POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Towards Adaptive PINNs For PDEs: A Numerical Exploration

Tesi di Laurea Magistrale in
Computational Science & Engineering - Mathematical Engineering

Riccardo Patroni, 10498009

Advisor:
Prof. Marco Verani

Co-advisors:
Prof. Edie Miglio

Academic year:
2021-2022

Abstract: Among several other fields related to applied mathematics, *Deep Neural Networks* (DNNs) have been recently deployed to the study and analysis of *Partial Differential Equations* (PDEs), emerging as the so-called *Physics-Informed Neural Networks* (PINNs). In this work, we focus our attention on their performance in approximating the exact solution of a general and significative set of scalar PDEs. During our discussion, we should expect to appreciate their renowned qualities and face some of the notorious drawbacks that are still responsible for their limited use in real-world applications. One of their major disadvantages consists in the actual problem of choosing the most convenient architecture to be used for our purposes, which represents the heart of our discussion. After a thorough analysis of the results concerning the performances related to the *basic* version of the PINN, where we study the relationship between the accuracy of the trained networks and their structural characteristics consisting in the number of hidden layers, the amount of neurons per layer and the cardinality of the training set, we provide a general overview of some innovative techniques, found in literature, that aim at enhancing this numerical tool. Finally, we propose the implementation and the relative heuristic justification of an attempted *adaptive* scheme, an evolution of the plain basic framework formerly introduced, that jointly employs the so-called *Growing Method* alongside the *Residual Adaptive Refinement* technique. The computational tool exploited for the construction of all models relies on a newly-developed Python library for the resolution of scalar PDEs over elementary hyper-rectangular domains by means of an ADAM - LBFGS optimizer (the related code is publicly available at: `https://github.com/patropolimi/Thesis`).

Key-words: Artificial Intelligence, Neural Networks, Partial Differential Equations, Adaptivity

1. Introduction

In the ever-expanding world of applied sciences, Partial Differential Equations (PDEs) have recently become the main character on the stage. These mathematical entities have arisen from our ability to gather, model and translate into a rigorous and consistent framework all the fundamental principles and the experimental evidence coming from the observation of natural phenomena. Appearing in countless fields of expertise, ranging from biology to finance, passing through physics, chemistry and statistics, the road-map that ideally leads to their general resolution is nowadays considered to be the most valuable of all the hunted treasures in science. Nevertheless, although PDEs still remain extremely hard to study and analyze in their most general form, many great advancements have been achieved throughout the last centuries in different specific areas (see [8, 19, 49]). Over the years, however, it has also rapidly become clear that the realization of a general methodology able to provide the closed analytical expression for the solution of any conceivable PDE is, put mildly, utopian (see [17, 22, 41, 50, 70]). In order to overcome this fundamental obstacle, scientists and engineers have developed and employed several numerical methods to find reliable approximations of the target, on top of which we find the famous *Finite Element Methods* (FEMs) (see [21]). Additionally, the contemporary growth of the nowadays ubiquitous Artificial Neural Networks (ANNs) have recently offered an alternative path in the search for a reliable tool to be used for this field of application (see [1, 6, 35–38, 53, 54, 65, 67]). This specific mathematical structure, belonging to the vast category of *Machine Learning* methodologies, is generally accompanied by a user-defined loss function which undergoes a minimization procedure through a proper optimization algorithm (see [5]). The main idea that greatly enhances its use consists in choosing a cost functional where both the contributions coming from PDE residuals (inside the computational domain) and boundary conditions (on the edges) are penalized, implying their *implicit* imposition during optimization as for the *unsupervised learning* methodologies. Thanks to this newborn technique, which conserves the soul of the so-called *black-box* methods, we eliminate (or, at least, drastically reduce) the need to build and install physical sensors for the direct measurement of target solution values over an arbitrarily chosen set of points that would be subsequently emplaced in the loss function expression to force the network to learn the desired behavior from pure data. Other than being outdated and extremely expensive, such an approach would be very naive and incomplete as well, in the sense that precious information provided by the equation and boundary conditions of the governing PDE would not be fully taken into account and/or exploited. Regarding the aforementioned innovative approach, the most visible drawbacks clearly reside in the computational overhead attached to the need of calculating the network function partial derivatives that appear in the studied PDE and, even more importantly, in the fact that, having to deal with a loss function which is generally not convex, we cannot have any guarantee about the reliability of the obtained numerical solution, which may well express one of its local (but not global) minima. We have at our disposal, nonetheless, a very convenient fix for the former issue: the Automatic Differentiation toolbox (AD) in [4], originally created alongside the much general class of compositional functions. The overall structure combining all the mentioned mathematical and computational elements is currently known as Physics-Informed Neural Network, the protagonist of our work (see [37]). In the following, we shall proceed by exploiting a specific subclass of Artificial Neural Networks, the so-called Deep Neural Networks (DNNs). After a brief introduction dedicated to the exposition of their essential features, we will linger on the notorious *Optimal Architecture Problem* coming along with their usage, eventually presenting two attempted proposals that aim at resolving it. Such an issue, born with DNNs, clearly remains an open problem for PINNs as well and represents the inspiring ground for this work. A subsequent deeper explanation about the inner workings of the PINN structure, accompanied by the presentation of some related formal results that have been recently discovered in this framework, is followed by the essential core of our work, whose aim is to explore in detail the behavior of PINNs, in terms of performance, in relation to their structural and architectural properties: number of hidden layers, amount of neurons per layer and cardinality of the training set. At this point it is crucial to disambiguate the meaning of a few important concepts: whenever we refer to basic PINNs, we intend those plain Deep Neural Networks whose structure remains fixed throughout the whole optimization process, while the so-called adaptive PINNs consist of those architectures that possess the ability to modify their properties during the learning phase as, for instance, changing the number of hidden layers with which they are embedded. In this work, the terms network, model and (in proper contexts) architecture will be used as synonyms. Continuing with the introduction of the path followed for this project, in Section 3 and 4 we present the results obtained analyzing a selected set of PDEs through the just explained basic and adaptive versions of the PINN proposed in this work, dedicating the main focus of our study to the relationship between performance and architectural properties of the models. Finally, consistently with the gathered data, we will draw our conclusions and suggest a series of possible further developments for future studies on this subject. In order to be clear from the beginning and avoid any possible misunderstanding, we premise that the content of this work must not be intended as a suggestion to pursue an alternative path to the consolidated FEM technique for the approximate resolution of PDEs. The latter has indeed proven to be a successful, consistent and reliable framework that, at least in this context, is well ahead of any other proposed methodology.

1.1. Objectives & Perspectives

The primary objective of our work is to study, verify and reproduce the performance of the PINN mathematical structure for a variety of cases that have already been analyzed in literature (as in [37, 49, 65]), and continue this journey on some other instances of physical interest. In this framework, the main part of our discussion will be dedicated to the so-called *sensitivity* analysis: this study essentially consists in comparing, over the same differential problem, the performance of differently-structured networks. In doing so, we will try to identify the most influential architectural features and hyper-parameters of PINNs, basing our conclusions on the L^2 relative error criterion. Because each trained model started with a random initialization, we launched three attempts for every different set of specifications (later referred to as first, second and third instances). In addition to this investigation, we also draw a few experimental considerations about the *convergence* properties of PINNs (as [60]). The last chapter of our journey will be devoted to the proposal of a trivial adaptive scheme for PINNs, an algorithm that merges two other techniques related to this field: Growing and Residual Adaptive Refinement (see [37]). In order to hint the effective exploration concerning the reliability of this experimental method, we will apply it to a few test cases and draw our very first conclusions on the basis of their results. Regarding this topic, the interested reader may even decide to pursue such road and explore other possible ways to develop this new framework, possibly integrating other techniques found in literature inside the presented algorithm.

2. Technical Background

The present section is dedicated to the exposition of the essential machinery that constitutes the basis of our work. First of all, we propose a general introduction to *Artificial Neural Networks* and Deep Neural Networks in particular, with a glance of the context in which they were born and a description of the key phases where they play a major role. Afterwards, it is presented a still debated and unresolved issue concerning the entire world of *Artificial Intelligence*, the so-called *Optimal Architecture Problem.* Finally, we explore the mathematical structure, detailed characteristics and key motivations behind the leading actor of our journey, the PINN.

2.1. Artificial Neural Networks

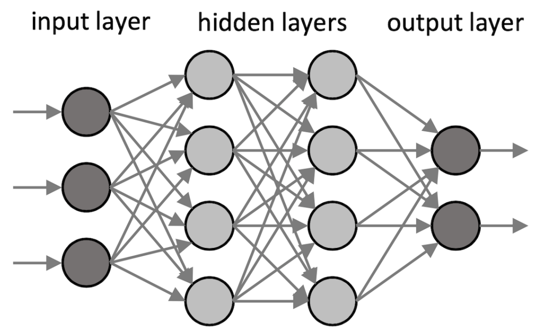
Developed in the context of classification problems, Artificial Neural Networks represent the core of many modern Machine Learning methodologies. Their flexibility allows the usage of this mathematical structure in a wide range of applications, especially where a great amount of data has to be constantly elaborated, updated and correctly interpreted. From an abstract standpoint, a Deep Neural Network is a particular compositional function formed by several layers made of fundamental building blocks, called neurons, that simulate the behavior of a human brain by communicating through a series of connections. The latter are the essential parameters of the model, and consist of a group of weights and activation thresholds (or biases) controlled by the user. During a successful *training phase*, the network assigns a proper combination of values to these parameters, enabling the model to reliably reproduce the phenomenon of our interest. Similarly to real neurons, the units of the network modulate their output by applying a proper (typically nonlinear) *activation function* to their input signal. A schematic visualization of a classical Deep Neural Network architecture is shown in Figure 1, where we also represent the internal structure and all the single components of a virtual neuron. In synthesis:

$$\mathcal{N} : R^m \to R^n, \quad \mathcal{N} = \mathcal{N}_{\boldsymbol{W}}(\boldsymbol{x})$$

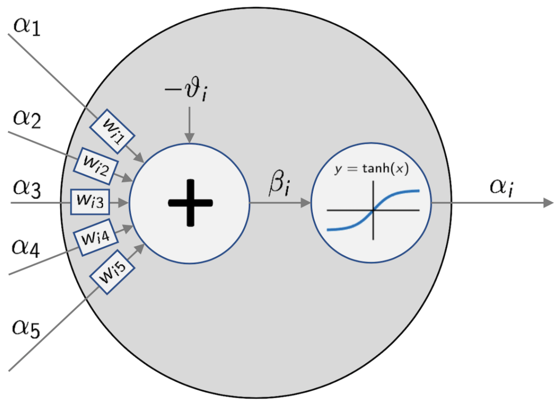
In the expression above, $\boldsymbol{x}$ is the m-dimensional input of the DNN $\mathcal{N}$, while $\boldsymbol{W}$ encodes the collection of weights and biases for the given fixed architecture. In the most general case the network maps the elements of the domain into n-dimensional output vectors. An alternative and more explicit functional formulation, where we particularly emphasize the compositional structure of the Neural Network and the role of its activation function ρ, can be written as follows (from [49]):

$$\mathcal{N}(\boldsymbol{x}) = T_L(\rho\ (T_{L-1}(\dots \rho\ (T_1(\boldsymbol{x}))))),$$

where $T_l(\boldsymbol{y}) = \boldsymbol{A}_l\boldsymbol{y} + \boldsymbol{b}_l$ for every $l \in \{1, \dots, L\}$. In these expressions, L indicates the number of hidden layers plus one (the output layer), $\boldsymbol{A}_l \in \mathbb{R}^{N_l \times N_{l-1}}$ is the weight matrix that maps the output of the $l-1$-th layer (of dimension N_{l-1}) into the successive layer's input (with dimension N_{l-1}) and $\boldsymbol{b}_l \in \mathbb{R}^{N_l}$ represents the l-th layer bias vector. Setting d as the network's input dimension, we necessarily have that $N_0 = d$. Notice that here ρ is intended to be applied coordinate-wise. In Figure 2 we have depicted, in a neighborhood of the origin, the two activation profiles that have been extensively exploited throughout our work. The introduced mathematical tool belongs to the evermore thriving category of *black-box* techniques, typically employed to predict the outcome of specific phenomena that are, on one hand, characterized by an unknown underlying functioning mechanism, but nevertheless considered to be scientifically addressable. The most relevant advantage carried by this methodology is not limited to the fact that it does not necessarily require any *a priori* insight on the studied process, but also in that it may even lead to the discovering of some hidden dynamics which can be useful for the consequent development of a mathematical theory that reproduces the observed behavior. It is by combining a great elasticity with a cheap computational cost that DNNs have entered a wide range of real applications in the last decades. As we will discuss later for a more specific case, the sudden birth and rapid evolution of this scientific discipline has been accompanied by the production of several rigorous results concerning their performance (see [1, 35, 37, 49]). The latter have proven to be crucial in taking the first steps towards a better understanding of the positive features and possible limitations connected to this technique.

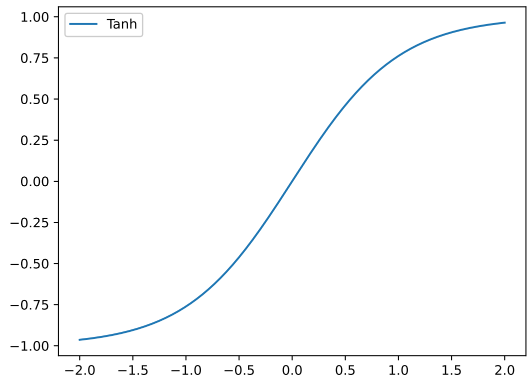


Structure of a DNN with two hidden layers composed by four neurons each.

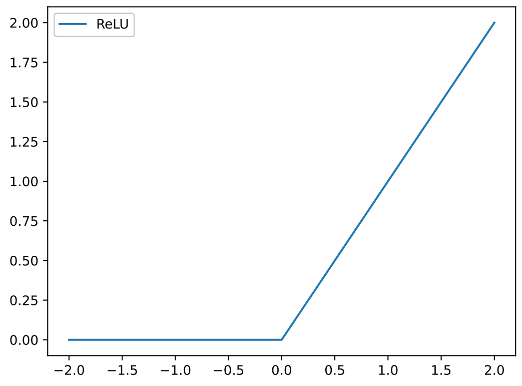


Structure of the i-th neuron: w_{ij} are the weights and θ_i is the activation threshold.

Figure 1: General representation of a Deep Neural Network. See [56].



(a) Tanh $(x) = \sinh(x)/\cosh(x)$.



(b) ReLU $(x) = \max\{0; x\}$.

Figure 2: Plots of the hyperbolic tangent (left) and ReLU (right) activation functions over $[-2, 2]$.

2.1.1 Training Phase

The training (or *learning*) phase of a *Neural Network* constitutes the heart of Machine Learning. This is the process that ultimately allows the constructed model to replicate and predict the essential features of the phenomenon that we are analyzing. In the branch of *supervised learning*, this procedure comes down to the minimization of a cost functional of the form (see [5, 56]):

$$J = \frac{1}{N} \sum_{i=1}^{N} \left\| \mathcal{N}_{\boldsymbol{W}}(\boldsymbol{x_i}) - \boldsymbol{y_i} \right\|^2,$$

where $\|\cdot\|$ denotes the euclidian norm. The cost functional J formally expresses the *average distance* of the true (observed) output $\boldsymbol{y}$ with respect to the prediction that the approximated model provides when *fed* with the relative input $\boldsymbol{x}$. This quantity is evaluated over a sample pool that is better known as *training set*, that may be depicted as the *training camp* where the model *learns* how to emulate the studied process. During this step, the network looks for an optimal combination of values for its parameters $\boldsymbol{W}$, in such a way that the resulting model is able to accurately reproduce the behavior of the input-output couples belonging to the training set. Since the birth of this field, a great variety of *learning methods* have been developed and published in literature (as in [5, 27, 48]). Without the need of exploring such details, we limit ourselves to the trivial consideration that each of these techniques carries its own advantages and drawbacks (see [64, 66]).

2.1.2 Testing Phase

Testing a Neural Network means evaluating its general performance. The importance of this step resides in the fact that the ultimate goal consists in the creation of a model which is able to *generalize* the knowledge acquired during the learning phase. Indeed, our final aim is to accurately predict not only the trend shown by the samples of the training set, but also and even more importantly to reveal the hidden features of any realistic and observable input-output couple. In other words, given a generic input $\boldsymbol{x}$, we would like our model to provide a reliable approximation $\mathcal{N}_{\boldsymbol{W}}(\boldsymbol{x})$ of the relative true output $\boldsymbol{y}$. The implicit rationale is that, during the training phase, we expect our network to grasp the underlying generic characteristics of the studied phenomenon. If this is the case, when tested on *unseen data*, the model will be able to approximately reproduce what would be measured through laborious and expensive experimental observations (see [62]). In order to test its performance, we employ the expression of J over a numerous and statistically significant dataset, better known as *test set.*

2.1.3 The Optimal Architecture Problem

Until now we never dwelt on the problem of choosing the size of the architecture to be used for our purpose. With no exaggeration, we might consider the latter as one of the major challenges affecting the whole mathematical branch of Artificial Neural Networks. In order to present the issue and give an intuitive explanation of what happens behind the scenes, let us give a look to Figure 3. In it, the black dots play the role of a noisy training set sampled from a parabola, while the red line indicates the approximated prediction of the constructed Neural Network. On the left, we see an exemplification of the so-called ***underfitting*** phenomenon, a trend that emerges when the network poorly fits both the training and the test samples. This problem always arises when the size of the model is too small, implying that the network does not possess the *capacity* needed to understand the *complexity* of the analyzed process. On the contrary, when the selected architecture happens to be excessively large with respect to the complexity of the studied phenomenon, the model typically shows the pattern on the right. This behavior, known as ***overfitting***, occurs when the network accurately fits the pattern presented by the training set and fails at representing the test set with a similar precision. In this scenario, the model tends to *memorize* the features (and the relative noise) of the input-output couples belonging to the training set, affecting its ability to generalize the acquired knowledge and consequently resulting in a low *train error* and a considerably higher *test error*. Only when the architecture has a nearly *optimal* number of free parameters can we fully appreciate the predictive power of the Neural Network. This situation is well depicted in the central plot: under these circumstances, the model is capable of describing both the training and test data with great precision. Unfortunately, the determination of the optimal capacity of the model is not straightforward, since it strongly depends on the case under study (see [9, 22, 25, 28, 44, 58]).

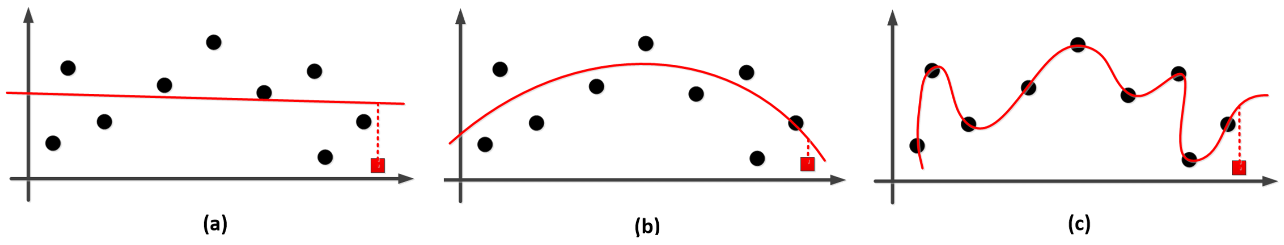


Figure 3: (a) Underfitting (b) Optimal fitting (c) Overfitting. See [56].

2.1.4 The Growing Procedure

Over the years, it has rapidly become clear that the concept of a complete mathematical theory that provides and formally justifies the correct dimension of a network for each possible problem lies well beyond any realistic scenario. It is in the aforementioned framework that the so-called Growing Methods were born. Their simple procedure, aimed at resolving the Optimal Architecture Problem, consists in starting with a small model and progressively increase its size until it reaches the right capacity for the description of the phenomenon of our interest. After a successful run, the network is expected to generalize the learned knowledge with a properly-sized architecture for the problem at hand. Part of the side benefits brought by this technique concern memory saving and the possibility, in some cases, of extracting unknown underlying rules that characterize the physics of the studied phenomenon (see [63]). Although in our analysis we mainly deal with PINNs, for which the concepts of underfitting and overfitting shall be reinterpreted with a slightly different perspective, we believe that it was worth mentioning these two concepts in the previous subsection for clarity and completeness, prior to the introduction of the Growing procedure. This method will then be thoroughly examined in the final section, where it has been encapsulated in a newly-developed adaptive scheme devised for the learning phase of PINNs.

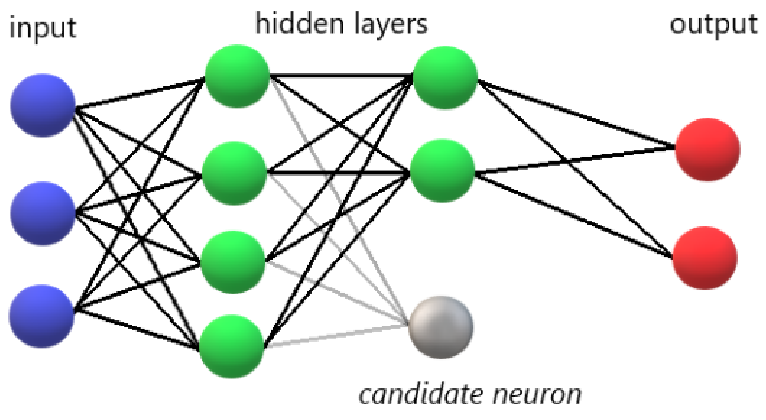


Figure 4: Growing step where a new candidate neuron is inserted into the last hidden layer. See [45].

In Figure 4 we present one among the essential operational modes through which we might attempt to grow the architecture in use for the resolution of the problem at hand. This operative step, usually embedded with either a formal mathematical procedural justification or just a heuristical reasoning, is generally accompanied by a specific criterion that identifies the most convenient way to enlarge the size of our structure (see [12, 14, 23, 31, 34, 40, 55]). For the particular instance illustrated below, we evaluate the convenience of adding a candidate neuron to the last hidden layer of the Neural Network: in the case where such modification is considered to be worthy and effective, the newly-drawn gray connections are properly initialized alongside the creation (and proper initialization) of the related output weights linked to our additional unit. The mentioned step, in general, has to be iteratively repeated until a proper stopping criterion is satisfied. At this ending point, we finally expect to possess a much more reliable structure for the interpretation of the studied phenomenon.

