chen, L. Liu, and L. Mu. Solving the linear transport equation by a deep neural network approach. *arXiv e-prints*, 2102.09157, 2021.

[20] M. Chiaramonte, M. Kiener, et al. Solving differential equations using neural networks. *Machine Learning Project*, 1, 2013.

[21] S.-B. Cho and J. Kim. Rapid backpropagation learning algorithms. *Circuits, Systems, and Signal Processing*, 12(2):155–175, 1993.

[22] Y. Cun, I. Guyon, L. Jackel, D. Henderson, B. Boser, R. Howard, J. Denker, W. Hubbard, and H. Graf. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, 1989.

[23] Y. L. Cun. A learning procedure for assymetric threshold network. *Proceedings of Cognitiva*, 85:599–604, 1985.

[24] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[25] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13:165–202, 2012.

[26] N. Discacciati, J. Hesthaven, and D. Ray. Controlling oscillations in high-order discontinuous galerkin schemes using artificial viscosity tuned by neural networks. *Journal of Computational Physics*, 409, 2020.

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

[28] S. Du, J. Lee, H. Li, L. Wang, and X. Zhai. Gradient descent finds global minima of deep neural networks, 2019.

[29] C. Duarte, I. Babuška, and J. Oden. Generalized finite element methods for three-dimensional structural mechanics problems. *Computers and Structures*, 77(2):215–232, 2000.

[30] I. Elbadawi, A. R. Gallant, and G. Souza. An elasticity can be estimated consistently without a priori knowledge of functional form. *Econometrica: Journal of the Econometric Society*, pages 1731–1751, 1983.

[31] J. Fu, H. Zheng, and T. Mei. Look closer to see better: Recurrent attention convolutional neural network for fine-grained image recognition. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4476–4484, 2017.

[32] K.-I. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

[33] A. R. Gallant and H. White. On learning the derivatives of an unknown mapping with multilayer feedforward networks. *Neural Networks*, 5(1):129–138, 1992.

[34] P. Grohs and L. Herrmann. Deep neural network approximation for high-dimensional elliptic pdes with boundary conditions. *arXiv e-prints*, 2020.

[35] S. Grossberg. Embedding fields: A theory of learning with physiological implications. *Journal of Mathematical Psychology*, 6(2):209–239, 1969.

[36] R. Hecht-Nielsen. Kolmogorov's mapping neural network existence theorem. In

Proceedings of the international conference on Neural Networks, volume 3, pages iii/11–14. IEEE Press New York, 1987.

[37] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[38] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[39] K. Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(5):551–560, 1990.

[40] T.-J. Huang. Imitating the brain with neurocomputer a "new" way towards artificial general intelligence. *International Journal of Automation and Computing*, 14(5): 520–531, 2017.

[41] M. Hutzenthaler, A. Jentzen, T. Kruse, and T. A. Nguyen. A proof that rectified deep neural networks overcome the curse of dimensionality in the numerical approximation of semilinear heat equations. *Partial Differential Equations and Applications*, 1(2): Paper No. 10, 34, 2020.

[42] R. A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4):295–307, 1988.

[43] X. Jiang, B. Hu, S. Chandra Satapathy, S.-H. Wang, and Y.-D. Zhang. Fingerspelling identification for Chinese sign language via AlexNet-based transfer learning and Adam optimizer. *Scientific Programming*, 2020, 2020.

[44] H. Jilani, A. Bahreininejad, and M. Ahmadi. Adaptive finite element mesh triangulation using self-organizing neural networks. *Advances in Engineering Software*, 40 (11):1097–1103, 2009.

[45] B. L. Kalman and S. C. Kwasny. Why tanh: choosing a sigmoidal function. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 4, pages 578–581. IEEE, 1992.

[46] B. Karlik and A. V. Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122, 2011.

[47] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[48] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[49] I. E. Lagaris, A. C. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998.

[50] I. E. Lagaris, A. C. Likas, and D. G. Papageorgiou. Neural-network methods for boundary value problems with irregular boundaries. *IEEE Transactions on Neural Networks*, 11(5):1041–1049, 2000.

[51] Y. LeCun. A theoretical framework for back-propagation. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School, CMU, Pittsburg, PA*, volume 1, pages 21–28. Morgan Kaufmann, 1988.