2.1.5 The Pruning Procedure

For completeness, we also cite the alter-ego of the just introduced Growing procedure: the *Pruning* technique. The latter follows the same ideal concept of the former, but acts with the opposite rationale.

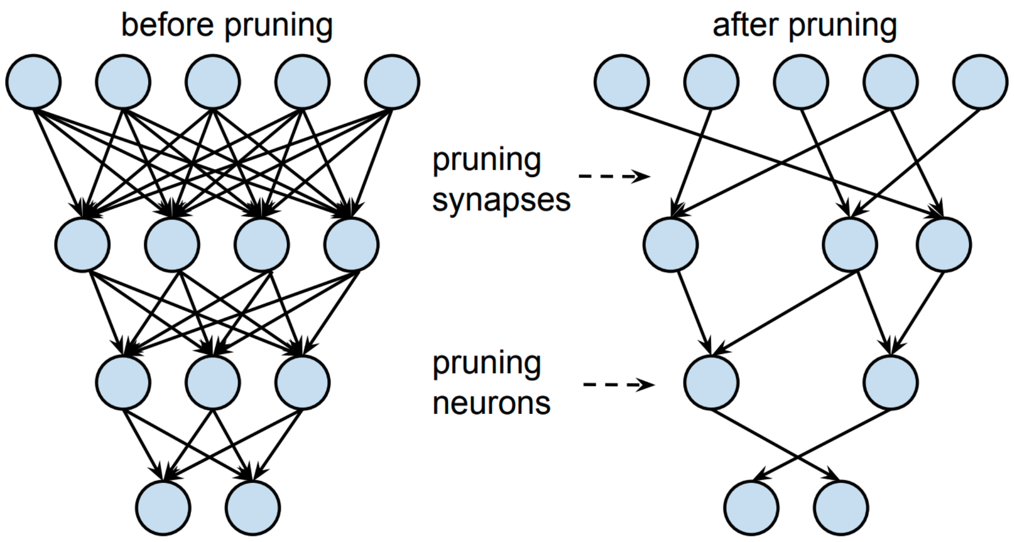


Figure 5: Generic depiction of the two main pruning approaches: elimination of single synapses (weight-by-weight cancellation) and deletion of entire units (neuron-by-neuron elimination). See [18].

This procedure, starting with an overestimation of the ideal size needed for our network, operates by iteratively removing subsets of internal connections (according to a chosen criterion) until it reaches an optimal architectural state for the prediction of the phenomenon under study. In light of all considerations made, we might say that Growing Methods seek the most convenient size of the model starting from a situation where underfitting occurs, while Pruning techniques try to reach the same goal by shrinking the network to cancel the effects of overfitting. Guided by the experimental evidence showing that, at least in the mathematical framework concerning the numerical resolution of PDEs through DNNs, larger architectures are not necessarily able to provide better fits than smaller models even on extremely smooth functions, in this work we decided to pursue and test only the first of these two methodologies. To the current author's knowledge, neither of these two techniques have ever been exploited in this context before. Figure 5 shows a symbolic but nonetheless thorough illustration of the two most important pruning approaches that can be found in literature (see [7, 16, 20, 30, 32, 47, 51]). Every elimination technique is based on a rigorous mathematical formulation or, at least, on a heuristical criterion that is used to pinpoint the physical entities that should be conveniently deleted from the actual structure of the network, whether they are a subset of single synapses or entire units, whose elimination consists in casting away all the connections directly linked to them. These deletions are performed by setting to zero the interested weights, along the ones that are consequently cut-off from the network by the former elimination (namely all synapses that can no longer contribute to the input-output relation expressed by the model). Even though the presented technique will not enter our adaptive algorithm proposal, we provide an overview of some important related procedures (with their formal or heuristical background) in Appendix B.

2.2. Physics-Informed Neural Networks

A vast multiplicity of Partial Differential Equations present a strong form expression that contains several terms with an arbitrary degree of derivation. On the other hand, Deep Neural Networks carry along a very important side-benefit of their usage in this specific context: automatic differentiation (see [4]). This revolutionary computational tool represents the key that opened the door to the resolution of PDEs with Machine Learning techniques. The great advantage coming from the combination of DNNs and AD resides in the exact (up to machine precision) and relatively cheap computation of any network derivative with respect not only to its inputs but also to all the model parameters. However, finding the solution to a Partial Differential Equation with a Deep Neural Network requires an approach that has little in common with all the other problems that involve the usage of this technique. In fact, the target function that we want to approximate is unknown and cannot be provided explicitly. It is therefore meaningless, in this context, to talk about the concepts of training and test set in the form that we previously presented. Hence, these will be accordingly substituted with their proper expression for this framework (from [37]). As a consequence, we have to accordingly provide a new expression for the cost functional that undergoes the minimization procedure during the learning phase of the models. Physics-Informed Neural Networks were born from a seething background, leveraging on the need of a new mathematical tool for the resolution of PDEs. Their essential innovation consists in combining the flexibility of Deep Neural Networks with a cost functional that implicitly provides, as in the branch of unsupervised learning, the target solution through the imposition of a null PDE residual inside the computational domain, while forcing the model to satisfy the known boundary conditions on its edges. The pure penalization of the strong form of the PDE residual has a relatively cheap cost thanks to the employment of the AD toolbox, as we anticipated. In this context, differently from the FEM framework, we need not transform the problem in any of the so-called *weak formulations* (also known as *integral formulations*) and we might as well resolve it without any further manipulation of its characterizing expression (diversely from [1, 35]).
Consider now the general form of a scalar Partial Differential Equation, possibly parametrized by a vector of coefficients $\boldsymbol{\lambda}$ for the solution $u(\boldsymbol{x})$ defined on a domain $\Omega \subset R^d$:

$$f\left(\boldsymbol{x}; \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 u}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right) = 0, \qquad \boldsymbol{x} \in \Omega,$$

while all boundary conditions on the edges are grouped under the following notation:

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

where $\mathcal{B}(u, \boldsymbol{x})$ could be Dirichlet, Neumann or periodic boundary conditions. For time-dependent problems we consider the time coordinate as a special component of $\boldsymbol{x}$, so that Ω represents the entire spatio-temporal domain. In this case, the initial condition can be simply treated as a special type of Dirichlet boundary condition. In the following, we show both a schematic and visual representation of the PINN algorithm (see [37]).

Algorithm 1 Solving PDEs using PINNs

```
1: Construct a network $\hat{u}(\boldsymbol{x}; \boldsymbol{W})$ such that all its parameters are encoded in $\boldsymbol{W}$
2: Specify the two training sets $\mathcal{T}_f$ and $\mathcal{T}_b$ for the equation and boundary conditions
3: Employ a proper cost functional that includes the sum of both the $L^2$ norm of the PDE residuals
   inside the domain and the contribution coming from the boundary conditions on the edges
4: Train the model to find a suitable combination of parameters $\boldsymbol{W}^*$ by minimizing the cost functional
```

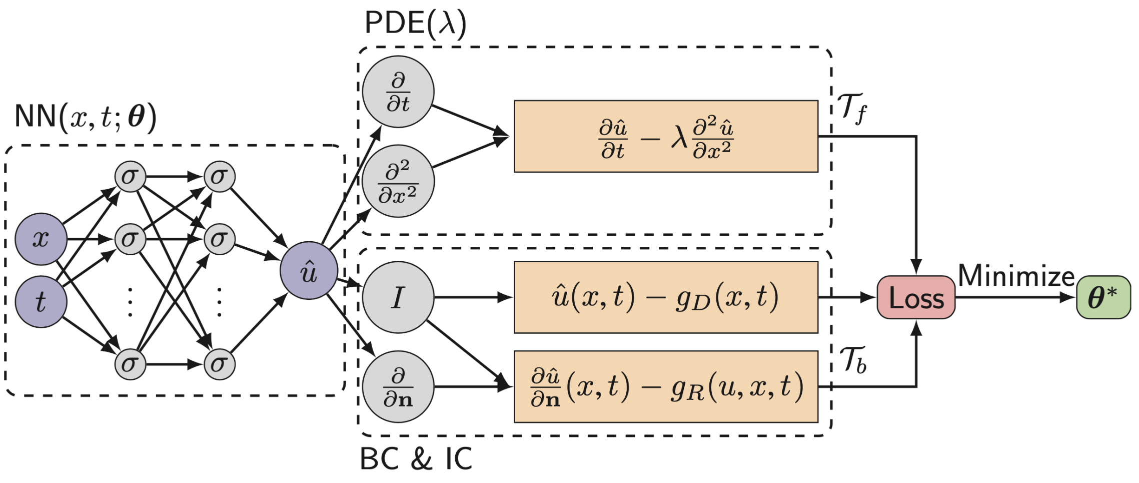
One convenient expression for our cost functional reads as follows:

$$\mathcal{L}\left(\boldsymbol{W};\mathcal{T}_f,\mathcal{T}_b\right)=\mathcal{L}_f\left(\boldsymbol{W};\mathcal{T}_f\right)+\mathcal{L}_b\left(\boldsymbol{W};\mathcal{T}_b\right),$$

where the explicit formulation of the written terms is provided as:

$$\begin{aligned}\mathcal{L}_f\left(\boldsymbol{W};\mathcal{T}_f\right)&=\frac{1}{|\mathcal{T}_f|}\sum_{\boldsymbol{x}\in\mathcal{T}_f}\left\|f\left(\boldsymbol{x};\frac{\partial\hat{u}}{\partial x_1},\ldots,\frac{\partial\hat{u}}{\partial x_d};\frac{\partial^2\hat{u}}{\partial x_1\partial x_1},\ldots,\frac{\partial^2\hat{u}}{\partial x_1\partial x_d};\ldots;\boldsymbol{\lambda}\right)\right\|_2^2,\\ \mathcal{L}_b\left(\boldsymbol{W};\mathcal{T}_b\right)&=\frac{1}{|\mathcal{T}_b|}\sum_{\boldsymbol{x}\in\mathcal{T}_b}\left\|\mathcal{B}\left(\hat{u},\boldsymbol{x}\right)\right\|_2^2,\end{aligned}$$

in which $\|\cdot\|_2$ is an alternative notation for the euclidian norm. $\mathcal{T}_f$ and $\mathcal{T}_b$ are the training sets containing the scattered points on which we evaluate the PDE residuals and impose the boundary conditions during the learning phase of the model, respectively. We subsequently expect from our network that, upon a successful training procedure, it has learned to accurately reproduce and satisfy both the PDE and the boundary conditions not only over the training sets but also everywhere else in the domain. In Figure 6 we illustrate an essential visualization of a Physics-Informed Neural Network applied to the resolution of the simple one-dimensional Poisson equation coupled with a set of mixed boundary conditions.



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

Figure 6: Complete representation of the structure for a Physics-Informed Neural Network applied to the resolution of the 1D Poisson equation coupled with mixed boundary conditions. Taken from [37].

Considering that the loss function is typically highly nonlinear and nonconvex with respect to the model parameters, it is generally advisable to minimize its value by using gradient-based optimizers such as *Gradient Descent*, *ADAM* and *LBFGS* (see [5, 27, 48]). The former two are *first-order* methods, while LBFGS has a *superlinear-order* of convergence since it also involves an approximate computation of the second-order derivatives of the loss function. Unlike other traditional numerical methods, for PINNs there is generally no guarantee of unique solution. This issue stems from the fact that the latter are obtained through the minimization of nonconvex optimization problems, which in general do not ensure any uniqueness result. We also note that PINNs may even converge to different realizations if starting from differently-initialized architectures. Thus, a common strategy consists in training several independent versions of the same PINN, each prompted with a different random initialization. At the end of their learning phase, we obviously choose the network presenting the smallest loss function value. In order to achieve a good level of accuracy, we not only need to exploit a suitable optimizer but we also have to properly tune all hyper-parameters of the model, such as the network size and the number of residual points.
One of the notorious drawbacks attached to PINNs is better known as *Frequency Principle* (see [70]), an issue that consists in the bias of learning the PDE's solution from low to high frequencies. To overcome this problem and, more generally, in the attempt of improving the performance of PINNs, many different adaptive schemes have been introduced in literature. A non-exhaustive list includes the usage of adaptive activation functions [24], the employment of a soft-attention mechanism through the proper variation of the multiplying coefficients for each addend of the loss function components [39], the so-called Residual Adaptive Refinement method [37] and other time-adaptive approaches for the resolution of time-dependent PDEs [67]. Furthermore, some of these

papers have also shed light on additional pathologies from which PINNs seem to suffer (see [64, 66]). Finally, it is worth recalling that PINNs have not been deployed only to the direct search for the approximation of PDE solutions, but that one of the most promising among their applications concerns their use in the resolution of inverse problems, as in [37]. In the following two subsections, we report a series of useful considerations concerning the results of approximation theory for PINNs and a brief comparison against FEMs (see [37, 49]).

2.2.1 Approximation Theory & Error Analysis

At this point we believe it is worth recalling that PINNs are nothing else but Deep Neural Networks coupled with a cost functional that enforces the underlying solution of the studied phenomenon in an implicit manner. Their approximation potentialities, features and properties are, therefore, identical to the latter. It is for this reason that all the formal results related to DNNs can be equivalently extended to our framework. One of the fundamental questions related to PINNs is whether there exists a network which is able to uniformly satisfy both the PDE equation inside the computational domain and the relative boundary conditions on its edges. In other words, we are interested in understanding under which circumstances we can construct a model that simultaneously approximates the exact solution and its partial derivatives. In order to address the mentioned problem, we need to introduce some useful notation: let $\mathbb{Z}_+^d$ be the set of d-dimensional nonnegative integers. Then, for $\boldsymbol{m} = (m_1, \dots, m_d) \in \mathbb{Z}_+^d$, we set $|m| := m_1 + \dots + m_d$, and define:

$$\mathcal{D}^{\boldsymbol{m}} := \frac{\partial^{|\boldsymbol{m}|}}{\partial x_1^{m_1} \dots \partial x_d^{m_d}}.$$

We say that $f \in C^{\boldsymbol{m}}(\mathbb{R}^d)$ if $\mathcal{D}^{\boldsymbol{k}} f \in C(\mathbb{R}^d)$ for all $\boldsymbol{k} \leq \boldsymbol{m}$, with $\boldsymbol{k} \in \mathbb{Z}_+^d$. The following result (from [37]) holds:

Theorem 2.1. *Let $\boldsymbol{m}^i \in \boldsymbol{k} \in \mathbb{Z}_+^d$, $i = 1, \dots, s$, and set $m = max_{i=1,\dots,s}|\boldsymbol{m}^i|$. Assume $\sigma \in C^m(\mathbb{R})$ and that σ is not a polynomial. Then, the space of single hidden layer networks:*

$$\mathcal{M}(\sigma) := span\left\{\sigma\left(\boldsymbol{w} \cdot \boldsymbol{x} + b\right) : \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}\right\},$$

is dense in

$$C^{\boldsymbol{m}^1,\dots,\boldsymbol{m}^s}(\mathbb{R}^d) := \cap_{i=1}^s C^{\boldsymbol{m}^i}(\mathbb{R}^d),$$

which means that, for any $f \in C^{\boldsymbol{m}^1,\dots,\boldsymbol{m}^s}(\mathbb{R}^d)$, any compact set $K \subset \mathbb{R}^d$, and any $\epsilon > 0$, there exists a network $g \in \mathcal{M}(\sigma)$ satisfying the relation:

$$\max_{\boldsymbol{x} \in K} |\mathcal{D}^{\boldsymbol{k}} f(\boldsymbol{x}) - \mathcal{D}^{\boldsymbol{k}} g(\boldsymbol{x})| < \epsilon$$

for all $\boldsymbol{k} \in \mathbb{Z}_+^d$ such that $\boldsymbol{k} \leq \boldsymbol{m}^i$ for some i.

The presented theorem shows that the class of Deep Neural Networks with a single layer are able, if provided with a sufficiently large number of neurons, to simultaneously and uniformly approximate any function with its partial derivatives. Nonetheless we must remember that, due to the finite memory of any computational tool, in reality we have physical constraints on the maximum number of units that can simultaneously belong to a model. Assume now $\mathcal{F}$ to be the family of all functions that can be represented by the chosen DNN architecture, and denote with u the exact solution that we want to approximate. Since it is extremely unlikely that the latter belongs to $\mathcal{F}$, we define $u_{\mathcal{F}} = \arg\min_{f \in \mathcal{F}} \|f - u\|$ as the best approximation of u among this class of functions. Define now $\mathcal{T} = \mathcal{T}_f \cup \mathcal{T}_b$. Since we train our DNN only over the points in $\mathcal{T}$, we also set $u_{\mathcal{T}} = \arg\min_{f \in \mathcal{F}} \mathcal{L}\left(f; \mathcal{F}; \mathcal{T}\right)$ as the network in $\mathcal{F}$ whose loss is at global minimum. In any realistic scenario, however, the optimization algorithm returns $\tilde{u}_{\mathcal{T}}$, an approximation of $u_{\mathcal{T}}$. We can therefore provide a symbolic expression for the upper-bound of the total approximation error as a sum of three independent contributions. Denoting the total error with ξ, we exploit the triangle inequality to write:

$$\xi := \|\tilde{u}_{\mathcal{T}} - u\| \leq \|\tilde{u}_{\mathcal{T}} - u_{\mathcal{T}}\| + \|u_{\mathcal{T}} - u_{\mathcal{F}}\| + \|u_{\mathcal{F}} - u\|.$$

The first term, also called optimization error, inevitably comes from the learning procedure: it mainly depends on the loss function landscape and the general training settings. The second term, the so-called generalization error, is determined by the total number and location of the training points in $\mathcal{T}$, as well as by the capacity of the family $\mathcal{F}$. Finally we have the approximation error, which measures how closely $u_{\mathcal{F}}$ can approximate the target solution u. In light of Theorem 2.1, we know that networks with larger size have smaller approximation errors. On the other hand, a wider architecture could lead to a higher generalization error. Such a balance is better known as *bias-variance trade-off*, and we generally say that overfitting occurs when the generalization error dominates the other terms. Despite some very recent forward steps that have been made in the estimation of rigorous bounds for the latter (see [43]), we are still far from a complete theoretical framework that provides a control on the total error for PINNs. More generally, the quantification of these three errors for the much wider mathematical branch of supervised learning is still an open research area (see [42, 71]). Still related to the subject of approximation theory, we introduce below another theorem which will be later referred to (from [49]). Such a result becomes useful when we compare the performances of shallow versus deep networks for the pure approximation problem of a particular class of functions, the so-called *saw-teeth* profiles, highlighting the essential role of depth for this specific set of targets.

Theorem 2.2. *Assume $\mathcal{N}$ to be a shallow network with one-dimensional input and output layers, embedded with the ReLU activation function and N neurons:*

$$\mathcal{N} \in \mathcal{S}(\sigma) := \left\{ \sum_{i=1}^{N} z_i \cdot \sigma\left(w_i \cdot x + b_i\right) + q : \boldsymbol{z}, \boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^N, q \in \mathbb{R} \right\}.$$

Then, it holds that $\mathcal{N} : \mathbb{R} \to \mathbb{R}$ is a piece-wise linear map characterized by 2N linear pieces at most.

2.2.2 Comparison PINN - FEM

In this subsection we make a concise comparison between the main features and differences of PINNs and the most renowned tool used for the numerical resolution of PDEs, called FEM.

- In FEM the solution is approximated by a piece-wise polynomial function whose parameters (or, equivalently, whose values on the mesh grid) are to be determined, while with PINNs we construct a network as its surrogate model. The latter is parametrized by the weights and biases of the architecture.
- FEMs typically require the generation of a mesh and a proper weak formulation. PINNs, instead, are suited for a strong and totally mesh-free imposition of the problem: we can either choose to employ a uniform grid or use randomly scattered points for the construction of the training sets $\mathcal{T}_f$ and $\mathcal{T}_b$.
- The core step for FEMs consists in converting the PDE to an algebraic system and solving it using a direct or iterative technique. PINNs, on the other hand, embed the PDE and the relative boundary conditions into the loss function that subsequently undergoes optimization, typically with a gradient-based method.

3. Basic PINN Results

This section, which represents the core of our work, will be entirely devoted to the thorough study of the results relative to the networks trained with the basic version of the PINN, in which all architectural features remain fixed throughout the entire learning procedure. Firstly, in subsection 3.2, we will dig into the detailed presentation of the outcomes concerning the so-called sensitivity analysis, which will almost entirely cover the content of this chapter by itself. We recall that such study essentially consists in discovering the performance patterns that emerge in relation to the main architectural hyper-parameters of the models, which are: the number of hidden layers, the amount of neurons per layer and the cardinality of the training set. Eventually, in subsection 3.3, we will dedicate our final considerations to the experimental evidence coming from the convergence analysis, performed on a few sample tests. Each of these subjects is further subdivided into three parts, named after the nature of the solutions therein analyzed: single-scale, multi-scale and generic.

3.1. Optimization Details

As we previously mentioned, all the PINNs that have been trained for this work exploit an optimization procedure based on the combined ADAM - LBFGS technique. The former has been deployed with its characteristic parameters set to their default values (see [27]), while the latter has been employed in its most renowned form, using at each iteration the *Two-Loops Recursion* algorithm and the so-called *Backtracking Line-Search* method to construct the optimal descent direction and the relative step coefficient, respectively. The implementative details of these procedures are omitted here: however, they can be accessed and consulted in [68], where their complete description is publicly available. All parameters involved in LBFGS were set to a reasonable value, in the same range from which they were chosen for several Python libraries that implement this technique.

3.2. Sensitivity Analysis

The general aim of this study consists in pinpointing the structural features and hyper-parameters that prove to play an important role in the approximation of the exact solution for a given PDE. For each combination of trial settings, we prompted the training of three networks with the exact same characteristics, initializing the values of their connections in a random fashion (employing the *Glorot Uniform Iniziatialization* method in [33]). In order to compare the performance of all models, we make use of a discrete approximation for the relative error measured in the L^2 norm, expressed as:

$$\|u\|_2 = \sqrt{\int_\Omega u^2(\boldsymbol{x})\ \boldsymbol{dx}} \approx \sqrt{\sum_{\boldsymbol{x}\in\mathcal{P}} u^2(\boldsymbol{x})},$$

where $\mathcal{P}$ represents a dense set of points in Ω. Guided by the implicit aleatory nature of all our experiments, we group and visualize the obtained results in two types of tables, one for each of the following criteria: *average* and *best* error. Our general conclusions mainly derive from the latter, considered to be the most representative.