[52] U. Lee. *Spectral Element method in structural dynamics.* John Wiley & Sons, 2009.

[53] R. J. LeVeque. *Finite volume methods for hyperbolic problems.* Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2002.

[54] R. J. LeVeque. *Finite difference methods for ordinary and partial differential equations.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2007.

[55] Z. Long, Y. Lu, and B. Dong. PDE-Net 2.0: learning PDEs from data with a numeric-symbolic hybrid deep network. *Journal of Computational Physics*, 399:108925, 17, 2019.

[56] M. Lynch, H. Patel, A. Abrahamse, A. Rajendran, and L. Medsker. Neural network applications in physics. In *Proceedings of the International Joint Conference on Neural Networks*, volume 3, pages 2054–2058, 2001.

[57] L. Lyu, Z. Zhang, M. Chen, and J. Chen. Mim: A deep mixed residual method for solving high-order partial differential equations. *arXiv e-print*, 2006.04146.

[58] A. Malek and R. S. Beidokhti. Numerical solution for high order differential equations using a hybrid neural network—optimization method. *Applied Mathematics and Computation*, 183(1):260–271, 2006.

[59] L. Manevitz, M. Yousef, and D. Givoli. Finite-element mesh generation using self-organizing neural networks. *Computer-Aided Civil and Infrastructure Engineering*, 12(4):233–250, 1997.

[60] S. Marra, M. Iachino, and F. Morabito. Tanh-like activation function implementation for high-performance digital neural systems. In *2006 Ph. D. Research in Microelectronics and Electronics*, pages 237–240. IEEE, 2006.

[61] G. L. Martin and J. A. Pittman. Recognizing hand-printed letters and digits using backpropagation learning. *Neural Computation*, 3(2):58–267, 1991.

[62] A. Maurer. Algorithmic stability and meta-learning. *Journal of Machine Learning Research (JMLR)*, 6:967–994, 2005.

[63] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[64] H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, and F. Hutter. Towards automatically-tuned neural networks. In F. Hutter, L. Kotthoff, and J. Vanschoren, editors, *Proceedings of the Workshop on Automatic Machine Learning*, volume 64 of *Proceedings of Machine Learning Research*, pages 58–65, 2016.

[65] C. Michoski, M. Milosavljević, T. Oliver, and D. Hatch. Solving differential equations using deep neural networks. *Neurocomputing*, 399:193–212, 2020.

[66] M. Minsky and S. Papert. Perceptrons: An introduction to computational geometry. *MIT Press*, 1969.

[67] Y. Mori and K. Yokosawa. Neural networks that learn to discriminate similar kanji characters. In *Advances in Neural Information Processing Systems*, pages 332–339, 1989.

[68] T. Nguyen-Thien and T. Tran-Cong. Approximation of functions and their derivatives: A neural network implementation with applications. *Applied Mathematical Modelling*, 23(9):687–704, 1999.

[69] A. Noviko. On convergence proofs for perceptrons. In *Report at the Symposium on Mathematical Theory of Automata*, pages 24–26, 1963.

[70] M. J. Orr. Introduction to radial basis function networks. *Technical Report, Center for Cognitive Science, University of Edinburgh*, 1996.

[71] D. B. Parker. Learning logic technical report tr-47. *Center of Computational Re-*

search in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA, 1985.

[72] M. Pillati. Neural networks in classification problems: a solution based on binary segmentation methods. *Statistica*, 61(3):407–421 (2002), 2001.

[73] J.-f. Qiao, Y. Zhang, and H.-g. Han. Fast unit pruning algorithm for feedforward neural network design. *Applied Mathematics and Computation*, 205(2):622–627, 2008.

[74] A. Quarteroni. *Numerical models for differential problems*, volume 16 of *MS&A. Modeling, Simulation and Applications*. Springer, Cham, 2017.

[75] I. Qureshi. Glaucoma detection in retinal images using image processing techniques: a survey. *International Journal of Advanced Networking and Applications*, 7(2):2705–2718, 2015.

[76] M. Raissi. Deep hidden physics models: deep learning of nonlinear partial differential equations. *Journal of Machine Learning Research (JMLR)*, 19:Paper No. 25, 24, 2018.