3.2.1 Single-Scale

In the *single-scale* sensitivity analysis we aim at studying the behavior of differently-sized architectures on the simple one-dimensional Poisson equation with homogeneous boundary conditions, varying the forcing term in order to evaluate their performance with respect to an increasingly oscillating sinusoidal solution.

$$\begin{cases} -u'' = f & x \in (-1,1) \\ u = 0 & x \in \{-1,1\} \end{cases} \tag{1}$$

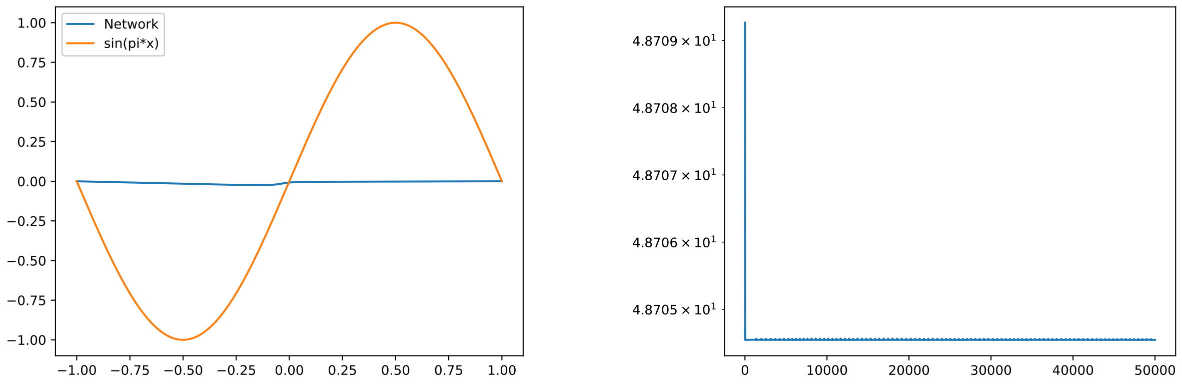
In this framework, a total of four frequencies have been considered. The extensive list of essential parameters involved in this study is composed by:

- Amount of uniformly distributed internal points over the computational domain $[-1,1]$: $\{80, 320, 1280\}$;
- Employed neurons per layer: $\{25, 50, 100\}$;
- Number of hidden layers: $\{1, 2\}$.

All models were trained with a total of 50000 ADAM iterations followed by another 50000 maximum number of LBFGS steps. The activation functions used throughout this analysis are the hyperbolic tangent and the well-known ReLU (Rectified Linear Unit), which have been defined and represented in Figure 2. As we previously anticipated, for every different combination of learning settings we have constructed three independent models, each prompted with a different random initialization. In all the analyses carried out in this work, with the terms *average error* and *best error* we respectively intend the arithmetical mean and the minimum value computed among the relative L^2 errors gathered for the triplets of networks trained with the same architectural features. The four sinusoidal solutions (frequencies) under study are listed here:

- $\sin(\pi x)$ (*low* frequency).
- $\sin(5\pi x)$ (*medium* frequency).
- $\sin(10\pi x)$ (*medium-high* frequency).
- $\sin(15\pi x)$ (*high* frequency).

The proper forcing term has been provided for all the trained Neural Networks for each of the presented targets. Our first important observation concerns the extremely poor performance shown by all networks trained with the ReLU activation function. A representative instance of this trend can be seen in Figure 7, where it is clear that the model immediately encountered the zero-solution local minimum of the cost functional. Surprisingly, despite the very smooth and simple sinusoidal target on which it was trained, the network was not able to escape this local pit. As a consequence of this repeated behavior, all these models present a relative error that is very close to the unity. In light of this first experimental result, a possible explanation for such failure must be related to a very tough landscape of the loss function, meaning that the latter evidently presents numerous local minima of this kind, at least when ReLU is employed as the activation function for the underlying PINN. This early conclusion will be actually validated *a posteriori* in the *generic* sensitivity analysis contained in a test of sub-subsection 3.2.3, where we will make use of several activation functions with different characteristics over a simple differential problem. Instead, analyzing the results coming from the networks trained with the hyperbolic tangent activation function, we face a completely different scenario. Indeed, for all the frequency solutions that we aimed to reproduce with the current test, we have been able to find at least one model that successfully emulates the phenomenon of our interest. These best candidates, in fact, present a relative L^2 error whose magnitude is well below 1%. All the tables contained in this work collect the relative L^2 errors computed for differently-structured networks, presenting their trends in relation to the number of hidden layers employed (indicated by HL) and the amount of neurons per layer with which they are embedded (NPL for short).



(a) Target solution and model approximation.

(b) Loss function evolution.

Figure 7: On the left, we visualize the poor approximation concerning the first among the networks trained with 1 hidden layer, 100 neurons and 320 residuals over the low frequency solution. On the right, we report the related cost functional evolution in semi-logarithmic scale. As we can infer from these two plots, the model got immediately stuck in a local minimum of the loss function.

NPL/HL	1	2
25	1.367280e-06	4.894299e-07
50	4.230057e-07	4.617714e-07
100	1.296052e-06	3.860986e-07

(a) Average error, 80 residuals.

NPL/HL	1	2
25	1.086028e-06	5.298863e-07
50	1.109894e-06	3.808153e-07
100	8.985316e-07	3.660251e-07

(b) Average error, 320 residuals.

NPL/HL	1	2
25	1.195013e-06	2.804097e-07
50	6.181360e-07	3.298432e-07
100	9.093512e-07	2.653261e-07

(c) Average error, 1280 residuals.

Table 1: Tables for the models trained to approximate the low frequency solution.

As we can see from Table 1, all PINNs endowed with the hyperbolic tangent are extremely reliable in the approximation of the low frequency solution, with an average error hovering around $1e-6$ and $1e-7$. Moreover, such a measure of success is also independent from both the amount of residual points employed during the learning phase and the structural hyper-parameters of the models. The latter clearly consist in the number of hidden layers employed and the amount of neurons per layer with which these are embedded.

NPL/HL	1	2
25	0.000050	1.481489
50	0.000004	0.000222
100	0.000015	0.000079

(a) Average error, 80 residuals.

NPL/HL	1	2
25	0.000133	0.000296
50	0.000013	0.000110
100	0.000010	0.000031

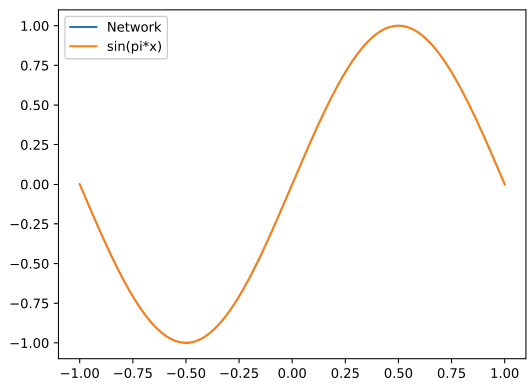
(b) Average error, 320 residuals.

NPL/HL	1	2
25	0.000074	0.000164
50	0.000023	0.000033
100	0.000029	0.000023

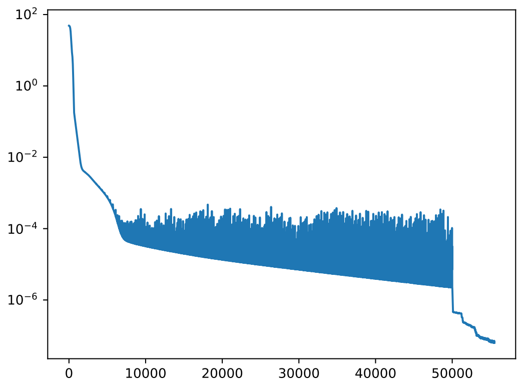
(c) Average error, 1280 residuals.

Table 2: Tables for the models trained to approximate the medium frequency solution.

Analogously to the first tranche of results, in Table 2 we can appreciate a range of excellent outcomes for the approximation of the medium frequency solution. Also here, in fact, we notice that the performances are more or less constant to a very high degree of accuracy, which remains unvaried with respect to the total number of residuals (or internal training points) and the size of the networks. We remark that, however good these approximations may be, there has been a consistent increase (of about one order of magnitude) in the average error with respect to the previous frequency target. Finally, it is clearly noticeable that one of the models trained with 2 hidden layers, 25 neurons per layer and 80 residual points got stuck in a local minimum associated to a quite large value of the loss function, compromising the average error result for the networks with this structure. The other two instances, however, show a relative error which is aligned to the performance of the other models. Figures 8 and 9 show the representation of the best networks for the two described settings alongside the plots concerning the evolution of the cost functional during the learning phase of the relative models, where we see the appearance of an oscillating phenomenon.

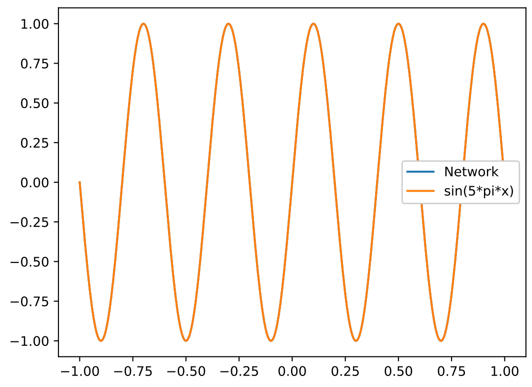


(a) Target solution and model approximation.

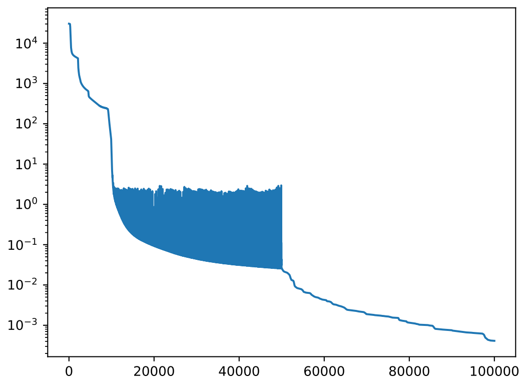


(b) Loss function evolution.

Figure 8: Successful approximation of the low frequency solution by the first model trained with 1 hidden layer, 100 neurons and 80 residuals. On the left, we can fully appreciate the perfectly overlapping plots of the network and the exact solution, while on the right we see the whole loss function evolution in semi-logarithmic scale. After about 10000 iterations, the first-order ADAM method starts oscillating.



(a) Target solution and model approximation.



(b) Loss function evolution.

Figure 9: Successful approximation of the medium frequency solution by the first model trained with 2 hidden layers, 25 neurons per layer and 80 residuals. On the left, we can fully appreciate the perfectly overlapping plots of the network and the exact solution, while on the right we see the entire loss function evolution in semi-logarithmic scale. After about 10000 iterations, the first-order ADAM method starts oscillating also in this case. The subsequent intervention of LBFGS dissipates this behavior.

The mentioned behavior is believed to manifest itself when the ADAM algorithm approaches, in the generally unknown landscape of the loss function, a region in which it is not able to understand the correct descent direction. The explanation of this phenomenon might reside in the fact that the ADAM optimizer only exploits the components of the gradient of the cost functional, with no additional information on its local curvature. Unsurprisingly, in both cases, we see a rapid decrease in the loss magnitude as soon as the LBFGS method comes into play. This technique, in fact, computes and subsequently exploits an approximate estimation of the local curvature of the cost function in order to select the most appropriate descent direction.
Let us now continue our analysis with the results concerning the approximation of the medium-high frequency solution, shown in Table 3. Contrarily to what we did in the previous cases, in this table we represent the best error trend of the models, instead of the average error performance. For the first time, we can make a clear distinction between the networks trained with a single hidden layer and the ones embedded with two of them. Independently from the number of neurons per layer and the amount of residual points employed during the learning phase, the models constructed with a single hidden layer exhibit a gain in terms of the relative L^2

error of 1.6% at worst. On the other hand, among the networks trained with two hidden layers we find just a few instances that are actually able to provide an error below an acceptable threshold. Regarding the attempts made with a single layer we can say that, keeping all the remaining hyper-parameters fixed, the best error trend tends to increase with respect to the number of exploited residual points and consistently decreases with respect to the width of the hidden layers.

NPL/HL	1	2
25	0.006956	2.176026
50	0.000391	9.132307
100	0.000318	6.072858

(a) Best error, 80 residuals.

NPL/HL	1	2
25	0.004699	15.495498
50	0.001069	15.223295
100	0.000312	0.000543

(b) Best error, 320 residuals.

NPL/HL	1	2
25	0.016992	15.584377
50	0.001717	15.975008
100	0.000404	0.029286

(c) Best error, 1280 residuals.

Table 3: Tables for the models trained to approximate the medium-high frequency solution.

NPL/HL	1	2
25	18.931327	3.275145
50	0.030746	8.548564
100	0.003015	13.817795

(a) Best error, 80 residuals.

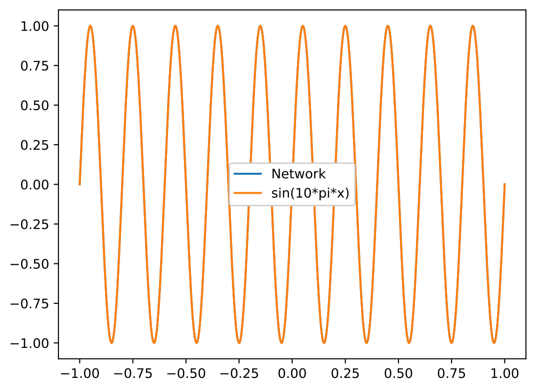
NPL/HL	1	2
25	20.300668	19.433272
50	0.081179	10.113400
100	0.043341	23.339901

(b) Best error, 320 residuals.

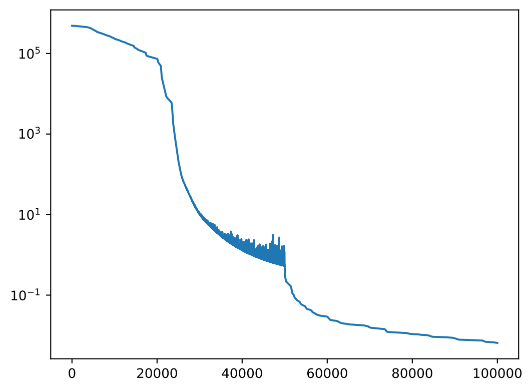
NPL/HL	1	2
25	18.397548	18.787365
50	0.097042	22.772743
100	0.016735	19.927546

(c) Best error, 1280 residuals.

Table 4: Tables for the models trained to approximate the high frequency solution.



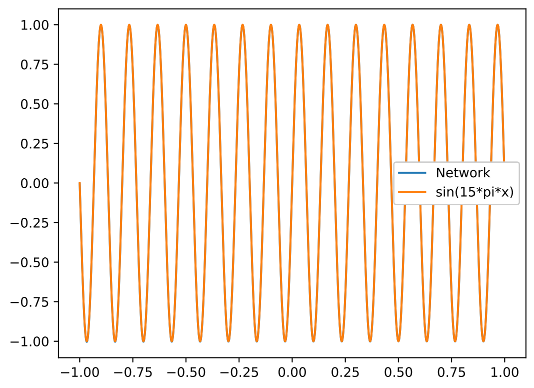
(a) Target solution and model approximation.



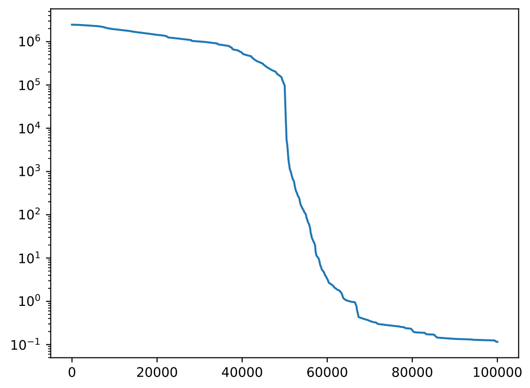
(b) Loss function evolution.

Figure 10: Successful approximation of the medium-high frequency solution by the first model trained with 1 hidden layer, 100 neurons and 80 residuals. On the left, we can fully appreciate the perfectly overlapping plots of the network and the exact solution, performing 10 complete oscillations in the computational domain. On the right, we see the loss function evolution in semi-logarithmic scale.

Concerning the results relative to the approximation of the solution with the highest frequency we clearly notice that, in order to obtain a reliable representation of the target, there is a minimum requirement on the number of neurons per layer that should be used. Indeed, all networks trained with less than 50 neurons per layer are not able to properly fit the solution. For the first time in this analysis, even the best instances of the models with a single layer composed by 25 neurons fail their task. All networks with two hidden layers exhibit a consistent error, while the architectures with one hidden layer endowed with 50 or 100 neurons are able to provide at least one successful instance with a relative L^2 error below 10%, independently from the number of residuals employed. Among the well behaving models we appreciate an increase in performance with a larger number of neurons, while it is not completely clear whether the same trend is followed with respect to the number of internal training points. The network that best represents the solution was instructed with 80 residual points, 1 hidden layer and 100 neurons per layer. In Figure 11 we visualize its plot and the relative loss evolution.

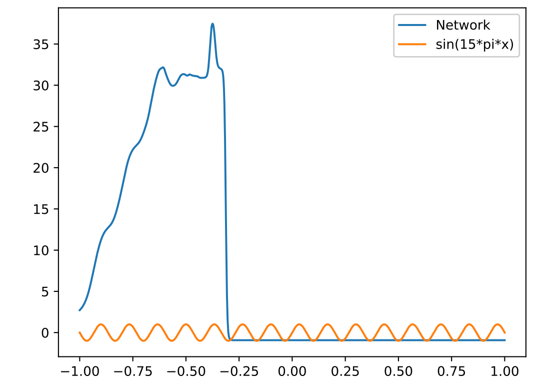


(a) Target solution and model approximation.

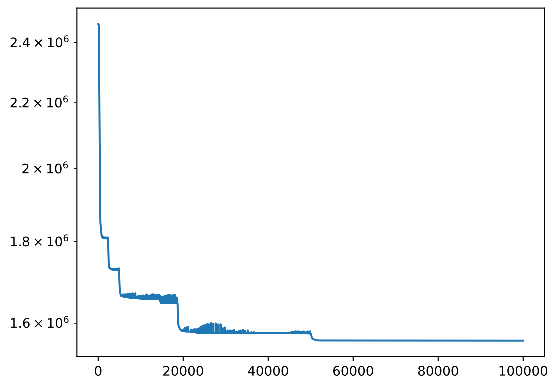


(b) Loss function evolution.

Figure 11: Best approximation of the high frequency solution relative to the first model trained with 1 hidden layer, 100 neurons and 80 residuals. On the left, we can fully appreciate the perfectly overlapping plots of the network and the exact solution, performing 15 complete oscillations in the computational domain. On the right, we see the loss function evolution in semi-logarithmic scale.



(a) Target solution and model approximation.



(b) Loss function evolution.

Figure 12: Poor approximation of the high frequency solution relative to the third model trained with 2 hidden layers, 100 neurons per layer and 80 residuals. On the left, we represent the network and the target that should be simulated. On the right, we see the cost evolution in semi-logarithmic scale.

For completeness, in Figure 12 we also show the bad performance of a model that was not able to extract the expected features from the target function. The overall accuracy of such network is representative of all the 2-layered architectures, which are never able to provide a reliable approximation of the solution. From their loss function evolution, which present a characteristic series of plateaux, we can easily infer their cost functional landscape to be extremely complicated to be interpreted by the optimization procedure.
In light of all results presented in this analysis, we can conclude that all the architectural features (number of hidden layers, amount of neurons per layer and cardinality of the training set) that were initially supposed to play a potentially significant role in the approximation capabilities of the underlying PINNs actually seem to confirm the validity of these expectations. Given the particular settings of this experiment, we generally appreciate the best results coming from the networks with a single hidden layer endowed with 100 neurons. Moreover, we have seen how it is not true that more training points necessarily lead to better approximations, at least under the environmental conditions exploited in our experiments. It should be needless to say that different instances of ground settings (that, among the others, involve the exploited target solutions and the number of learning iterations performed during the simulations) may well have led to other outcomes with, consequently, different final considerations.

3.2.2 Multi-Scale

Pursuing the path drawn by our sensitivity analysis for single-frequency targets, we move forward with two *multi-scale* tests, extending our study to the category of solution targets involving multiple frequencies in their analytical expression. Our objective still concerns the evaluation of the influence attached to each architectural parameter on the overall performance shown by the models. Such variables consist in the number of uniformly distributed residual points, the total number of hidden layers and the amount of neurons per layer employed.

Test 1

In this first multi-scale test we reuse the machinery introduced for the single-scale sensitivity study. In particular, we exploit the linearity of the one-dimensional Poisson equation with homogeneous boundary conditions (1) to impose a multi-frequency function as the solution of this differential problem. Such target is nothing else than the weighted sum (with decreasing coefficients) of the four frequency solutions previously encountered. All models were trained with a total of 25000 ADAM iterations followed by another 75000 maximum number of LBFGS steps. The remaining settings, concerning the list of ranges for the architectural options and the activation functions embedded in the networks, are exactly identical to the ones used in the previous analysis. In Table 5, analogously to what we observed in the single-scale sensitivity analysis, it seems very likely that all models trained with the ReLU activation function encountered a bad local minimum of the loss during their learning phase. This conjecture is actually confirmed by the plots shown in Figure 13, where it is evident that the illustrated network immediately got stuck in the zero-solution local minimum of the cost functional.

NPL/HL	1	2
25	0.996798	0.999333
50	0.998223	1.000019
100	1.000138	0.999838

(a) Best error, 80 residuals.

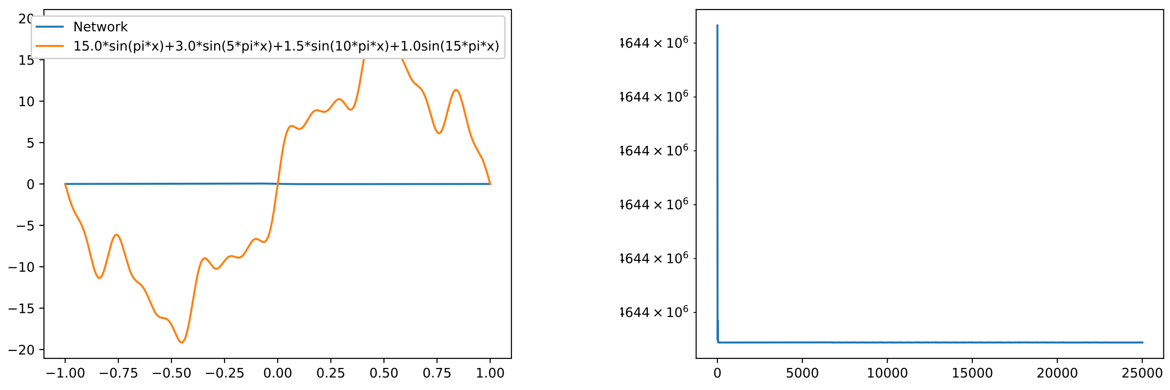
NPL/HL	1	2
25	0.997638	1.000008
50	0.996829	0.999735
100	0.998572	0.999960

(b) Best error, 320 residuals.