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

[78] P. Ramuhalli, L. Udpa, and S. S. Udpa. Finite-element neural networks for solving differential equations. *IEEE Transactions on Neural Networks*, 16(6):1381–1392, 2005.

[79] P. J. Roache. Code verification by the method of manufactured solutions. *Journal of Fluids Engineering*, 124(1):4–10, 2002.

[80] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[81] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv e-print*, 1609.04747, 2017.

[82] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[83] S. Salsa. *Partial differential equations in action*, volume 99 of *Unitext*. Springer, [Cham], third edition, 2016. From modelling to theory, La Matematica per il 3+2.

[84] L. Schwander, D. Ray, and J. Hesthaven. Controlling oscillations in spectral methods

by local artificial viscosity governed by neural networks. *Journal of Computational Physics*, 431, 2021.

[85] S. O. Shahdi and S. A. R. Abu-Bakar. Neural network-based approach for face recognition across varying pose. *International Journal of Pattern Recognition and Artificial Intelligence*, 29, 2015.

[86] P. Sibi, S. Allwyn Jones, and P. Siddarth. Analysis of different activation functions using back propagation neural networks. *Journal of Theoretical and Applied Information Technology*, 47(3):1344–1348, 2013.

[87] J. Sirignano and K. Spiliopoulos. DGM: a deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.

[88] D. Stathakis. How many hidden layers and nodes? *International Journal of Remote Sensing*, 30(8):2133–2147, 2009.

[89] A. J. Turner and J. F. Miller. Neuroevolution: evolving heterogeneous artificial neural networks. *Evolutionary Intelligence*, 7(3):135–154, 2014.

[90] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv e-prints*, 1609.08144, 2016.

[91] K. Xu, B. Shi, and S. Yin. Deep Learning for Partial Differential Equations (PDEs). 2018.

[92] N. Yadav, A. Yadav, M. Kumar, et al. *An introduction to neural network methods for differential equations*. Springer, 2015.

[93] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[94] W. Yue, Z. Wang, H. Chen, A. Payne, and X. Liu. Machine learning with applications in breast cancer diagnosis and prognosis. *Designs*, 2(2), 2018.

[95] G. Zhang. Neural networks for classification: a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 30(4):451–462, 2000.

[96] J.-S. Zhang and X.-C. Xiao. Predicting chaotic time series using recurrent neural network. *Chinese Physics Letters*, 17(2):88–90, 2000.

[97] Z. Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th International Symposium on Quality of Service, IWQoS 2018*, 2019.

A | Python code

The implementation of the method is based on Python **TensorFlow** library, an open-source platform for creation and training of machine learning models.

Three base classes are defined for networks solving generic bidimensional PDE problems and their extension to $d \in \mathbb{N}$ dimensions.
For more details about the code see
https://github.com/beatricecrippa/ANNs-for-PDEs.
Starting from the base classes (contained in the file pdebase.py) some derived ones are defined in file problems.py, each one related to a specific example among those discussed in Chapters 5 and 6.
For the execution of this code the installation of the Pyhton libraries **TensorFlow** and **NumPy**, for training, and **MatplotLib**, **mpl_toolkits.mplot3d** and **drawnow**, for the plots, is required. After checking the installation of these packages, download the git repository linked above, modify the file **training.py**, according to the desired problem defined in **problems.py**, and run it in order to obtain the plots reported in Chapters 5 and 6.

Let's start from some fundamental functions for the computation of the partial derivatives and their evaluation at some locations x in the domain. In particular, **compute_dx** and **compute_dy** return the derivatives with respect to x and y coordinates, while **compute_delta** returns the approximate Laplacian. All of them rely on the same tensorflow method **gradients(y,x)**, that computes the derivatives in every direction of a function u, given a tensor x of coordinates and a tensor y of the corresponding images u($\mathbf{x}$), by interpolating the input points.

```
#compute_delta: approximate laplacian
def compute_delta(u, x):
        grad = tf.gradients(u, x)[0]
        g1 = tf.gradients(grad[:,0], x)[0]
        g2 = tf.gradients(grad[:,1], x)[0]
        delta = g1[:,0] + g2[:,1]
```

```
        return delta

#compute_dx: x-derivative
def compute_dx(u,x):
        grad = tf.gradients(u, x)[0]
        dudx = grad[:,0]
        return dudx

#compute_dy: y-derivative
def compute_dy(u,x):
        grad = tf.gradients(u, x)[0]
        dudy = grad[:,1]
        return dudy
```

In this appendix we will focus on NNPDE, the class implementing the proposed method for the Poisson-Dirichlet problem.