NPL/HL	1	2
25	0.998380	0.999768
50	0.997587	0.999458
100	1.000764	0.999885

(c) Best error, 1280 residuals.

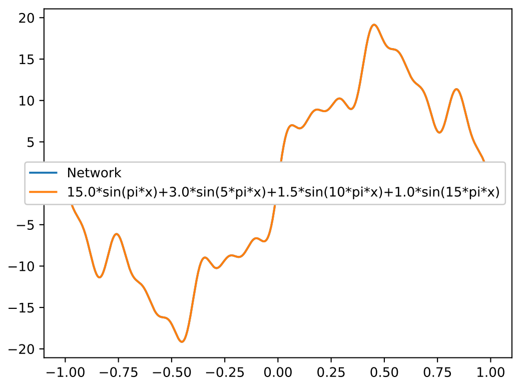
Table 5: Tables for the ReLU models trained to approximate the multi-scale solution.



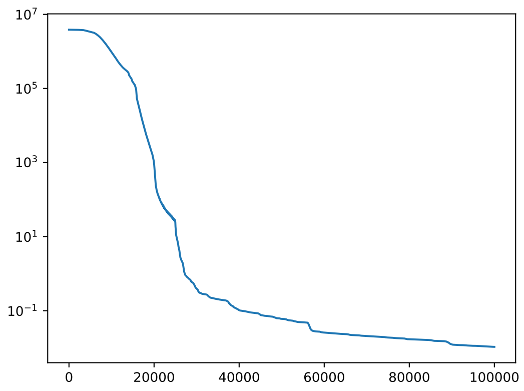
(a) Target solution and model approximation.
(b) Loss function evolution.

Figure 13: Poor approximation of the multi-frequency solution relative to the first model trained with 1 hidden layers, 100 neurons and 80 residuals with the ReLU activation function. On the left, we represent the network and the target that should be simulated. On the right, we see the cost evolution in semi-logarithmic scale, basically constant throughout the entire range of optimization iterations.

In Figure 14 we visualize the functional representation and the loss function evolution of the network whose approximation is the most reliable among all models trained for this test. Table 6, on the other hand, reports the performance resume of all networks embedded with the hyperbolic tangent activation function.



(a) Target solution and model approximation.



(b) Loss function evolution.

Figure 14: Best approximation of the multi-frequency solution by the first model trained with 1 hidden layer, 100 neurons and 320 residuals. On the left, we can fully appreciate the perfectly overlapping plots of the network and the exact solution, while on the right we see the entire loss function evolution in semi-logarithmic scale, passing from nearly $1e7$ to a final value in between $1e-1$ and $1e-2$.

NPL/HL	1	2
25	4.108624	1.218205
50	0.004938	1.802231
100	0.000058	1.402370

(a) Best error, 80 residuals.

NPL/HL	1	2
25	3.132702	1.586473
50	0.006641	0.728400
100	0.000020	0.460028

(b) Best error, 320 residuals.

NPL/HL	1	2
25	2.169297	1.817634
50	0.001935	1.935542
100	0.000036	2.466783

(c) Best error, 1280 residuals.

Table 6: Tables for the Tanh models trained to approximate the multi-scale solution.

In light of the results revealed by the single-scale sensitivity analysis concerning the high frequency solution, it is not surprising to witness a lack of accuracy for all the models embedded with two hidden layers. Recall, in fact, that such frequency belongs to the spectrum of the analyzed multi-scale solution. For the same reason, as expected, the networks with a single layer and 25 neurons are not sufficiently large to grasp all the features of such target. Regarding the other architectures we always appreciate, among the usual three attempts performed, at least one successful instance which well approximates the solution to this problem. We particularly observe an evident improvement in performance for a higher number of neurons (nearly two orders of magnitude better), while we ascertain a basically constant (or at least non-decreasing) trend when we simply vary the number of uniformly distributed residual points that are employed during the optimization procedure.

Test 2

Inspired by [65], we try to approximate the multi-scale solution $\sin(2\pi x) + 0.1\sin(50\pi x)$ for the associated one-dimensional Poisson equation with homogeneous boundary conditions (1). Let us now provide the ranges of variation for the usual architectural parameters for this test case.

- Amount of uniformly distributed internal points over the computational domain $[-1, 1]$: $\{160, 320, 480\}$;
- Employed neurons per layer: $\{100, 200, 400\}$;
- Number of hidden layers: $\{1, 2\}$.

All models were trained with a total of 25000 ADAM iterations followed by another 175000 maximum number of LBFGS steps. Discouraged by the results shown by the models trained with ReLU in the previous analysis, we decided to select only the hyperbolic tangent as the activation function for this experimental trial. The overall performances of the relative architectures have been entirely gathered in Table 7.
As we can see, in terms of pure error there is only one network that has been able to reach a satisfactory level of accuracy with the given settings. Such model is characterized by 2 hidden layers, 400 neurons per layer and 320 residuals. For the first time in this work, we have launched a test where not even a single architecture embedded with a unique hidden layer has been able to accurately reproduce the expected outcome. Nevertheless, all models endowed with two hidden layers were approximately ten times more (computationally)

expensive than the single-layered networks. For a fair comparison in terms of computational cost, we performed three additional simulations with only one hidden layer and all the other architectural parameters fixed to the values that were exploited for the unique successful (double-layered) model of this test (400 neurons per layer and 320 uniformly distributed residual points) and ten times more learning iterations.

NPL/HL	1	2
100	1.981729	3.013872
200	2.970425	12.243158
400	0.599130	4.283990

(a) Best error, 160 residuals.

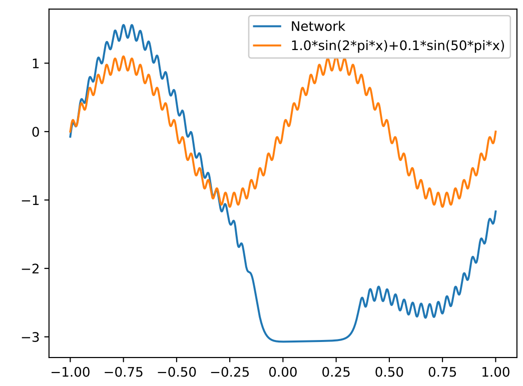
NPL/HL	1	2
100	1.765399	3.701198
200	1.614595	2.508974
400	2.929941	0.019762

(b) Best error, 320 residuals.

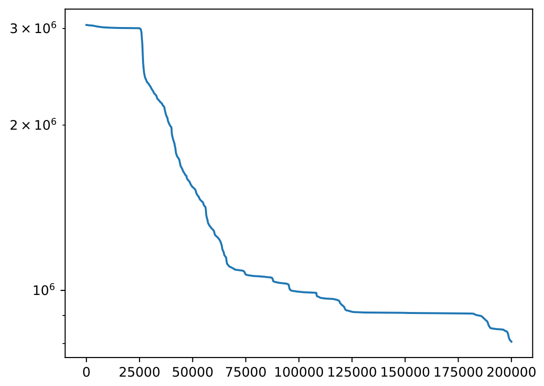
NPL/HL	1	2
100	1.271179	3.588748
200	1.780399	5.676355
400	1.250672	0.268985

(c) Best error, 480 residuals.

Table 7: Tables for the models trained to approximate the multi-scale solution.

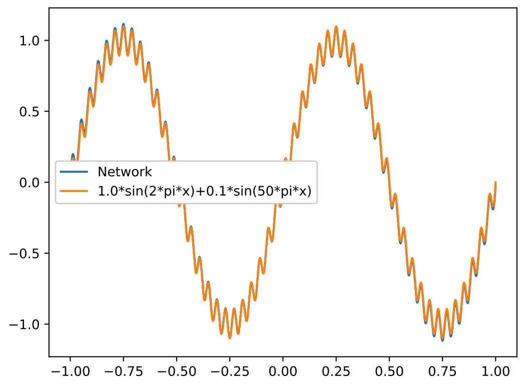


(a) Target solution and model approximation.

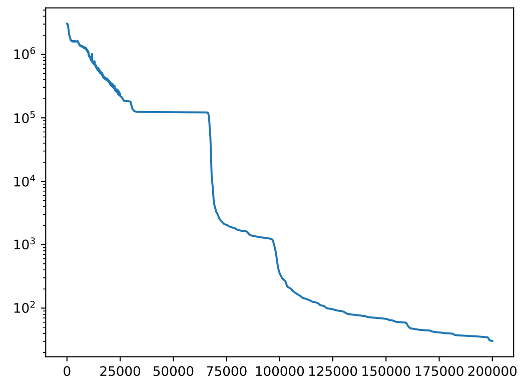


(b) Loss function evolution.

Figure 15: Unsuccessful approximation of the multi-frequency solution by the first model trained with 1 hidden layer, 400 neurons and 320 residuals. On the left, we can sharply distinguish the plots relative to the network and the exact solution, while on the right we see the entire loss function evolution in semi-logarithmic scale. This network does not belong to the additional tranche of models.



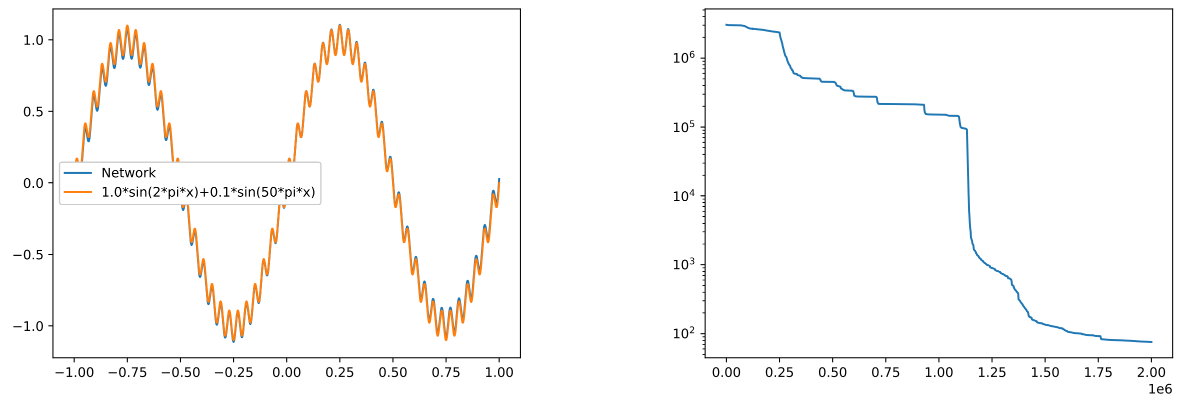
(a) Target solution and model approximation.



(b) Loss function evolution.

Figure 16: Best approximation of the multi-frequency solution by the third model trained with 2 hidden layers, 400 neurons per layer and 320 residuals. On the left, we can fully appreciate the almost perfectly overlapping plots of the network and the exact solution, while on the right we see the entire loss function evolution in semi-logarithmic scale, passing from nearly $1e6$ to a final value close to $1e1$.

In Figure 15, we represent a single-layered network whose learning phase was seemingly going in a promising direction. In the last few thousands iterations, in fact, the loss magnitude started a sudden decrease that might have led to a satisfactory instance of such architecture. Furthermore, except for the central band of the domain, the PINN has been able to understand the low and especially the high frequency spectrum of the solution. These observations further justify our efforts put in the additional attempts that we performed, in which we enforced a much longer optimization procedure in order to observe whether also the shallow networks are potentially able to grasp this stiff solution, if provided with a sufficient amount of learning time. After all these necessary premises we are finally ready to analyze the aforementioned extra tranche of single-layered PINNs, which have been endowed with a boosted number of learning steps. Figure 17 shows the plots related to the best network among these three extra models, all trained with ten times more learning iterations than the others. Its relative L^2 error, of about 2.4%, is surprisingly close to the one associated with the best among the models of the first tranche, depicted in Figure 16. Also the other two PINNs belonging to the second set of models are able to closely fit the solution, even though they present a slightly higher value for their relative error.



(a) Target solution and model approximation.

(b) Loss function evolution.

Figure 17: Best approximation of the multi-frequency solution by the third model trained with 1 hidden layer, 400 neurons, 320 residuals and an increased number of learning iterations. On the left, we can fully appreciate the almost perfectly overlapping plots of the network and the exact solution, while on the right we see the entire loss function evolution in semi-logarithmic scale.

In line with what we previously said at the end of the single-scale analysis, we can naturally conclude that the architectural settings (consisting in the number of hidden layers, amount of neurons per layer and number of uniformly distributed training points) represent the main keys that determine the approximation quality of the resulting networks, as expected. Thanks to our last case of study, however, we also understood that the number of learning iterations plays a fundamental role as well. Let us now readily move to the central part of our work.

3.2.3 Generic

We shall now dig into the details of the sensitivity *generic* tests for our basic PINN, exploring its behavior over a set of classical PDEs presenting a variety of solutions. Such analysis, that represents the heart of our work, is based on the evaluation of the relative L^2 error (Rel_Err). As an equivalent measure for the reliability of our networks we can also consider the accuracy (Acc), defined as: $Acc = 1 - Rel\_Err$.

Saw-Teeth Tests

Test 1, 2 and 3 of this section share the same underlying solutions, the so-called saw-teeth waves (later represented in Figures 18, 20 and 21), to the respective governing PDE: it is for this reason that the former are jointly introduced here. Each of them is further subdivided into three parts, where a different target profile is indirectly imposed through a distinct source term coupled with the same equation and boundary conditions. Test 1 consists in evaluating the performance of several architectures for the problem of pure approximation, Test 2 implicitly passes the same solutions through an ODE involving their almost everywhere well-defined first order derivative, while Test 3 resolves a Poisson problem whose right hand side is a regularization of the *Dirac delta*, the *distributional* second order derivative associated to these targets. It is crucial to remember that any saw-teeth wave is obtainable as the DNN realization corresponding to the composition of an elementary ReLU network with itself (for a specific number of times) (see [49]). Such profile presents 2^{N+1} linear pieces, where N stands for the number of performed compositions. As we can see, the total amount of the former increases exponentially with respect to the latter, accordingly to the number of teeth (equal to 2^N). It is in this specific framework that Theorem 2.2 comes into play: due to its statement, we immediately understand that shallow networks must be necessarily endowed with a large number (exponential with respect to N) of neurons in order to be potentially able to approximate these solutions. We therefore expect to appreciate much more reliable results from models embedded with at least two hidden layers, at least in this context. With the aim to explore and validate the outcome of these tests in light of the cited theorem, we selected a proper range for the number of neurons per layer employed in our architectures, in which the lowest values are not sufficient to allow the shallow networks to reliably approximate our targets. The highest values in these lists, instead, permit an accurate result even for these slim architectures, at least *in principle*. These three tests, originally designed for the ReLU networks, have been extended also to identical architectures that utilize the hyperbolic tangent activation function. Before proceeding with a complete description of all these cases, we conclude this overview by summing up the fundamental aspects that we want to explore through these correlated studies:

- Understand the role of depth in the approximation for a specific class of functions.
- Observe the behavior of indefinitely regular Tanh models while they learn piece-wise linear targets.
- Investigate the ability of PINNs in grasping a fixed solution by interpreting it under three different perspectives: pure approximation, enforcing its first order derivative through an ODE and imposing a regularization of its second order derivative by means of the Poisson equation.

Test 1

Let us now consider the pure approximation problem for the mentioned class of saw-teeth waves.

$$u = f \qquad x \in [0, 1] \tag{2}$$

For these experiments we chose to vary the architectural parameters in the following ranges:

- For the 4-saw-teeth solution, corresponding to $N = 2$ (see Figure 18):
  - Amount of uniformly distributed internal points over the computational domain $[0, 1]$: $\{20, 40, 80\}$;
  - Employed neurons per layer: $\{2, 4, 8, 16, 32, 64, 128\}$;
  - Number of hidden layers: $\{1, 2, 3\}$.
- For the 8-saw-teeth solution, corresponding to $N = 3$ (see Figure 20):
  - Amount of uniformly distributed internal points over the computational domain $[0, 1]$: $\{50, 100, 200\}$;
  - Employed neurons per layer: $\{2, 4, 8, 16, 32, 64, 128, 256\}$;
  - Number of hidden layers: $\{1, 2, 3\}$.
- For the 16-saw-teeth solution, corresponding to $N = 4$ (see Figure 21):
  - Amount of uniformly distributed internal points over the computational domain $[0, 1]$: $\{250, 500, 1000\}$;
  - Employed neurons per layer: $\{5, 10, 20, 40, 80, 160, 320\}$;
  - Number of hidden layers: $\{1, 2, 3\}$.

During the learning phase of all models we performed 25000 ADAM iterations followed by 75000, 125000 and 175000 maximum LBFGS steps for these three parts, respectively. The reason for which we boosted the training iterations for the solution targets with a higher number of linear pieces obviously resides in the increasing stiffness of the respective functional profiles, for which we expect our models to need a longer learning procedure. As anticipated, we employed both ReLU and the hyperbolic tangent (Tanh) activation functions.

NPL/HL	1	2	3
2	0.473970	0.303289	0.330498
4	0.335536	0.101400	0.083329
8	0.081556	0.166978	0.207404
16	0.105302	0.118761	0.087268
32	0.100130	0.104283	0.106371
64	0.090872	0.094726	0.086730
128	0.093540	0.123094	0.125643

(a) Best error, 20 residuals.

NPL/HL	1	2	3
2	0.464863	0.301092	0.357771
4	0.332342	0.047544	0.028198
8	0.057466	0.024726	0.024074
16	0.043763	0.025314	0.023858
32	0.043299	0.025926	0.022286
64	0.054347	0.025638	0.023264
128	0.039938	0.025363	0.023296

(b) Best error, 40 residuals.

NPL/HL	1	2	3
2	0.464857	0.464277	0.297231
4	0.332301	0.053901	0.033823
8	0.058822	0.030465	0.011049
16	0.051910	0.017289	0.008405
32	0.049633	0.015921	0.008107
64	0.047400	0.010865	0.007474
128	0.049400	0.011354	0.007905

(c) Best error, 80 residuals.

Table 8: Tables for the Tanh models trained to approximate the 4-saw-teeth solution ($N = 2$).

NPL/HL	1	2	3
2	0.591705	0.527785	0.591710
4	0.527785	0.465181	0.466103
8	0.465180	0.415417	0.293268
16	0.465199	0.416296	0.293730
32	0.591730	0.327118	0.060990
64	0.439188	0.043996	0.058448
128	0.326080	0.029619	0.056192

(a) Best error, 20 residuals.

NPL/HL	1	2	3
2	0.592457	0.569203	0.661452
4	0.592457	0.569203	0.592457
8	0.465466	0.324929	0.389396
16	0.414214	0.007199	0.293054
32	0.418693	0.293176	0.018639
64	0.292910	0.022210	0.025200
128	0.009365	0.015527	0.026813

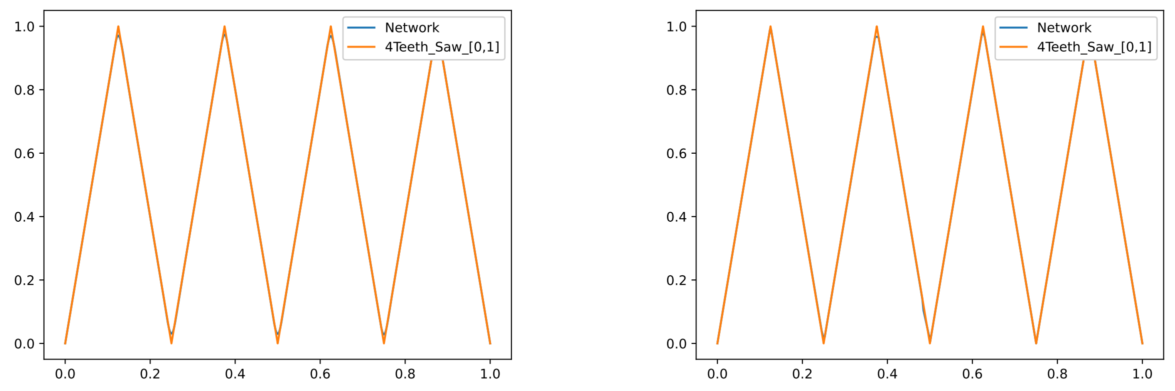
(b) Best error, 40 residuals.

NPL/HL	1	2	3
2	0.592484	0.569111	0.661452
4	0.569111	0.592484	0.437429
8	0.437422	0.464102	0.389281
16	0.389270	0.324899	0.005755
32	0.325007	0.292926	0.005905
64	0.292894	0.008450	0.007756
128	0.004502	0.004558	0.009267

(c) Best error, 80 residuals.

Table 9: Tables for the ReLU models trained to approximate the 4-saw-teeth solution ($N = 2$).

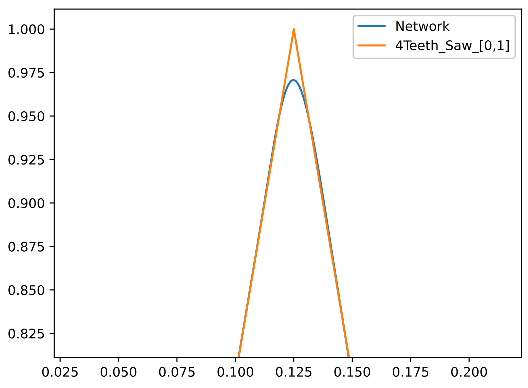
Tables 8 and 9 show the pattern followed by the best relative L^2 error related to the networks trained to approximate the 4-saw-teeth solution. In both we can evidently appreciate the importance connected to the number of uniformly distributed residual points. Increasing their density inside the domain, in fact, generally leads to better performances in basically all situations. Even more importantly, we clearly see what we expected to observe in relation to the role played by the number of hidden layers employed. Indeed, enlarging the architecture by adding new layers not only improves the general performance of the model, but also makes the approximation easier to realize even with fewer neurons per layer, in agreement with Theorem 2.2. Deep networks present, indeed, a clear edge with respect to their shallow counterparts, which typically fail (apart from sporadic cases) in finding a reliable approximation until they possess a considerably high number of neurons.
In Figure 18 we appreciate the graphical representation of the best models obtained in the first part of this test, basically overlapping with the exact solution. Taking into account the statement of Theorem 2.2, it is not surprising to ascertain that the networks endowed with multiple hidden layers have proven a much higher stability, in terms of performance, with respect to shallow models. Despite the validity of our previous consideration, we see that the most reliable among the ReLU architectures is characterized by a single hidden layer, even though the other two attempts made with the same specifications actually failed to reach an acceptable degree of precision, confirming the previous observation concerning the instability of the shallow networks. Due to the nature and regularity of the solution, it has been easily verified that ReLU outplays Tanh here. In order to fully understand the main effects, advantages and drawbacks depending on the activation functions employed for this specific test case we can inspect Figure 19, where we appreciate how their characteristic nature affects the approximation of the 4-saw-teeth solution target in the delicate regions of the domain where the solution suddenly inverts the sign of its derivative. We can clearly observe that, in correspondence of the mentioned sharp peaks and pits, the models embedded with the hyperbolic tangent activation function are not able to grasp the correct profile while ReLU networks prove to be much more reliable in doing so (as we should expect).



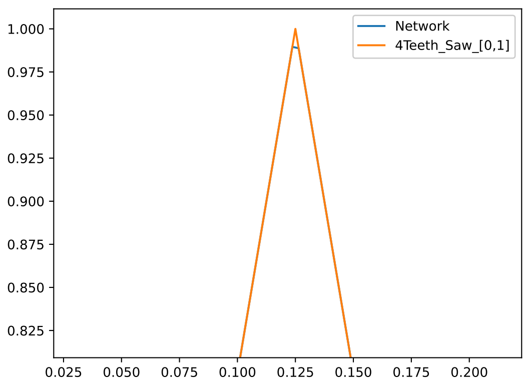
(a) Target solution and Tanh model approximation.

(b) Target solution and ReLU model approximation.

Figure 18: Best approximations of the 4-saw-teeth solution by the third Tanh network trained with 3 hidden layers, 64 neurons per layer and 80 residuals (left) and by the second ReLU network trained with 1 hidden layer, 128 neurons and 80 residuals (right). Even though they both match the expected behavior with a satisfying level of precision, the smooth Tanh model presents evident lacks of accuracy near the sharp peaks and troughs of the solution, as expected (see Figure 19).



(a) Tanh model.



(b) ReLU model.

Figure 19: Zoom of the first peaks related to the best networks shown in Figure 18. The indefinitely regular Tanh model struggles visibly more than the suited ReLU architecture in this region.

Tables 10 and 11 present the best relative L^2 errors related to the models trained with the hyperbolic tangent and ReLU activation functions for the second part of this test, respectively. Looking at the former, we immediately notice that these networks work acceptably well even with only 50 residuals, although it must be noticed that they are never able to stay below the error threshold of 5%. Another interesting consideration concerns the fact that, for the mentioned architectures, there are not clear clues about the role played by the number of hidden layers or the amount of neurons per layer. It is by using 100 and 200 internal points that the expected hierarchy comes back to the center of our stage: with these configurations the single-layered models are still not able to overcome the 5% relative error, while the other architectures reach a best accuracy of about 96%-98% when they exploit 100 residual points and more than 99% using 200 of them.

NPL/HL	1	2	3
2	0.483842	0.454231	0.416491
4	0.428456	0.319222	0.104757
8	0.426125	0.063296	0.083011
16	0.302581	0.063895	0.172420
32	0.062738	0.062754	0.145462
64	0.061551	0.059931	0.176483
128	0.062212	0.210803	0.112224
256	0.062369	0.140507	0.058574

(a) Best error, 50 residuals.

NPL/HL	1	2	3
2	0.483646	0.415747	0.450111
4	0.428397	0.376780	0.214147
8	0.313455	0.061924	0.045200
16	0.069404	0.048284	0.030372
32	0.058254	0.024484	0.024442
64	0.057371	0.032799	0.026957
128	0.058643	0.031344	0.027478
256	0.057134	0.029791	0.037336

(b) Best error, 100 residuals.

NPL/HL	1	2	3
2	0.483645	0.415832	0.414092
4	0.428387	0.349524	0.064410
8	0.340414	0.059984	0.041696
16	0.077873	0.050662	0.015590
32	0.058090	0.026023	0.008639
64	0.058764	0.031628	0.008304
128	0.058315	0.022669	0.007965
256	0.058076	0.016042	0.009275

(c) Best error, 200 residuals.

Table 10: Tables for the Tanh models trained to approximate the 8-saw-teeth solution ($N = 3$).

NPL/HL	1	2	3
2	0.573765	0.661906	0.573765
4	0.573765	0.483326	0.604777
8	0.478153	0.478874	0.480136
16	0.463899	0.418317	0.276717
32	0.464381	0.230769	0.209022
64	0.394254	0.275768	0.209084
128	0.378964	0.025940	0.025754
256	0.214168	0.025437	0.037347

(a) Best error, 50 residuals.

NPL/HL	1	2	3
2	0.573765	0.661906	0.487534
4	0.573765	0.483318	0.483318
8	0.483318	0.478771	0.428379
16	0.483318	0.460372	0.229815
32	0.444528	0.296122	0.275304
64	0.328192	0.021822	0.006964
128	0.309418	0.004320	0.008856
256	0.207454	0.008119	0.012176

(b) Best error, 100 residuals.

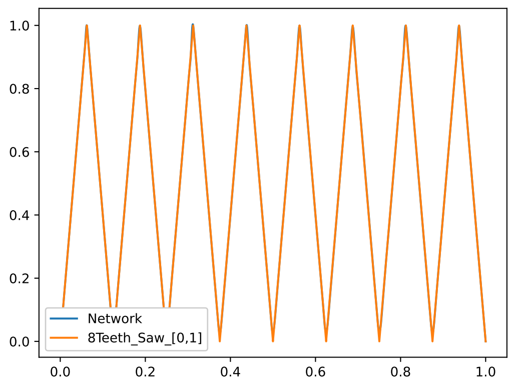
NPL/HL	1	2	3
2	0.572816	0.604619	0.572816
4	0.483317	0.600640	0.449694
8	0.483317	0.483317	0.390820
16	0.478765	0.456218	0.229735
32	0.456218	0.346420	0.296061
64	0.404229	0.207151	0.207144
128	0.229846	0.207144	0.003600
256	0.275854	0.003493	0.004448

(c) Best error, 200 residuals.

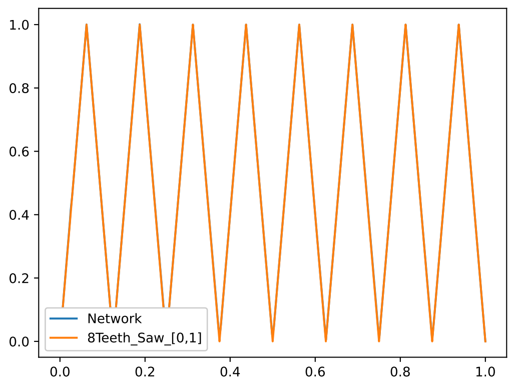
Table 11: Tables for the ReLU models trained to approximate the 8-saw-teeth solution ($N = 3$).

Regarding the ReLU networks whose learning procedure was performed with the lowest number of training points, we acknowledge success only for the multiple-layered architectures embedded either with 128 or 256 neurons per layer. Such trend is extended also for the models trained with 100 residual points, which exhibit

a similar pattern and generally improve their performance, reaching a satisfying level of accuracy even when they are endowed with a smaller number of neurons per layer. In absolute terms, the best architectures were obtained exploiting 200 internal spots. Nevertheless, some structural settings (e.g. the ones corresponding to the two-layered networks with 64 or 128 neurons each) are able to provide excellent levels of accuracy with a lower number of residuals but not with 200 of them, showing a perceivable lack of stability with respect to the mentioned parameter.
In Figure 20 we graphically show the plots linked to the best networks obtained for the second part of this test, trained with identical architectural settings. The ReLU representative presents a smaller relative error, as expected, even though the best model embedded with Tanh carries an accuracy of the same order of magnitude.



(a) Target solution and Tanh model approximation.



(b) Target solution and ReLU model approximation.

Figure 20: Best approximations of the 8-saw-teeth solution by the first Tanh network trained with 3 hidden layers, 128 neurons per layer and 200 residuals (left) and by the second ReLU network trained with the exact same architecture (right). Both show a very high level of accuracy.

Tables 12 and 13 report the best relative L^2 errors for the models trained to approximate the 16-saw-teeth solution target. In the former, where we represent the outcomes concerning the Tanh networks, we see that by keeping the amount of hidden layers fixed, the networks accuracy clearly increases with a growing number of neurons per layer until it reaches a more or less stable saturation level. Vice versa, fixing the latter and raising the former, we still see important improvements. In particular, wider architectures (embedded with more hidden layers) prove to reach a satisfactory performance needing fewer neurons per layer. Regarding the second table, where we refer to the ReLU architectures, we can provide similar general comments and observations. There are, nevertheless, a few distinctions that immediately emerge: on one hand ReLU models, with respect to Tanh networks, typically require a higher minimum number of neurons per layer to reach good levels of accuracy. On the other hand, however, they possess the best candidates among all. Relating to this fact, it is worth highlighting that the most reliable ReLU architectures beat, in terms of pure performance, the best Tanh networks by approximately one order of magnitude. Another important difference resides in the fact that no single-layered ReLU network has been able to get even close to the solution of this problem, showing a lack of reliability from these architectures even when they possess (in principle) the needed capacity to provide an accurate approximation. We finally notice that augmenting the cardinality of the training set typically leads to better performances, independently from the activation function in use.

NPL/HL	1	2	3
5	0.457641	0.392238	0.389087
10	0.371480	0.368605	0.288920
20	0.365798	0.059660	0.048496
40	0.332071	0.056279	0.018936
80	0.059669	0.047758	0.016864
160	0.060418	0.052315	0.017574
320	0.060062	0.041518	0.018140

(a) Best error, 250 residuals.

NPL/HL	1	2	3
5	0.450262	0.420352	0.396867
10	0.455074	0.367055	0.255565
20	0.400755	0.106033	0.053831
40	0.280051	0.058448	0.020463
80	0.060664	0.055957	0.013948
160	0.060294	0.053410	0.019115
320	0.060275	0.040950	0.006873

(b) Best error, 500 residuals.

NPL/HL	1	2	3
5	0.465914	0.429389	0.383387
10	0.441058	0.323248	0.279350
20	0.398551	0.062156	0.057328
40	0.235473	0.054416	0.034279
80	0.060394	0.056609	0.015498
160	0.060683	0.058572	0.009427
320	0.059855	0.044218	0.006693

(c) Best error, 1000 residuals.

Table 12: Tables for the Tanh models trained to approximate the 16-saw-teeth solution ($N = 4$).

NPL/HL	1	2	3
5	0.491935	0.490530	0.490767
10	0.491935	0.483532	0.445216
20	0.491294	0.491936	0.406077
40	0.475781	0.456147	0.287172
80	0.467045	0.359986	0.245594
160	0.458087	0.243935	0.005066
320	0.402185	0.011704	0.008276

(a) Best error, 250 residuals.

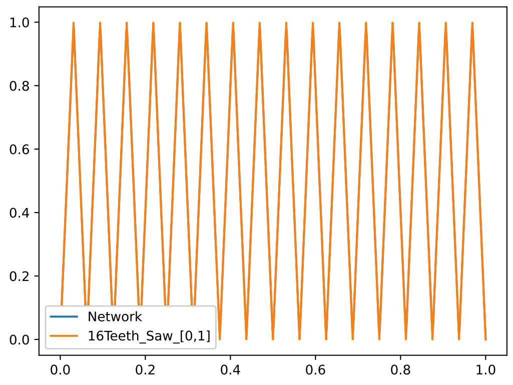
NPL/HL	1	2	3
5	0.575364	0.481559	0.490768
10	0.491936	0.491294	0.459530
20	0.482984	0.476364	0.401010
40	0.472609	0.437093	0.342825
80	0.464358	0.361989	0.223663
160	0.431833	0.162467	0.002723
320	0.421541	0.003341	0.002642

(b) Best error, 500 residuals.

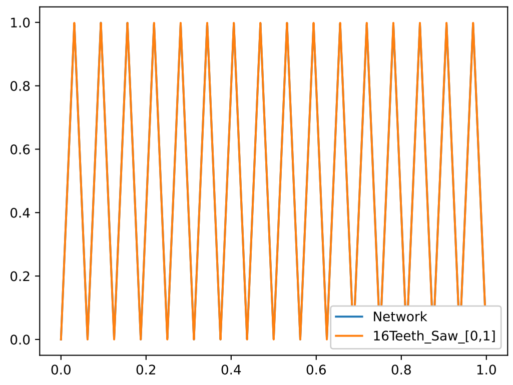
NPL/HL	1	2	3
5	0.491935	0.490054	0.465546
10	0.491181	0.486471	0.429122
20	0.481683	0.441678	0.389848
40	0.467036	0.365021	0.276299
80	0.455378	0.337787	0.194639
160	0.456487	0.162446	0.001207
320	0.319734	0.001364	0.000927

(c) Best error, 1000 residuals.

Table 13: Tables for the ReLU models trained to approximate the 16-saw-teeth solution ($N = 4$).



(a) Target solution and Tanh model approximation.



(b) Target solution and ReLU model approximation.

Figure 21: Best approximations of the 16-saw-teeth solution by the first Tanh network trained with 3 hidden layers, 320 neurons per layer and 1000 residuals (left) and by the first ReLU network trained with the exact same architecture (right). Both show a very high level of accuracy, well below 1%.

Let us conclude the discussion of our first generic test with an important remark related to Theorem 2.2. Analyzing the aforementioned results we are forced to acknowledge that, even if some single-layered architectures showed well functioning instances for the studied cases, possessing a theoretically sufficient capacity for the prediction of a certain complex phenomenon does not guarantee a successful learning procedure for the relative model. Sometimes, as we appreciated above, we necessarily need to significantly enlarge the number of neurons per layer to employ in our shallow networks if we want to obtain satisfactory outcomes from our experiments. Nevertheless, in presence of particularly stiff saw-teeth solutions, this trick might not even be sufficient (see the ReLU models with only one hidden layers for the last part of this test). Increasing the number of hidden layers for the studied class of saw-teeth targets for this test case, instead, has proven to bring appreciable advantages.

Test 2

Let us now consider the simple ODE problem for the mentioned class of saw-teeth waves.

$$\begin{cases} u' = f & x \in (0,1) \\ u = 0 & x = 0 \end{cases} \tag{3}$$

For these experiments we chose to vary the architectural parameters in the following ranges:

- For the 4-saw-teeth solution, corresponding to $N = 2$:
  - Amount of uniformly distributed internal points over the computational domain $[0,1]$: $\{20, 40, 80\}$;
  - Employed neurons per layer: $\{2, 4, 8, 16, 32, 64, 128\}$;
  - Number of hidden layers: $\{1, 2, 3\}$.

- For the 8-saw-teeth solution, corresponding to $N = 3$:
  - Amount of uniformly distributed internal points over the computational domain $[0, 1]$: $\{50, 100, 200\}$;
  - Employed neurons per layer: $\{2, 4, 8, 16, 32, 64, 128, 256\}$;
  - Number of hidden layers: $\{1, 2, 3\}$.
- For the 16-saw-teeth solution, corresponding to $N = 4$:
  - Amount of uniformly distributed internal points over the computational domain $[0, 1]$: $\{250, 500, 1000\}$;
  - Employed neurons per layer: $\{5, 10, 20, 40, 80, 160, 320\}$;
  - Number of hidden layers: $\{1, 2, 3\}$.

During the learning phase of all models we performed 25000 ADAM iterations followed by 75000, 125000 and 175000 maximum LBFGS steps for these three tests, respectively. The reason why we boosted the training iterations for the targets with a higher number of linear pieces obviously resides in their increasing stiffness. As anticipated, we employed both ReLU and the hyperbolic tangent (Tanh) activation functions.
Our first comment concerns the surprising inability expressed by every ReLU network in approximating the rather simple and well-suited 4-saw-teeth solution. With no exceptions, all these models immediately got stuck in the zero-solution local minimum of the cost functional. This behavior appears to be extremely puzzling: we must remember, indeed, that with the exact same learning and architectural specifications, in the first part of Test 1 we managed to achieve good levels of accuracy for the same solution. The only detail that differs between the two tests regards the way by which we impose the target: in the former we explicitly resolve a pure approximation problem, while in the latter we implicitly provide the solution through its first derivative by means of an ODE. As for the previous single-scale and multi-scale sensitivity analyses (in sections 3.2.1 and 3.2.2), ReLU networks fail at grasping relatively simple solution targets when an actual derivation is contained in the expression of the PDE. In light of these results, we reasonably conjecture that there must be an unknown feature related to ReLU that is responsible for the mentioned issues when dealing with non-degenerate differential operators. Such a trend is *a fortiori* shown also in the second and third parts of the current test. This subject will be further investigated in one of our subsequent trials (Test 5), where we will dig into the details of this phenomenon. On the basis of what we just said we limit our considerations to Table 14, representing the best L^2 relative errors related to the models trained with the hyperbolic tangent activation function. We see that 20 uniformly distributed points are not enough to interpret the solution with satisfactory accuracy. Instead, employing 40 residuals, we appreciate several working architectures with some instances of the multi-layered networks reaching a relative L^2 error in the order of 1%-2%. Here, shallow models show performance levels that are positively correlated to an increasing total number of neurons per layer, reaching a saturation accuracy of about 95%. Using 80 training points leads to a very high accuracy (well past 99%) for these models, while multi-layered networks do not improve appreciably. Concerning the typical architectural parameters under study, in the last two cases that we described there are no general related patterns that emerge distinctly. In Figure 22 we represent the most successful approximating network of the 4-saw-teeth solution target for this part of the test, along with the related cost functional evolution. Figure 23, instead, shows the notorious behavior of the ReLU architectures, which appears whenever they are trained to learn the solution of a PDE that contains effective differential terms in its expression.

NPL/HL	1	2	3
2	0.920594	0.727893	0.504348
4	0.514314	0.692048	1.290939
8	0.641434	1.708846	0.435716
16	0.674161	0.460413	0.403932
32	0.522378	0.459221	0.430089
64	0.510950	0.429403	0.398886
128	0.770445	0.448640	0.513728

(a) Best error, 20 residuals.

NPL/HL	1	2	3
2	0.610074	0.416505	0.665898
4	0.396115	0.049787	0.047099
8	0.059533	0.250497	0.184904
16	0.062870	3.073549	0.021354
32	0.044272	0.021754	0.016591
64	0.049279	0.023817	0.021972
128	0.051243	0.186686	0.040941

(b) Best error, 40 residuals.

NPL/HL	1	2	3
2	0.687386	0.597666	0.431485
4	0.364336	0.048274	0.091481
8	0.058471	0.036431	0.014043
16	0.047344	0.064007	0.077940
32	0.008358	0.010178	0.051681
64	0.007155	0.017330	0.074364
128	0.006418	0.145762	0.018758

(c) Best error, 80 residuals.

Table 14: Tables for the Tanh models trained to approximate the 4-saw-teeth solution to (3).

Table 15 shows the best relative L^2 error results for the architectures trained in the second part of this test, built to approximate the 8-saw-teeth wave. With the selected maximum number of learning iterations, the models have not been able to give appreciable results neither with 50 nor with 100 residual points. The only case in which, given the mentioned learning configuration, we obtained acceptable results (with a relative L^2 error of about 8%-9%) refers to the instances trained with 200 uniformly distributed training points. Figure 24 provides a graphical representation of the best architecture obtained for this part of the test.

NPL/HL	1	2	3
2	0.709838	0.710325	0.828946
4	0.575349	0.628883	0.633521
8	0.417751	0.576178	4.218417
16	2.459462	1.186847	1.155442
32	0.395278	2.016019	0.477766
64	0.293250	0.168155	0.266048
128	0.227262	0.199784	0.195286
256	0.395510	0.153651	0.145828

(a) Best error, 50 residuals.

NPL/HL	1	2	3
2	0.520246	0.523033	0.504719
4	0.475513	0.551188	1.789424
8	0.442974	0.383959	0.295322
16	0.152286	0.280635	0.256253
32	0.237664	0.456316	3.017988
64	0.254598	0.645825	0.232782
128	0.260487	0.198131	0.216551
256	0.256340	0.209013	0.225106

(b) Best error, 100 residuals.

NPL/HL	1	2	3
2	0.785472	0.752976	0.775996
4	0.589097	0.452756	0.743016
8	0.367243	0.113602	0.150532
16	0.107657	0.139436	0.180609
32	0.098185	0.110316	0.088574
64	0.094483	0.086540	0.124224
128	0.095669	0.101022	0.105222
256	0.089450	0.104572	0.087223

(c) Best error, 200 residuals.

Table 15: Tables for the Tanh models trained to approximate the 8-saw-teeth solution to (3).

In Table 16 we eventually report the results concerning the approximations for the stiffest among our solution targets, the 16-saw-teeth wave. First of all we notice one more time the importance of the role associated to the number of residual points used in our experiments. Exploiting 250 of them, with the current learning settings, we are never able to reach an acceptable level of accuracy. Embedding our networks with 500 training points, we begin to spot sporadic instances where the relative L^2 error stays below 10%. However, it is only by using 1000 residuals that we start to appreciate visible patterns in the error table: fixing the number of hidden layers, the performance increases as the amount of neurons per layer grows, at least until it reaches a saturation level that depends on the former parameter. The best architectures are composed by two or three hidden layers. The most reliable network produced for this part of the test is also illustrated in Figure 25.

NPL/HL	1	2	3
5	0.519144	0.594262	0.546625
10	0.550090	0.645890	0.486174
20	0.394933	0.299214	0.478642
40	0.245825	0.182306	0.509257
80	0.171314	0.148630	0.686234
160	0.174021	0.148974	0.242684
320	0.174872	0.183394	0.139937

(a) Best error, 250 residuals.

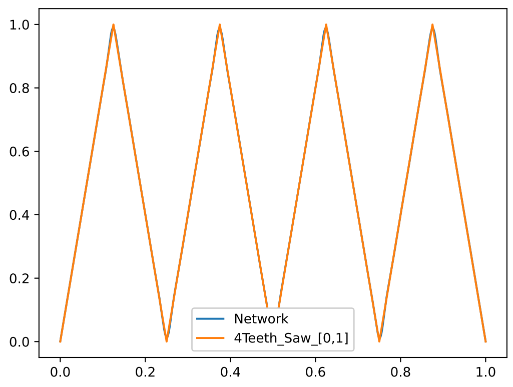
NPL/HL	1	2	3
5	0.754264	0.750128	0.684918
10	0.631826	0.813879	0.524691
20	0.453729	0.330968	0.608885
40	0.124598	0.084565	0.514669
80	0.129309	0.122246	0.347749
160	0.121357	0.100947	0.079897
320	0.111164	0.104554	0.109690

(b) Best error, 500 residuals.