NNPDE is initialized by the variables **batch_size** (size of batches for Adam optimizer), **N** (number of hidden dense layers) and **refn** (number of reference points), whose values are assigned to homonymous class variables. In the initialization phase the three measures of error discussed in Chapter 3, represented by the tensors **rloss**, **rbloss** and **rl2** are set to **NULL**.

The neural networks are implemented as two member functions **self.subnetwork** and **self.bsubnetwork**, whose variable scopes are respectively called "inner" and "boundary". The two subnetworks are equivalent and constructed with N hidden dense layers with 64 nodes each and tanh activation function and a 1-dimensional linear output layer. The networks are trained by the member function self.train, that evaluates first the boundary one until it has $loss \leq 10^{-5}$ and then the PDE ones, with learning rate 0.001 and objective functions bloss, given by the sum of square errors on the boundary, and loss, defined by a method **loss_function** that returns the sum of square errors in the PDE equation, making use of the previously introduced function **compute_delta**.

Finally, the approximation given by the training on the boundary is saved in the variable **self.u_b** and the approximation in the inner domain is recorded in **self.u**, given by the sum of **self.u_b** and the result returned by the PDE network, multiplied by a function B such that $B\big|_{\partial\Omega} = 0,\ B(x,y) = x(1-x)y(1-y)\ \forall (x,y) \in \Omega$.

In order to apply this class to a concrete problem, the function members **self.f**, **self.exactsol** and **self.tfexactsol**, corresponding to the external source data and analytic solution (in

terms of **numpy** and **tensorflow** libraries) must be defined in derived classes. The exact solution is not necessary for the learning process, but it is only used for the plot representation of the approximation error.

In the following lines, the class initializer and the definition of the functions B and of the PDE loss:

```
class NNPDE:
    def __init__(self, batch_size, N, refn):
        # measures of error
        self.rloss = []              # interior loss
        self.rbloss = []             # bounding loss
        self.rl2 = []                          # err_2

        # grid for the error computation
        self.refn = refn   # reference points for the error computation
        x = np.linspace(0, 1, refn)
        y = np.linspace(0, 1, refn)
        self.X, self.Y = np.meshgrid(x, y)
        self.refX = np.concatenate([self.X.reshape((-1, 1)),
          self.Y.reshape((-1, 1))], axis=1)

        self.batch_size = batch_size  # batchsize (input dimensionality)
        self.N = N                             # number of hidden layers

        self.x = tf.placeholder(tf.float64, (None, 2))
        self.x_b = tf.placeholder(tf.float64, (None, 2))

        self.u_b = self.bsubnetwork(self.x_b, False)
        self.u = self.bsubnetwork(self.x, True) +
            self.B(self.x) * self.subnetwork(self.x, False)

        self.bloss = tf.reduce_sum((self.tfexactsol(self.x_b)
           -self.u_b)**2)
        self.loss = self.loss_function()

        var_list1 =
            tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
```

```
            "boundary")
        self.opt1 = tf.train.AdamOptimizer(learning_rate=
            0.001).minimize(self.bloss, var_list=var_list1)
        var_list2 =
            tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
            "inner")
        self.opt2 = tf.train.AdamOptimizer(learning_rate=
             0.001).minimize(self.loss, var_list=var_list2)
        self.init = tf.global_variables_initializer()

    # B = 0 on the boundary
    def B(self, x):
        return (0-x[:, 0]*(1-x[:, 0])*(0-x[:, 1])*(-x[:, 1])

    def loss_function(self):
        deltah = compute_delta(self.u, self.x)
        delta = self.f(self.x)
        res = tf.reduce_sum((deltah - delta) ** 2)
        assert_shape(res, ())
        return res
```