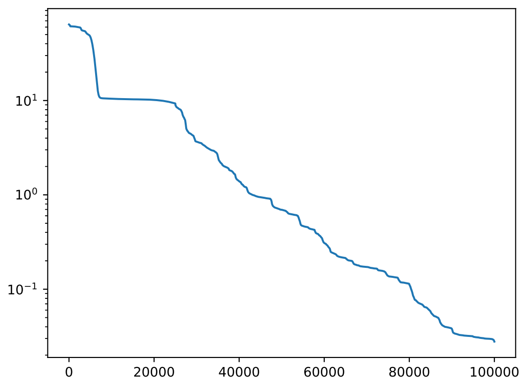
NPL/HL	1	2	3
5	0.552737	0.727696	0.462595
10	0.624218	0.458627	0.592527
20	0.360279	0.475390	0.575436
40	0.213597	0.063409	0.548513
80	0.068715	0.041562	0.397271
160	0.066581	0.048937	0.039099
320	0.070134	0.048792	0.044086

(c) Best error, 1000 residuals.

Table 16: Tables for the Tanh models trained to approximate the 16-saw-teeth solution to (3).

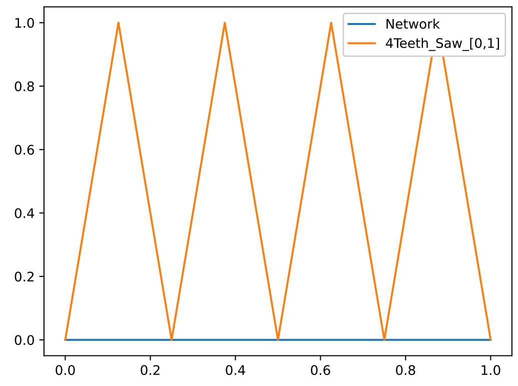


(a) Target solution and Tanh model approximation.

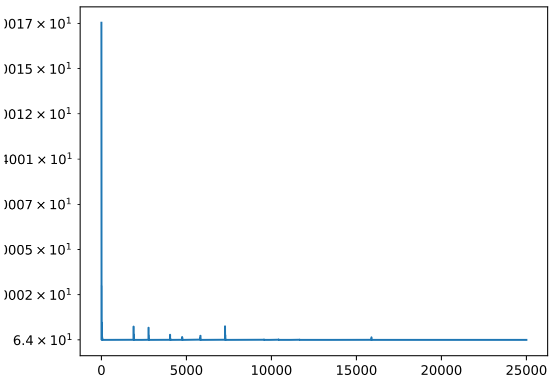


(b) Loss function evolution.

Figure 22: Best approximation plot of the 4-saw-teeth solution to (3) by the third Tanh network trained with 1 hidden layer, 128 neurons and 80 residuals (left) and its related cost functional evolution (right) in semi-logarithmic scale. As we can see from the latter, this model seemingly still had some unexpressed potential that would have probably become visible with more learning iterations.



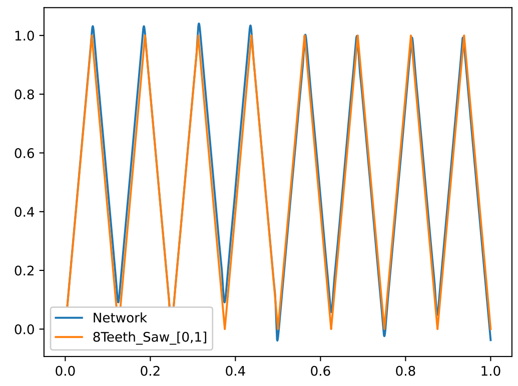
(a) Target solution and Tanh model approximation.



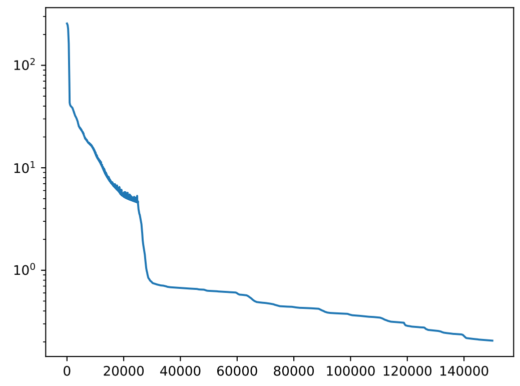
(b) Loss function evolution.

Figure 23: Poor approximation plot of the 4-saw-teeth solution to (3) by the first ReLU network trained with 3 hidden layers, 128 neurons per layer and 80 residuals (left) and its related cost functional evolution (right) in semi-logarithmic scale. This model closely represents all architectures trained with the ReLU activation function, which present identical characteristics and issues.

Looking at Figure 24 we see that, as for Figure 22, the best model that we obtained for the approximation of the 8-saw-teeth wave appears to have some unexpressed potential that could have become visible with more learning iterations. The final network state actually shows poor approximation capabilities close to the vast majority of the sharp corners characterizing the mentioned saw-teeth solution target, and the related cost functional does not seem to have reached convergence. It is therefore reasonable to assume that, with a longer learning procedure, they would have probably got adjusted. Even if this would have been the case, however, also in relation to what we observed in the first part of this test, we can ascertain that it seems to be much harder to interpret the (exact same) target profile when it is provided by means of a PDE (or ODE) rather than through its explicit expression. What interests us the most is the fact that, keeping all learning parameters unchanged, the higher the order of derivation appearing in the equation, the tougher it seems to interpret all the key features of the solution. We anticipate that the overall results obtained in Test 1, 2 and 3 are indeed compatible with the suggested hypothesis, at least for the studied cases.

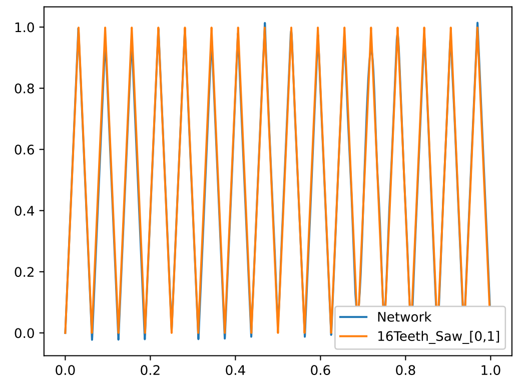


(a) Target solution and Tanh model approximation.

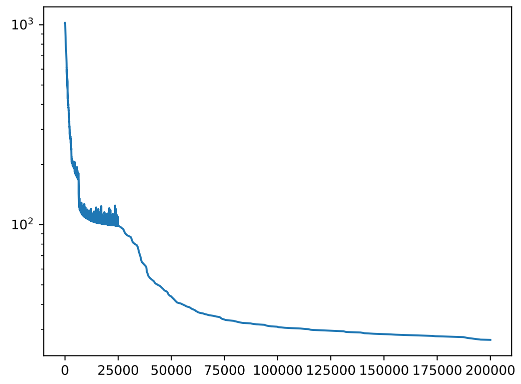


(b) Loss function evolution.

Figure 24: Best approximation plot of the 8-saw-teeth solution to (3) by the second Tanh network trained with 2 hidden layer, 64 neurons per layer and 200 residuals (left) and its related cost functional evolution (right) in semi-logarithmic scale. As we can see from the latter, this model seemingly had some unexpressed potential that would have probably become visible with more learning iterations. Also for this reason, the network still shows lacks of accuracy in correspondence to several sharp corners of the exact solution, as we can clearly notice from the left graph.



(a) Target solution and Tanh model approximation.



(b) Loss function evolution.

Figure 25: Best approximation plot of the 16-saw-teeth solution to (3) by the third Tanh network trained with 3 hidden layer, 160 neurons per layer and 1000 residuals (left) and its related cost functional evolution (right) in semi-logarithmic scale. The network shows lacks of accuracy in correspondence to several sharp corners of the exact solution, as we can clearly notice from the left graph. The cost function is still slowly decreasing at the end of the learning procedure.

A further important observation is in order. The conclusions made in the second part of this test, concerning the potential to better interpret the 8-saw-teeth wave if allowed to perform more learning iterations, are actually formally confirmed by the results obtained in the third (and last) part of this experimental trial. Indeed, in the latter we have been able to prove that it is possible to emulate an even stiffer solution, with a relative L^2 error in the order of 4%, using a set of structural parameters that is very similar to the one previously exploited. For completeness it has to be remarked that, other than a larger number of learning iterations, in the last part of this test we even embedded our architectures with much more residual points.

Test 3

Let us now consider the one-dimensional Poisson equation with homogeneous boundary conditions (1) upon the class of saw-teeth waves. Before proceeding, we need to clarify the sense with which we intend to resolve it.

$$\begin{cases} -u'' = f & x \in (0,1) \\ u = 0 & x \in \{0,1\} \end{cases}$$

In the expression above, we have written a *strong* formulation of the mentioned differential problem. However, assuming a saw-teeth wave to be the relative solution target, it is crucial to notice that we cannot interpret the equation in a classical way: the second order derivatives of such functions are actually not well defined in correspondence of their sharp corners. To be precise, this consideration clearly holds also for their first order derivative. Nevertheless, resolving the system (3) in an almost everywhere sense provides the expected expression for our solution. Here this rather technical detail, omitted in the presentation of Test 2 for simplicity, cannot be neglected because it plays a completely different and decisive role. Indeed, if we attempted to solve the system above in a similar fashion (interpreting it in an almost everywhere sense), we would be seeking for the zero-function solution, which is not our objective. The second order derivative for our class of functions, in fact, turns out to be identically null except for a finite number of points, where it cannot be classically defined. To overcome this intrinsic issue we treat the source term as a regularization of the distributional second order derivative of the solution target (with inverted sign), that consists in a sum of Dirac deltas (see Figure 26). Thanks to this expedient we are able to restore the consistency of our problem, which will be interpreted in the usual strong formulation. Due to the introduction of an erroneous regularization, however, the exact solutions of our new systems do not coincide with the saw-teeth waves, but with an approximation of theirs. An alternative approach, here not pursued, is to resort to the weak formulation of the problem (differently from [1, 35]). The regularized exact solutions' accuracy (with respect to the real saw-teeth targets) will depend on a positive real parameter, ϵ. The smaller its value, the (theoretically) closer the relative approximations to the desired functions: in the limit, we retrieve the exact target solutions.

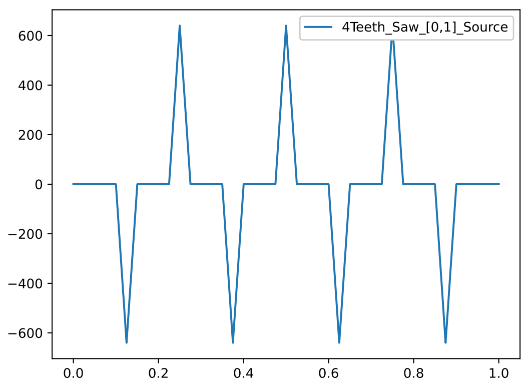


Figure 26: Regularized version of the sum of Dirac deltas corresponding to the source term (with inverted sign) in use for the 4-saw-teeth target solution ($N = 2$) to (1).

For these experiments we chose to vary the architectural parameters in the following ranges of values:

- For the 4-saw-teeth solution, corresponding to $N = 2$:
  - Fixed amount of uniformly distributed internal points over the computational domain $[0, 1]$: $\{400\}$;
  - Employed neurons per layer: $\{2, 4, 8, 16, 32, 64, 128\}$;
  - Number of hidden layers: $\{1, 2, 3\}$.
- For the 8-saw-teeth solution, corresponding to $N = 3$:
  - Fixed amount of uniformly distributed internal points over the computational domain $[0, 1]$: $\{800\}$;
  - Employed neurons per layer: $\{2, 4, 8, 16, 32, 64, 128, 256\}$;
  - Number of hidden layers: $\{1, 2, 3\}$.
- For the 16-saw-teeth solution, corresponding to $N = 4$:
  - Fixed amount of uniformly distributed internal points over the computational domain $[0, 1]$: $\{1600\}$;
  - Employed neurons per layer: $\{5, 10, 20, 40, 80, 160, 320\}$;
  - Number of hidden layers: $\{1, 2, 3\}$.

During the learning phase of all models we performed 25000 ADAM iterations followed by 75000, 125000 and 175000 maximum LBFGS steps for these three tests, respectively. The reason why we boosted the training iterations for the targets with a higher number of linear pieces obviously resides in their increasing stiffness. As anticipated, we employed both ReLU and the hyperbolic tangent (Tanh) activation functions. The (always large) amount of residuals points employed in our trials is calibrated in such a way that, for all models across the three tests performed, each regularization zone is embedded with an identical number of the former.
Our first consideration concerns, as usual, the fact that all ReLU models are definitely not able to grasp any feature of the target solutions under study. Once again, they all seem to fall immediately towards the zero-solution local minimum of the loss function, experimentally confirming our conjecture about their evident inability to understand the right training path when tested on problems involving effective differential terms. Table 17 collects the complete spectrum of best relative L^2 error results related to the architectures embedded with the hyperbolic tangent activation function, for all three tests performed.

NPL/HL	1	2	3
2	1.208059	1.009891	2.350983
4	2.010388	2.356984	1.000177
8	3.824559	2.410723	3.362979
16	6.120269	3.605160	3.583601
32	0.497861	3.788019	0.025933
64	0.158427	0.026248	0.028666
128	0.039368	0.026866	0.030798
\	\	\	\

(a) $N = 2$, 400 residuals.

NPL/HL	1	2	3
2	1.214229	1.012062	2.414002
4	1.491775	2.523375	2.366880
8	2.952532	2.466374	4.157850
16	3.400309	6.170633	6.203755
32	5.977035	7.325827	2.065365
64	2.289498	8.055386	1.102825
128	0.870293	0.079352	6.258685
256	0.226520	0.175243	0.067705

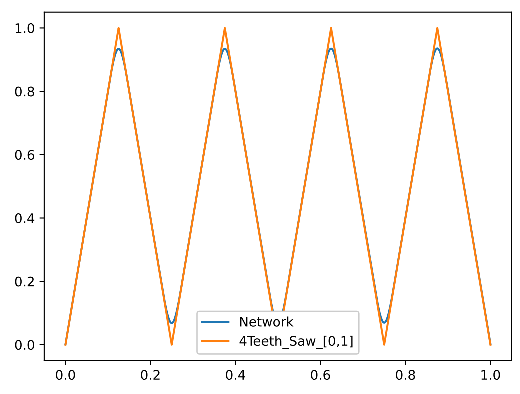
(b) $N = 3$, 800 residuals.

NPL/HL	1	2	3
5	0.541032	2.508024	2.605327
10	4.250449	3.224238	3.684910
20	5.817959	6.211009	13.806236
40	9.138324	3.032845	10.321249
80	7.907577	10.335821	10.610042
160	14.255289	23.221757	3.093698
320	6.860933	21.305189	14.751557
\	\	\	\

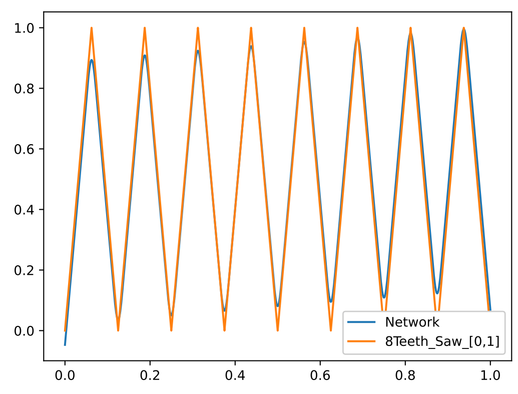
(c) $N = 4$, 1600 residuals.

Table 17: Best error tables for the Tanh models trained to approximate the 4-saw-teeth (left), 8-saw-teeth (middle) and 16-saw-teeth (right) solutions to (1) with the relative number of internal points.

For the 4-saw-teeth target, corresponding to $N = 2$, we appreciate approximation performances that highlight some visible patterns. First of all, keeping the amount of hidden layers fixed, the networks accuracy clearly increases with a growing number of neurons per layer until it reaches a more or less stable saturation level. Vice versa, fixing the latter and raising the former, we still see important improvements. In particular, wider architectures (embedded with more hidden layers) prove to reach a satisfactory performance needing fewer neurons per layer. The best models are obtained with two and three hidden layers, with a performance in the order of 97%-98%. Regarding the 8-saw-teeth solution, corresponding to $N = 3$, we do not find straightforward trends that emerge from data. A few important observations concern the fact that shallow networks have not been able to reach reliable approximations, and that the only two cases in which we see acceptable levels of accuracy are characterized by architectures with a large number of neurons per layer and either two or three hidden layers. In particular, we acknowledge success for at least one attempt among the networks with the following structures: 2 hidden layers endowed with 128 neurons and 3 hidden layers composed by 256 of them. We report, for completeness, also the results related to the stiffest saw-teeth solution under study, characterized by $N = 4$. As we can see from its relative L^2 error table, there is not even a single model that has been able to get even close to a consistent approximation of the target. The complexity of its source term is evidently too difficult to be reliably interpreted by these networks. In Figure 27, we represent the best network approximations for the first and second saw-teeth waves. We observe that both architectures present issues close to the sharp corners of the exact solution, where the slope suddenly changes sign. Given the evident lack of reliability shown simulating the 16-saw-teeth target for this test case, we do not report any related plot here.



(a) 4-saw-teeth solution and model approximation.



(b) 8-saw-teeth solution and model approximation.

Figure 27: Best approximation plots of the 4-saw-teeth solution by the second Tanh network trained with 3 hidden layer, 32 neurons per layer and 400 residuals (left) and of the 8-saw-teeth solution by the first Tanh network trained with 3 hidden layers, 256 neurons per layer and 800 residuals (right). Both architectures present lacks of accuracy in correspondence to all the sharp corners of the exact solutions to (1), where the regularization effectively takes place. Their entire loss evolution, not represented here, suggest that they expressed all the available potential during the relative learning phases.

In Table 17 we assume to be observing a non-disposable component of the error due to the inexact regularization of the exact solution's second order derivative, that can be formally defined just in a distributional sense. We therefore decreased the regularization parameter, ϵ, by a factor of 10, in order to explore the accuracy trend when the source term *tends* (in the specified sense) to the proper sum of Dirac deltas for the easiest saw-teeth target of our study. In order to keep the same number of residual points inside the key zones where regularization occurs, we accordingly augmented the number of internal training points. If on one hand we should expect to achieve a better approximation thanks to the improvement of the regularized source term, on the other we know that the latter becomes increasingly stiff when ϵ decreases. The resulting approximation unfortunately indulges our initial concern, showing a very poor accuracy even for the easiest test solution: optimization is unable to overcome the mentioned obstacle. On this basis, we would expect to observe an increasing error trend as ϵ tends to zero. This would mean that our networks are not suitable for this kind of approximation, failing at correctly interpreting the solution of our original problem under the lenses of this PDE through these learning settings. In order to conclude the description of this test we remark that, differently from the previous two, we decided to involve a much higher and uniquely fixed number of residual points here. This has been done with the aim of focusing on the essential structural parameters of our networks, willingly avoiding an analysis of the results with a dependency from the amount of training points exploited.

Take Home Messages

- ReLU seems to be an unreliable activation function when effective differential terms are involved in the governing equation. We will later discover the presumed reason that explains such failure (Test 5).
- Increasing the maximum order of derivation appearing in the PDE that implicitly expresses the (exact same) searched target leads to much more difficult and unstable learning procedures.
- Involving the results coming from the single-scale and multi-scale sensitivity analyses, we can confidently affirm that it is not necessarily true that it is more convenient to use deep architectures with respect to shallow networks. By means of the presented tests we found instances where, depending on the analyzed case, one choice was more suitable than the other. Moreover, we also referred to Theorem 2.2 to experimentally verify that shallow architectures having the theoretical capacity to emulate a specific objective function are not necessarily able to do so in a straightforward manner. As we saw, it could be necessary for them to be embedded with a very large number of neurons to obtain satisfactory results, while deep architectures are able to achieve comparable performances with fewer neurons. In this context we particularly refer to Test 1,2 and 3 for the models trained with ReLU.

We are now ready to discuss the results concerning a series of tests performed on a generic variety of Partial Differential Equations. Other than the usual Poisson equation, we will also explore several time-dependent differential formulations like the so-called Burger's equation, the Heat equation and a particular case of the Advection-Diffusion-Reaction problem. Let us now dig into the details for each of them.

Test 4

One of our main aims consists in understanding whether the ReLU activation function is, as it seems from Tests 2 and 3 of this section, really unable to provide reliable approximations of the solutions underlying non-degenerate PDEs (whose differential operator is not the identity), even when these targets are particularly suited for this activation function (as the saw-teeth profiles seen before). The aim of this experiment is twofold: investigating the behavior of our basic PINN on a simplified version of the Advection-Diffusion-Reaction (ADR) problem and observing the features that characterize ReLU even further. In the following, as anticipated, we consider the one-dimensional homogeneous ADR system where only advection is considered (with constant velocity v, set to 0.5). Recall (see subsection 2.2) that time-dependent PDEs are computationally treated by embedding the time variable t in the spacetime coordinate $\boldsymbol{x}$, letting it vary in our hyper-rectangular domain $\Omega = [0,1]^2$.

$$\begin{cases} \dfrac{\partial u}{\partial t} + v\dfrac{\partial u}{\partial x} = 0 & (x,t) \in \Omega \\ u = g & (x,t) \in \Gamma \subset \partial\Omega \end{cases} \tag{4}$$

Notice that, in order to provide a mathematically well-defined formulation, the boundary conditions for this problem are to be imposed only on a portion of $\partial\Omega$, corresponding to the *inflow boundary* of our domain:

$$\Gamma = \{(0,t)\ \forall t \in [0,1]\} \cup \{(x,0)\ \forall x \in [0,1]\}$$

This test is split in two parts, each corresponding to a different solution belonging to the class of saw-teeth functions, travelling with speed v. Differently from Test 2 and 3, where we attempted to solve a differential problem having a fully implicit saw-teeth target, here we actually provide a partially explicit hint for some features of the exact solution through the initial condition, having imposed the latter in a Dirichlet fashion. Keeping the meaning associated to parameter N as in the previous tests, for these experiments we chose to vary the architectural parameters in the following ranges:

- For the 1-saw-teeth travelling solution, corresponding to $N = 0$:
  - Amount of uniformly distributed internal points over the computational domain $[0,1]^2$: $\{100, 200\}$;
  - Employed neurons per layer: $\{10, 20, 40\}$;
  - Number of hidden layers: $\{1, 2\}$.
- For the 8-saw-teeth travelling solution, corresponding to $N = 3$:
  - Amount of uniformly distributed internal points over the computational domain $[0,1]^2$: $\{250, 500\}$;
  - Employed neurons per layer: $\{25, 50, 100\}$;
  - Number of hidden layers: $\{1, 2\}$.

During the learning phase of all models we performed 25000 or 50000 ADAM iterations followed by 75000 or 150000 maximum LBFGS steps, depending on the considered part of the test. The reason why we boosted the training iterations for the targets with a higher number of linear pieces (i.e. growing values of N) resides in their increasing stiffness. All networks possess 200 uniformly distributed boundary points on Γ. As always, we have employed both ReLU and the hyperbolic tangent (Tanh) activation functions for the sake of comparison.

NPL/HL	1	2
10	0.006036	0.002103
20	0.005392	0.001877
40	0.005026	0.001943

(a) Tanh, 100 residuals.

NPL/HL	1	2
10	0.006585	0.001750
20	0.005284	0.001657
40	0.005971	0.001989

(b) Tanh, 200 residuals.

NPL/HL	1	2
10	0.000276	0.000425
20	0.000477	0.001211
40	0.000923	0.001315

(c) ReLU, 100 residuals.

NPL/HL	1	2
10	0.000203	0.001151
20	0.000769	0.000921
40	0.000825	0.000822

(d) ReLU, 200 residuals.

Table 18: Best error tables for the models trained to approximate the 1-saw-teeth solution ($N = 0$) to (4) with the hyperbolic tangent and ReLU activation functions, with the related number of residuals.

NPL/HL	1	2
25	0.060264	0.032930
50	0.060249	0.023083
100	0.058477	0.019961

(a) Tanh, 250 residuals.

NPL/HL	1	2
25	0.060229	0.036502
50	0.059530	0.023483
100	0.059058	0.020410

(b) Tanh, 500 residuals.

NPL/HL	1	2
25	0.415206	0.337360
50	0.395114	0.332299
100	0.394417	0.074689

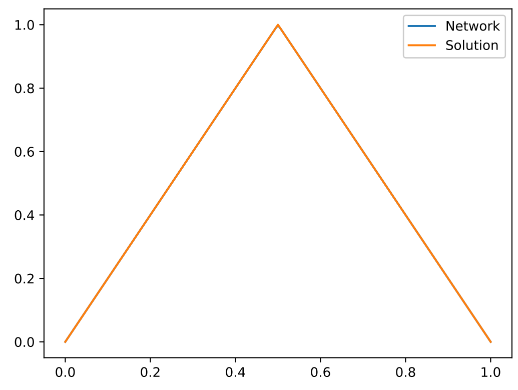
(c) ReLU, 250 residuals.

NPL/HL	1	2
25	0.438893	0.344116
50	0.419388	0.255373
100	0.358435	0.225070

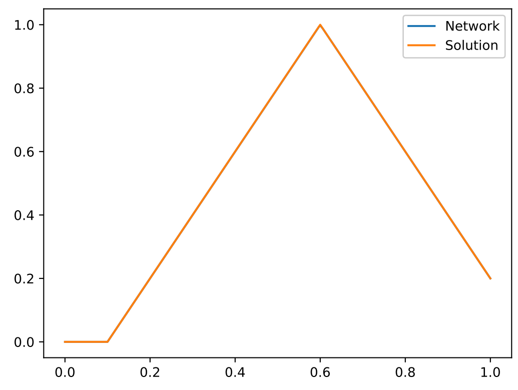
(d) ReLU, 500 residuals.

Table 19: Best error tables for the models trained to approximate the 8-saw-teeth solution ($N = 3$) to (4) with the hyperbolic tangent and ReLU activation functions, with the related number of residuals.

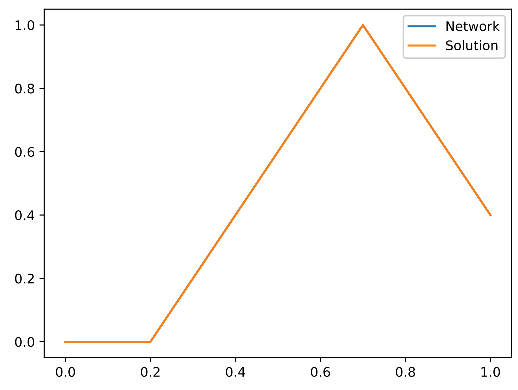
In the first part of the test (corresponding to $N = 0$), in terms of the evolution for the respective loss functions, all models have reached convergence. Table 18 refers to the related results, where we can appreciate reliable approximations for all of them. In particular, ReLU models show a relative L^2 error with a value close to one order of magnitude lower than the identically-structured (trained with the same values of the hyper-parameters) Tanh networks. Independently from the number of neurons per layer and the amount of residual points employed, shallow Tanh architectures present a constant performance with an error hovering around 0.5%-0.6%. When embedded with two hidden layers, the latter show an accuracy trend that remains nearly constant with respect to the number of neurons per layer, but slightly improves if more residuals are deployed. Regarding the ReLU networks, we do not see any identifiable pattern with respect to their architectural parameters. They generally provide, however, a better interpretation for the sharp travelling corner of the exact solution, as expected (see Figure 19). By imposing a PDE with explicit hints of some target features through the initial condition, even ReLU is able to grasp the underlying solution with excellent accuracy, at least for the simple target under study. In the second part of the test (corresponding to $N = 3$), Tanh models generally provide a loss function evolution that appears to have not reached convergence yet. It is therefore presumable that, with a larger number of learning iterations, they could have shown better results. These architectures represent, nevertheless, the only reliable instances for the approximation of the 8-saw-teeth solution. In the only case where a ReLU network has proven satisfactory levels of accuracy, its architecture is composed by 250 residual points, two hidden layers and 100 neurons per layer. Table 19 gathers all relative L^2 errors for the introduced test case. Looking at the evolution of their cost functional, we can confidently affirm that almost all of ReLU models have been rapidly trapped in a local minimum of the cost functional, being able to interpret only partially the solution's signature throughout the whole computational domain. Despite their clear potentialities, the loss function landscape proves to be extremely variegated for ReLU networks, at least for the proposed basic version of the PINN. Recalling the premise that Tanh models have not had the time to fully express their potential, we notice that the shallow architectures present a best error of approximately 5%-6%, while the double-layered models hover around 2%-3% with a slight improvement for an increasing number of neurons per layer employed. Figures 28 and 29 show the best functional approximations obtained for these experimental trials: the first refers to the travelling 1-saw-teeth, while the second is linked to the 8-saw-teeth moving target.
We conclude the discussion of this test with a series of useful remarks. On the basis of the obtained results we can ascertain that ReLU networks, when tested on PDEs that contain effective differential terms (characterized by a differential operator that does not coincide with the identity) with a simple (easy to be interpreted) solution, might be capable of emulating the desired features if provided with a hint of the target. Indeed, with a sufficiently complex solution, ReLU still proves to be unreliable. The networks trained with the hyperbolic tangent activation, on the other hand, show satisfactory accuracy levels in all cases (see Figure 29).



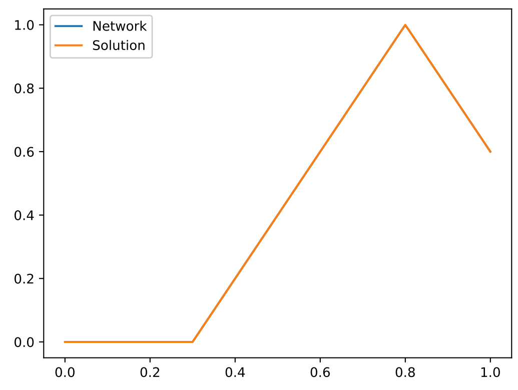
(a) Target solution and network approximation, $t = 0$.



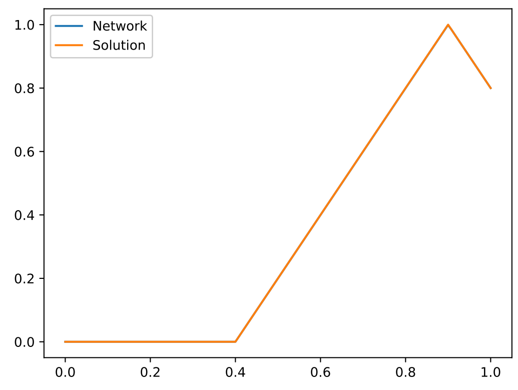
(b) Target solution and network approximation, $t = 0.2$.



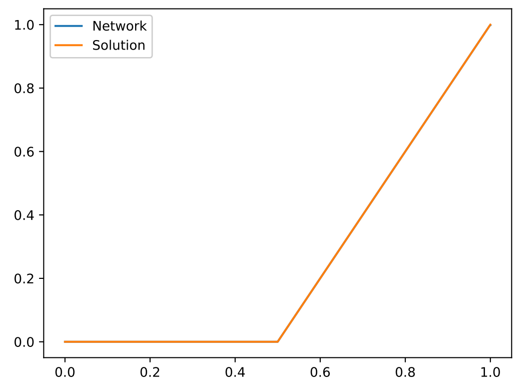
(c) Target solution and network approximation, $t = 0.4$.



(d) Target solution and network approximation, $t = 0.6$.

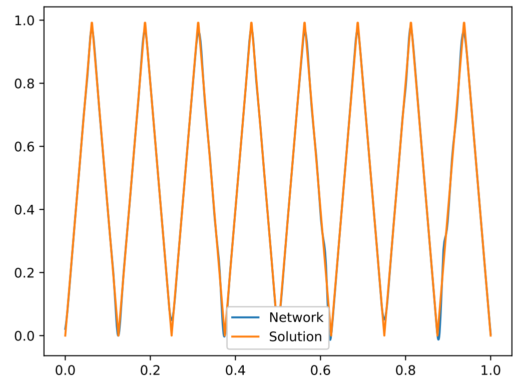


(e) Target solution and network approximation, $t = 0.8$.

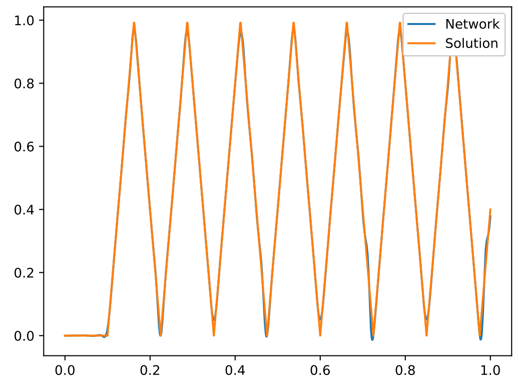


(f) Target solution and network approximation, $t = 1$.

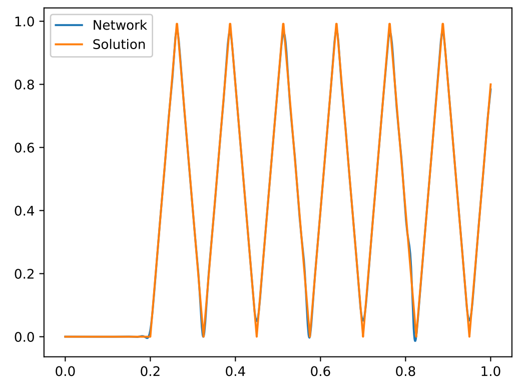
Figure 28: Best approximation plots, at six uniformly-spaced time steps, of the travelling 1-saw-teeth ($N = 0$) solution to (4) by the third ReLU network trained with 1 hidden layer composed by 10 neurons and 200 uniformly distributed residual points.



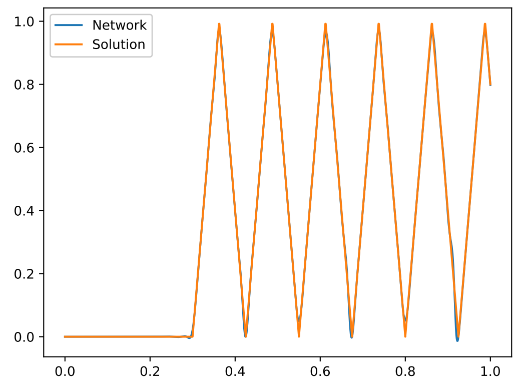
(a) Target solution and network approximation, $t = 0$.



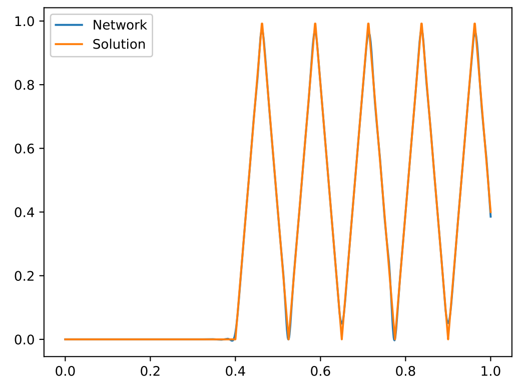
(b) Target solution and network approximation, $t = 0.2$.



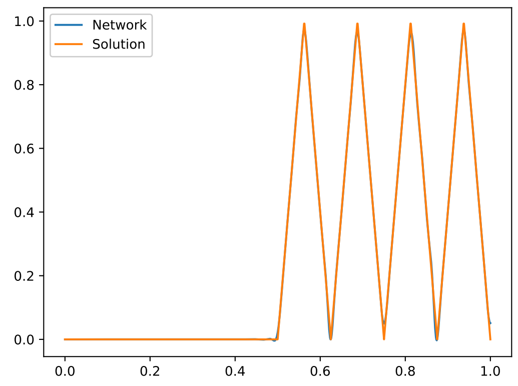
(c) Target solution and network approximation, $t = 0.4$.



(d) Target solution and network approximation, $t = 0.6$.



(e) Target solution and network approximation, $t = 0.8$.



(f) Target solution and network approximation, $t = 1$.

Figure 29: Best approximation plots, at six uniformly-spaced time steps, of the travelling 8-saw-teeth ($N = 3$) solution to (4) by the first Tanh network trained with 2 hidden layers composed by 100 neurons each and 250 uniformly distributed residual points.

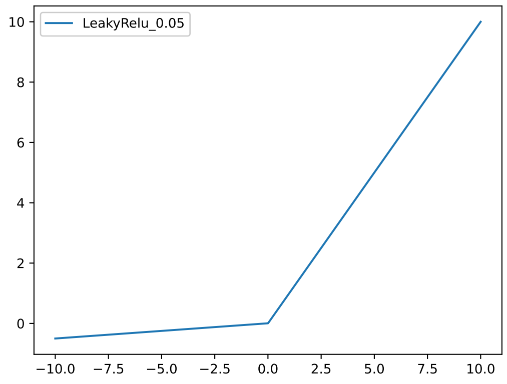
To sum up, every time we have dealt with a non-degenerate differential equation (that is to say a PDE whose operator is not the identity, as for Test 1 of this section), we have observed several difficulties associated with the networks trained with the ReLU activation function. The next test is mainly devoted to the search for the motivations hidden behind the transversal failure of ReLU architectures.
We conjecture two possible explanations for the failure proven by the architectures embedded with ReLU:

- Since this activation function identically evaluates to zero for the negative half-line of the real numbers, it might introduce *sparsity* in the gradient back-propagation inside the network.
- Differently from the hyperbolic tangent function, it has a discontinuous derivative in correspondence to the origin that may be the root of the unexpected behavior seen throughout our experiments.

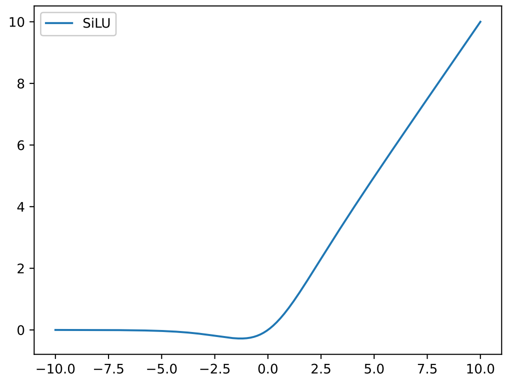
In order to discover which (if any) of these motivations concur in manifesting the mentioned issue, we employed two additional activation functions for our models: the so-called LeakyReLU and SiLU (see [11, 69]). The former should be able to avoid the possible problem consisting in the phenomenon known as *dying gradient*, related to the first point made above. The latter, instead, is an indefinitely regularized version of ReLU, expected to overcome the possible issue connected to the discontinuous derivative in the axis origin. The aforementioned activation functions have been graphically represented in Figure 30, and they can be formally defined as:

$$LeakyReLU(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

$$SiLU(x) = \frac{x}{1 + e^{-x}}$$



(a) LeakyReLU.



(b) SiLU.

Figure 30: Graphical visualization of the LeakyReLU (left) and SiLU (right) activation functional profiles over the domain [-10,10]. For the former plot, α has been taken equal to 0.05.

An important remark resides in the fact that LeakyReLU keeps the discontinuity of its derivative in the origin (as ReLU), except for the case in which $\alpha = 1$ (representing the linear profile). For our experiments, we decided to employ a value equal to 5% for this parameter, that defines the function's slope for $x < 0$.

Test 5

In order to search for the issue characterizing the ReLU activation function, while continuing our basic PINN analysis, we consider the simple one-dimensional Poisson equation with homogeneous boundary conditions (1). In this test case we impose a peculiar *bell*-like function (see Figure 31 (a)) as the solution target of our problem.

$$\begin{cases} -u'' = f & x \in (0,1) \\ u = 0 & x \in \{0,1\} \end{cases}$$

The models constructed for this experiment were prompted with a set of learning specifications and architectural parameters within the following ranges of values:

- Amount of uniformly distributed internal points over the computational domain $[0,1]$: $\{10, 20, 40, 80\}$;
- Employed number of neurons per layer: $\{10, 20, 40, 80\}$;
- Number of hidden layers: $\{1, 2\}$.

All models were trained with a total of 15000 ADAM iterations followed by another 35000 maximum number of LBFGS steps. As anticipated, the exploited activation functions are: Tanh, ReLU, LeakyReLU and SiLU. The current test has been inspired by the work shown in [35].

Tables 20, 21, 22 and 23 report the best relative L^2 error results for this experiment. In Table 20 we appreciate the trends shown by the Tanh models, which present excellent levels of accuracy when provided with more than 20 residual points. Even when just 10 training spots are exploited, all double-layered networks and a few instances of shallow architectures provide satisfactory performances. Moreover, the error typically decreases if any of the following parameters is increased: amount of neurons per layer, number of hidden layers or cardinality of the training set. A significant exception to the last point made can be observed using 80 residual points: in this case, the performance slightly worsens with respect to the models trained with only half of them. The best Tanh network is represented in Figure 31, alongside its loss function evolution.

NPL/HL	1	2
10	0.053937	0.053090
20	0.080961	0.041974
40	0.185773	0.019171
80	0.264085	0.053871

(a) 10 residuals.

NPL/HL	1	2
10	0.004502	0.000280
20	0.002051	0.000460
40	0.000592	0.000197
80	0.000055	0.000163

(b) 20 residuals.

NPL/HL	1	2
10	0.002190	0.000160
20	0.000028	0.000285
40	0.000067	0.000079
80	0.000149	0.000088

(c) 40 residuals.

NPL/HL	1	2
10	0.005263	0.000704
20	0.000768	0.000618
40	0.000213	0.001567
80	0.000064	0.000119

(d) 80 residuals.

Table 20: Best error tables for the models trained with the Tanh activation function.

Tables 21 and 22 are related to the architectures that exploit ReLU and LeakyReLU, respectively. Regardless of the differences between these two profiles, our new activation function does not show any sign of improvement with respect to ReLU. As a matter of fact, all the networks that have been trained with one of these activation functions eventually proved their incapability in finding reliable approximations even for the extremely simple solution target underlying our differential problem.

NPL/HL	1	2
10	1.000000	0.966978
20	1.069109	0.950462
40	0.958389	1.012757
80	0.949887	0.917786

(a) 10 residuals.

NPL/HL	1	2
10	0.836206	0.888710
20	0.916266	0.899800
40	0.839887	0.918538
80	0.991914	0.965247

(b) 20 residuals.

NPL/HL	1	2
10	0.716903	0.894153
20	0.958032	0.905396
40	0.974853	0.866829
80	0.875598	0.942498

(c) 40 residuals.

NPL/HL	1	2
10	1.009563	0.925407
20	0.962618	1.014600
40	0.901529	1.017370
80	1.008544	0.984397

(d) 80 residuals.

Table 21: Best error tables for the models trained with the ReLU activation function.

NPL/HL	1	2
10	0.799493	0.792292
20	0.818724	1.004341
40	0.828104	0.874943
80	0.916210	0.988886

(a) 10 residuals.

NPL/HL	1	2
10	0.688966	0.821222
20	1.020829	0.922211
40	0.899339	0.991628
80	0.968637	0.989566

(b) 20 residuals.

NPL/HL	1	2
10	0.781606	0.937788
20	0.774002	0.833073
40	0.969570	0.966451
80	0.988512	0.992599

(c) 40 residuals.

NPL/HL	1	2
10	0.992159	0.901548
20	0.854928	0.938832
40	0.862462	0.944870
80	0.865678	0.992672

(d) 80 residuals.

Table 22: Best error tables for the models trained with the LeakyReLU activation function.

Finally, in Table 23 we may appreciate the successful results characterizing the SiLU models. Differently from the Tanh networks embedded with the same structure, for these architectures 10 residual points are too few. Apart from this fact, we highlight stunning levels of accuracy, with a precision that permanently stays well above 99.9%. The only trend which clearly emerges from data concerns the relation between the number of internal training points and the general performance of the models. These two are, in fact, positively correlated with each other. The best SiLU network is represented in Figure 32, alongside its loss function evolution.

NPL/HL	1	2
10	0.258267	0.379755
20	0.245813	0.119700
40	0.312816	0.175850
80	0.453008	0.109568

(a) 10 residuals.

NPL/HL	1	2
10	0.000147	0.000098
20	0.000206	0.000078
40	0.000454	0.000115
80	0.000299	0.000314

(b) 20 residuals.