In the class NNPDE the following two methods define the networks structure:

```
#subnetwork defines a dense neural network on inner points
#with tanh activation
def subnetwork(self, x, reuse = False):
        with tf.variable_scope("inner"):
                for i in range(self.N):
                        x = tf.layers.dense(x, 256,
                            activation=tf.nn.tanh,
                            name="dense{}".format(i), reuse=reuse)
        x = tf.layers.dense(x, 1, activation=None, name="last",
            reuse=reuse)
        x = tf.squeeze(x, axis=1)
        return x

#bsubnetwork defines a dense neural network on boundary
#points with tanh activation
```

```
def bsubnetwork(self, x, reuse = False):
        with tf.variable_scope("boundary"):
                for i in range(self.N):
                        x = tf.layers.dense(x, 256,
                           activation=tf.nn.tanh,
                           name="bdense{}".format(i), reuse=reuse)
        x = tf.layers.dense(x, 1, activation=None,
           name="blast", reuse=reuse)
        x = tf.squeeze(x, axis=1)
        return x
```

Finally, the method self.train of the class NNPDE must be called in order to perform the training session:

```
def train(self, sess, i=-1):
        #random (boundary) coordinates
        bX = np.zeros((4*self.batch_size, 2))
        bX[:self.batch_size,0] = np.random.rand(self.batch_size)
        bX[:self.batch_size,1] = 0.0
        bX[self.batch_size:2*self.batch_size, 0] =
           np.random.rand(self.batch_size)
        bX[self.batch_size:2*self.batch_size, 1] = 1.0
        bX[2*self.batch_size:3*self.batch_size, 0] = 0.0
        bX[2*self.batch_size:3*self.batch_size, 1] =
           np.random.rand(self.batch_size)
        bX[3*self.batch_size:4*self.batch_size, 0] = 1.0
        bX[3 * self.batch_size:4 * self.batch_size, 1] =
           np.random.rand(self.batch_size)

        # training of the boudnary network
        bloss = sess.run([self.bloss], feed_dict={self.x_b: bX})[0]
        if bloss>1e-5:
                for _ in range(5):
                        _, bloss = sess.run([self.opt1, self.bloss],
                           feed_dict={self.x_b: bX})

    # random coordinates
        X = np.random.rand(self.batch_size, 2)
```

```
# training of the PDE network
_, loss = sess.run([self.opt2, self.loss],
    feed_dict={self.x: X})

########## record loss ############
self.rbloss.append(bloss)
self.rloss.append(loss)
uh = sess.run(self.u, feed_dict={self.x: self.refX})
Z = uh.reshape((self.refn, self.refn))
uhref = self.exactsol(self.X, self.Y)
self.rl2.append( np.sqrt(np.mean((Z-uhref)**2)) )
########## record loss ############
```

The problems definition is made by defining derived class of NNPDE where the methods must be defined:

```
self.f, self.exactsol and self.tfexactsol.
```

List of Figures

2.1 Feedforward dense neural network structure 20
2.2 Scheme of neurons functioning 22
2.3 Plots of the most common activation functions 24
2.4 Scheme of backpropagatio neurons 25

3.1 Coupled neural networks structure 42
3.2 Data sampling: boundary points (°) and interior points (×) 43

5.1 TC.E1: Measures of error as a function of the iteration counts. 62
5.2 TC.E1: Computed and exact solutions after 0,500 and 1000 iterations. . . 62
5.3 TC.E2: Measure of error as a function of the iteration counts. 64
5.4 TC.E2: Computed and exact solutions after 0, 500 and 1000 iterations. . . 64
5.5 TC.E2 with smooth correction: Approximation error as a function of the iteration counts 65
5.6 TC.E2 with smooth correction: Computed and exact solutions after 0, 500 and 1000 iterations. 66
5.7 TC.E3: Measure of error as a function of the iteration counts 67
5.8 TC.E3: Computed and exact solutions after 0, 500 and 1000 iterations. . . 68
5.9 TC.E1: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3. 70
5.10 TC.E2: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3. 70
5.11 TC.E2 with smooth correction: Measures of error as a functon of the iteration counts when varying the number of layers among 1, 2 and 3. 71
5.12 TC.E3: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3. 71
5.13 TC.E1: Measures of error as a function of the iteration counts when varying the number of neurons. 72
5.14 TC.E2: Measures of error as a function of the iteration counts when varying the number of neurons. 73

5.15 TC.E3: Measures of error as a function of the iteration counts when varying the number of neurons. 73
5.16 TC.E1: Measures of error as a function of the iteration counts when varying the learning rate. 75
5.17 TC.E2: Measures of error as a function of the iteration counts when varying the learning rate. 75
5.18 TC.E3: Measures of error as a function of the iteration counts when varying the learning rate. 76
5.19 TC.E3 with tuned hyperparameters: Measures of error as a function of the iteration counts. 76
5.20 TC.E1b: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3. 78
5.21 TC.E2b: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3. 78
5.22 TC.E3b: Measures of error as a function of the iteration counts when varying the number of layers among 1, 2 and 3. 79
5.23 TC.E4: Measures of error as a function of the iteration counts. 80
5.24 TC.E4: Computed and exact solutions after 0, 500 and 1000 iterations. . . 80
5.25 TC.E5: Approximation error as a function of the iteration counts. 81
5.26 TC.E5: Computed and exact solutions after 0, 500 and 1000 iterations. . . 81
5.27 TC.E6:Approximation error as a function of the iteration counts. 82
5.28 TC.E6: Computed and exact solutions after 0, 500 and 1000 iterations. . . 82
5.29 TC.E5 solved with tuned hyperparameters: Approximation error as a function of the iteration counts. 84
5.30 TC.E5 solved with tuned hyperparameters: Computed and exact solutions after 0, 500 and 1000 iterations. 84
5.31 TC.E6 solved with tuned hyperparameters: Approximation error as a function of the iteration counts. 85
5.32 TC.E6 solved with tuned hyperparameters: Computed and exact solutions after 0, 500 and 1000 iterations. 85

6.1 TC.P1: Computed and exact solutions after 0, 400 and 1000 iterations for $t = 0,\ t = 0.5,\ t = 1$ 88
6.2 TC.P1: Computed error as a function of the iteration counts. 89
6.3 TC.P2: Computed and exact solutions after 0, 400 and 1000 iterations for $t = 0,\ t = 0.5,\ t = 1$ 90
6.4 TC.P2: Computed error as a function of the iteration counts. 90

6.5 TC.P3: Computed and exact solutions after 0, 400 and 1000 iterations for $t = 0,\ t = 0.5,\ t = 1$ 91
6.6 TC.P3: Computed error as a function of the iteration counts. 92
6.7 TC.P4: Computed and exact solutions after 0, 400 and 1000 iterations for $t = 0,\ t = 0.5,\ t = 1$ 93
6.8 TC.P4: Computed error as a function of the iteration counts. 93
6.9 TC.P5: Computed and exact solutions after 0, 400 and 1000 iterations for $t = 0,\ t = 0.5,\ t = 1$ 94
6.10 TC.P5: Computed error as a function of the iteration counts. 95
6.11 TC.H1: Computed error as a function of the iteration counts. 96
6.12 TC.H2: Computed error as a function of the iteration counts. 97

List of Tables

5.1 TC.E1: Performance of the ANN-based method stopped after 400 training iterations. 61
5.2 TC.E1: Performance the of Finite Element Galerkin method. 61
5.3 TC.E2: Performance of ANN-based method on a regular grid, stopped after 400 training iterations. 63
5.4 TC.E2: Performance of the ANN-based method stopped after 400 training iterations, data gathered around the peak. 63
5.5 TC.E2: Performance of the Galerkin Finite Element method. 64
5.6 TC.E3: Performance of the ANN-based method stopped after 400 training iterations. 67
5.7 TC.E3: Performance the of Finite Element Galerkin method. 67

Acknowledgements

I would like to thank my advisor, Professor Paola Francesca Antonietti, for the precious recommendations and support during the research for my thesis and for showing me many interesting unknown aspects and applications of this subject. I am also grateful for all the suggestions not only about this work but also with a view to possible future extensions, for the trust laid on me and finally because her passion for this topics has renewed my motivation in the study of mathematics.
Also thanks to my family and friends for always believing in me.