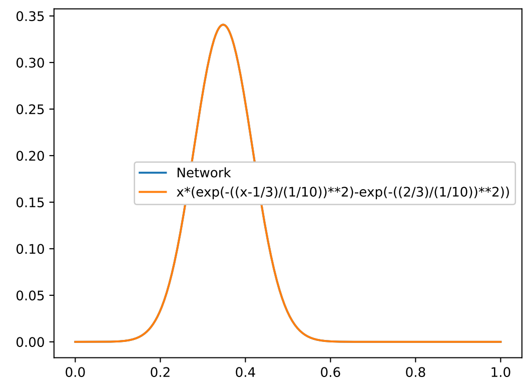
NPL/HL	1	2
10	0.000066	0.000103
20	0.000029	0.000020
40	0.000088	0.000043
80	0.000026	0.000030

(c) 40 residuals.

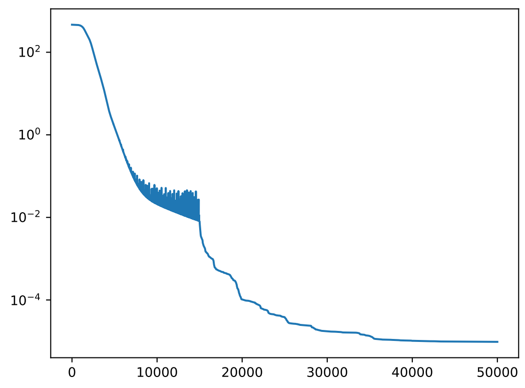
NPL/HL	1	2
10	0.000144	0.000046
20	0.000116	0.000023
40	0.000049	0.000010
80	0.000016	0.000030

(d) 80 residuals.

Table 23: Best error tables for the models trained with the SiLU activation function.

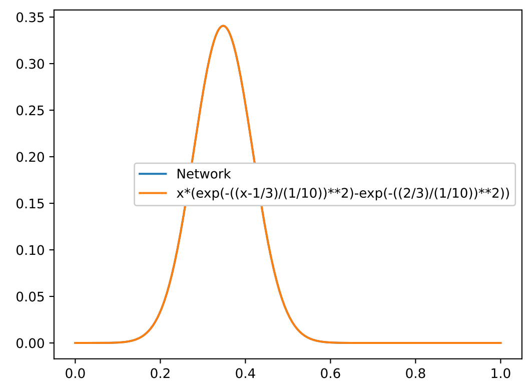


(a) Target solution and Tanh model approximation.

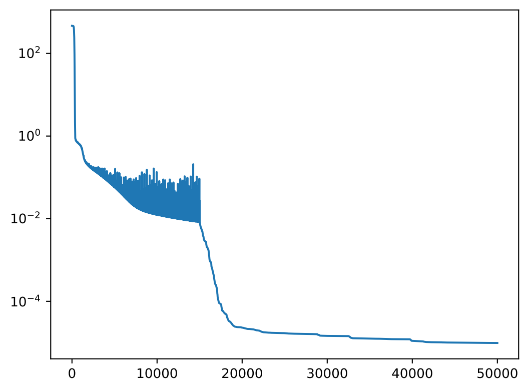


(b) Loss function evolution.

Figure 31: Best approximation plot of the target solution to (1) by the second Tanh network trained with 1 hidden layer, 20 neurons and 40 residuals (left) and its related cost functional evolution (right) in semi-logarithmic scale. Given the final flat shape of the latter after all the available LBFGS learning iterations were performed, we can confidently ascertain that convergence has been reached by this model. From the former, we see that the network and the exact solution plots basically overlap.



(a) Target solution and SiLU model approximation.



(b) Loss function evolution.

Figure 32: Best approximation plot of the target solution to (1) by the first SiLU network trained with 2 hidden layers of 40 neurons each and 80 residuals (left) and its related cost functional evolution (right) in semi-logarithmic scale. Given the final flat shape of the latter after all the available LBFGS learning iterations were performed, we can confidently ascertain that convergence has been reached by this model. From the former, we see that the network and the exact solution plots basically overlap.

Remarkably, we observe that the heaviest architectures (in terms of computational cost) trained for this trial took not more than a few minutes to complete their learning phase. For all our simulations, we have exploited a machine that employs a 2,6 GHz 6-Core Intel Core i7 processor and a 16 GB 2400 MHz DDR4 memory.
The inevitable conclusion that we infer on the basis of the exposed results consists in considering the lack of regularity in the first order derivative of ReLU as the main reason behind the failure of the related models throughout all tests performed so far. SiLU has actually been able to overcome, for the analyzed case, these obstacles in a brilliant manner. On the other hand, LeakyReLU shares the same issues of ReLU. If the unknown issue had been related to the dying gradient phenomenon, we would have expected to observe the opposite outcome. The successful activation profiles (tried until now) for PINNs have the common property of belonging to the set of analytical and indefinitely differentiable functions. In virtue of these considerations, in the next two tests we will limit ourselves to the analysis of the results coming from the models trained with the hyperbolic tangent activation profile, neglecting the other unsuccessful trials performed by means of ReLU.

Test 6

This experiment has been executed taking inspiration from an analogous test presented in [37]. In the present context we want to analyze the behavior of our basic PINN structure in understanding a very stiff target profile, namely the solution to the well-known one-dimensional homogeneous (regularized, with $\nu \neq 0$) Burger's equation over the spacetime domain $\Omega = [0,1]^2$:

$$\begin{cases} \dfrac{\partial u}{\partial t} + u\dfrac{\partial u}{\partial x} - \nu\dfrac{\partial^2 u}{\partial x^2} = 0 & (x,t) \in \Omega \\ u = g & (x,t) \in \Gamma \subset \partial\Omega \end{cases} \tag{5}$$

The initial and boundary conditions are imposed over the so-called parabolic frontier of the domain, that is:

$$\Gamma = \{(x_b, t)\ \forall t \in [0,1] : x_b \in \{0,1\}\} \cup \{(x,0)\ \forall x \in [0,1]\}$$

The models constructed for this test were prompted with a set of learning specifications and architectural parameters within the following ranges of values:

- Amount of uniformly distributed internal points over the computational domain $[0,1]^2$: $\{400, 800\}$;
- Employed number of neurons per layer: $\{25, 50, 100\}$;
- Number of hidden layers: $\{1, 2\}$.

All models were trained with a total of 25000 ADAM iterations followed by another 225000 maximum number of LBFGS steps. Overall 400 boundary points were exploited, 200 of which solely reserved for the imposition of the initial condition. As anticipated, we are essentially interested in studying the results coming from the models trained with the hyperbolic tangent activation function. In fact, once again, all ReLU networks provide extremely poor performances that are not reported here. Finally, ν is set to $2.5e-3$ and $g(x,t) = 1 - \tanh(200(x-t))$, which also represents the exact solution of the problem. The latter, as we can see from its analytical expression, is mainly characterized by an extremely sharp transitional interface in proximity of the spacetime line $t = x$, where its value suddenly (but smoothly) drops.

NPL/HL	1	2
25	0.000023	0.385417
50	0.385021	0.301815
100	0.368582	0.387447

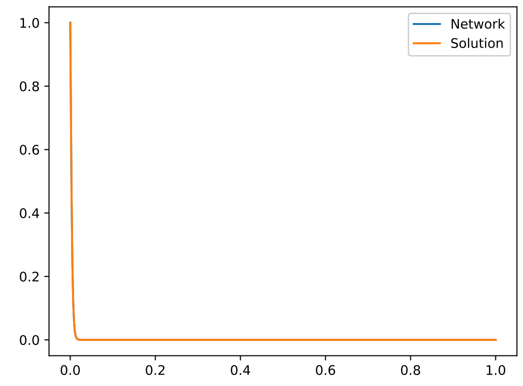
(a) 400 residuals.

NPL/HL	1	2
25	0.414848	0.000022
50	0.001711	0.370403
100	0.000250	0.412992

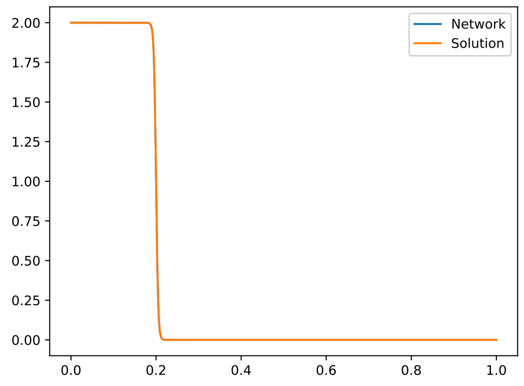
(b) 800 residuals.

Table 24: Best error tables for the models trained with the Tanh activation function.

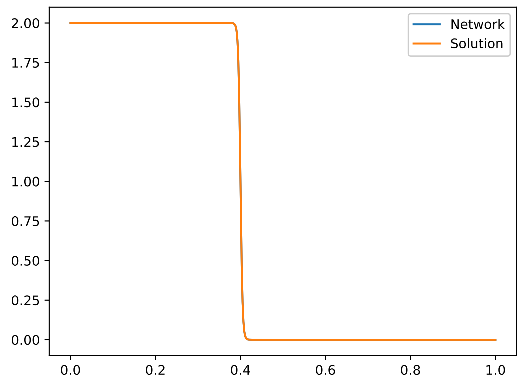
Table 24 shows the best relative L^2 errors for the models trained with the hyperbolic tangent activation function. It can be immediately seen that there are no patterns that appear to be generally valid for this case of study. For instance, the errors coming from the architectures trained with 800 residual points seem to suggest two contradictory conclusions. In fact, shallow networks present significative signs of improvements when the number of neurons increases, while double-layered models show the exact opposite trend. On the other hand, when we employ 400 internal training points, only the smallest architecture proves to be capable of obtaining a satisfactory accuracy, which in this case is even stunning. Embedding the models with twice as much residuals, we appreciate three successful network structures. One of the few reasonable conclusions that can be made on the basis of these results consists in the observation that a larger number of residuals seems to contribute in favor of a better performance. This fact is not surprising: we actually expect that, involving more training spots close to the stiff interface where the solution presents huge gradient values, our approximation naturally comes closer to the target. Concerning the rest we can say that, given the stiff nature of the phenomenon under study, it is probable that the initial random guess with which we prompt the learning phase of the models plays a crucial role in determining the quality of their final approximation. Even though most of the successful architectures are characterized by a single hidden layer, the best one is actually reached with two of them. As we said, however, augmenting the number of neurons per layer in the double-layered networks leads to unsatisfactory performances. On the basis of all these considerations, it seems reasonable to infer that the absence of emerging patterns for this test could be due to the difficult optimization procedure linked to such a stiff solution target, that introduces a large number of local minima for our loss function (especially for the double-layered structures that possess a larger amount of neurons per layer).



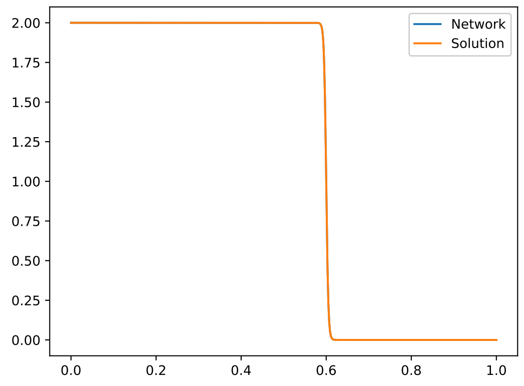
(a) Target solution and network approximation, $t = 0$.



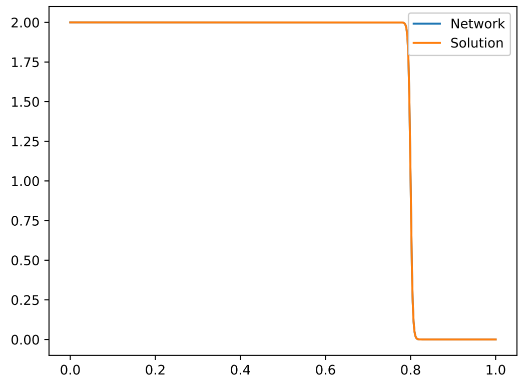
(b) Target solution and network approximation, $t = 0.2$.



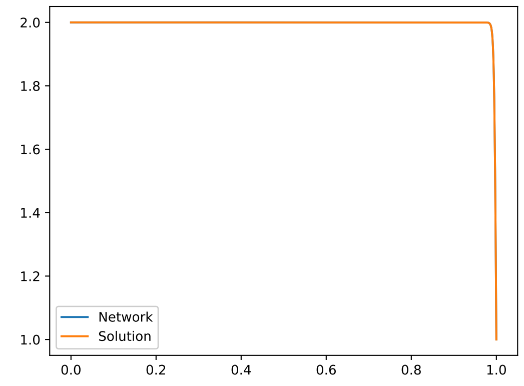
(c) Target solution and network approximation, $t = 0.4$.



(d) Target solution and network approximation, $t = 0.6$.



(e) Target solution and network approximation, $t = 0.8$.



(f) Target solution and network approximation, $t = 1$.

Figure 33: Best approximation plots, at six different time steps, of the target solution to (5) by the second Tanh network trained with 2 hidden layers of 25 neurons each and 800 residuals.

Figure 33 above illustrates the target solution's profile at six different time steps, as well as the best model approximation that we have managed to obtained for this test. We can clearly appreciate that, in all of the following representations, they almost coincide.
We observe that the heaviest architectures (in terms of computational cost) trained for this experiment took approximately 2-3 hours to execute their whole learning phase.

Test 7

The last experiment concerning our basic PINN sensitivity analysis involves another classical Partial Differential Equation, the one-dimensional homogeneous Heat problem over the spacetime domain $\Omega = [-1, 1] \times [0, 1]$:

$$\begin{cases} \dfrac{\partial u}{\partial t} - \nu \dfrac{\partial^2 u}{\partial x^2} = 0 & (x, t) \in \Omega \\ u = g & (x, t) \in \Gamma \subset \partial\Omega \end{cases} \tag{6}$$

The initial and boundary conditions are imposed over the so-called parabolic frontier of the domain, that is:

$$\Gamma = \{(x_b, t)\ \forall t \in [0, 1] : x_b \in \{-1, 1\}\} \cup \{(x, 0)\ \forall x \in [-1, 1]\}$$

We chose g so that the exact solution is a highly regular function characterized by an exponentially decaying sinusoidal profile given by $g(x, t) = \exp(-t)\sin(3\pi x)$ (therefore ν is correspondingly set to the value of $1/9\pi^2$). The models built for this test were prompted with a set of learning specifications and architectural parameters within the following ranges:

- Amount of uniformly distributed internal points over the computational domain $[-1, 1]^2$: $\{100, 200, 400\}$;
- Employed number of neurons per layer: $\{25, 50, 100\}$;
- Number of hidden layers: $\{1, 2\}$.

All models were trained with a total of 25000 ADAM iterations followed by another 125000 maximum number of LBFGS steps. An overall amount of 400 boundary points were exploited, 200 of which solely reserved for the imposition of the initial condition. As we anticipated at the end of Test 5, we are essentially interested in studying the results coming from the models trained with the hyperbolic tangent activation function. Also in this case, all ReLU networks provide extremely poor performances that are not reported here.

NPL/HL	1	2
25	0.061475	0.038649
50	0.038218	0.156886
100	0.046123	0.209087

(a) 100 residuals.

NPL/HL	1	2
25	0.010572	0.001866
50	0.007145	0.005628
100	0.004237	0.007973

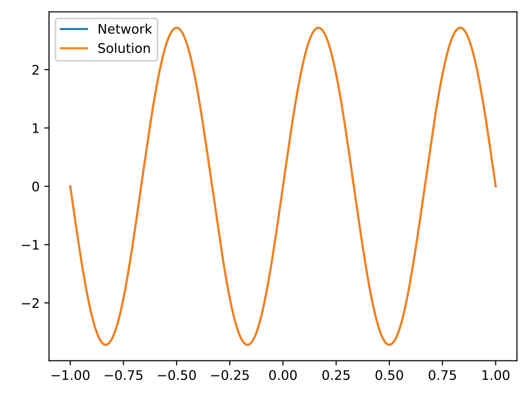
(b) 200 residuals.

NPL/HL	1	2
25	0.006809	0.000294
50	0.001248	0.000478
100	0.002705	0.000960

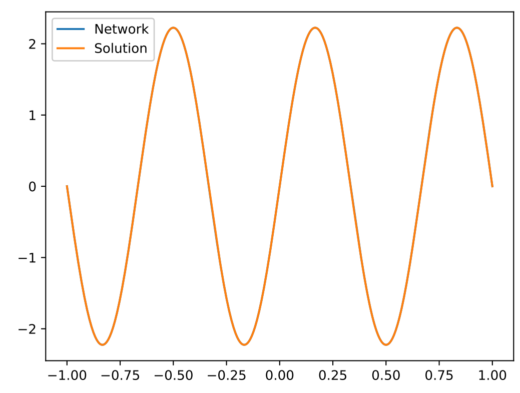
(c) 400 residuals.

Table 25: Best error tables for the models trained with the Tanh activation function.

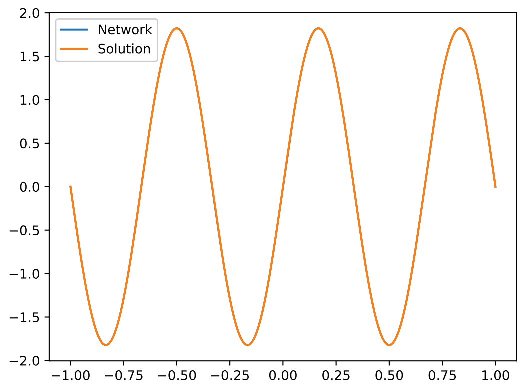
Table 25 shows the best L^2 relative error results related to the networks trained with the hyperbolic tangent activation function. The overall evaluation concerning the quality of our basic PINN structure on this experiment is positive. Moreover, all models show a loss function evolution that presents a plateaux at the end of the learning phase (hence, we can assume that they have had enough time to reach convergence) The architectures endowed with the lowest number of training points present performances that do not exhibit any identifiable accuracy trend with respect to an increasing number of hidden layers or neurons per layer. Nevertheless, all shallow models and the smallest double-layered network provide satisfactory levels of accuracy with a relative error in the 3%-6% range. Exploiting 200 residual points, we always obtain a best accuracy of at least 99%. Moreover, in this case, increasing the number of neurons per layer yields an improvement for shallow networks and a deterioration for multi-layered models. By employing architectures endowed with 400 internal training points, we appreciate the following patterns: multi-layered networks drop their accuracy with an increasing number of neurons per layer, while the best shallow model is obtained in correspondence to an intermediate amount of neural connections. The most influential parameter is clearly represented by the number of residual points: performances improve as it is increased.
Figure 34 below illustrates a six time-frame evolution of the target solution and the best model approximation that we managed to obtained for this experimental trial. As we can see, in all these representations they basically overlap with each other. Notice that the amplitude of the sinusoidal solution diminishes over time.
The heaviest architectures (in terms of computational cost) trained for this trial took approximately 10-15 minutes to execute their whole learning phase.



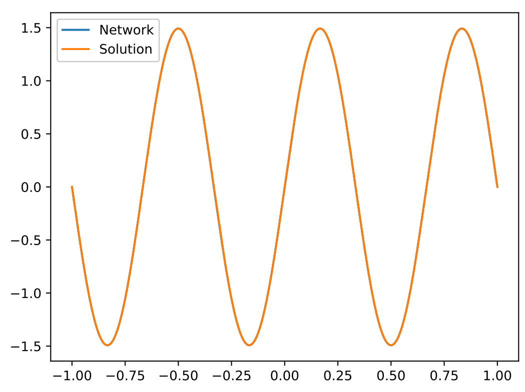
(a) Target solution and network approximation, $t = 0$.



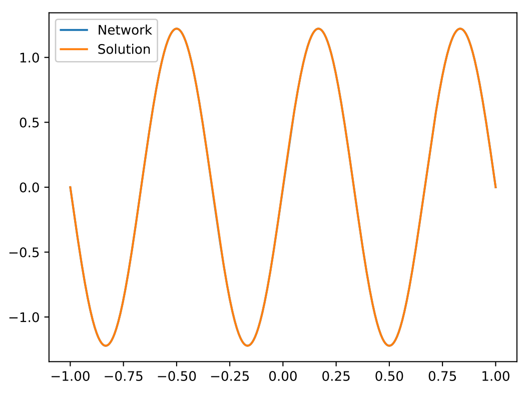
(b) Target solution and network approximation, $t = 0.2$.



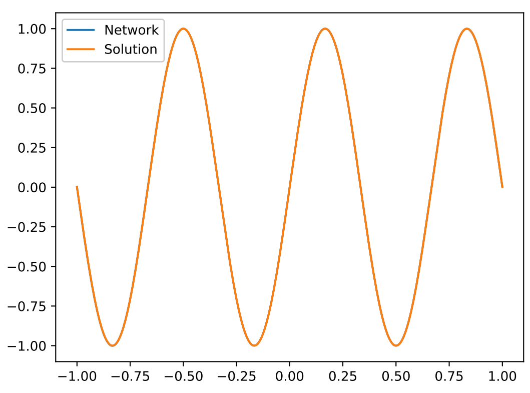
(c) Target solution and network approximation, $t = 0.4$.



(d) Target solution and network approximation, $t = 0.6$.



(e) Target solution and network approximation, $t = 0.8$.



(f) Target solution and network approximation, $t = 1$.

Figure 34: Best approximation plots, at six different time steps, of the target solution to (6) by the first Tanh network trained with 2 hidden layers of 25 neurons each and 400 residuals.

Take Home Messages

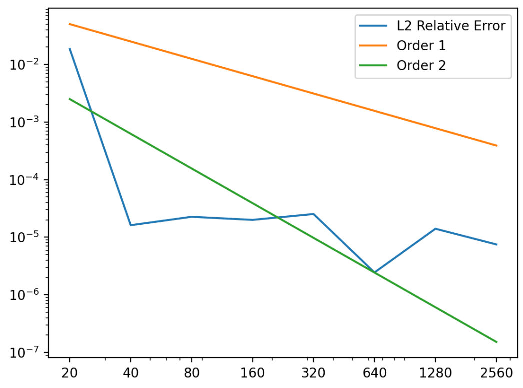
- The outcomes of Test 4 definitively discouraged the usage of ReLU: even when we provide a hint (through the initial condition) of a target profile that is suited for this activation function, in fact, the related models fail at obtaining satisfactory approximation performances unless the underlying solution is extremely simple (as for the first part of the mentioned test, corresponding to $N = 0$).
- Test 5 shed light on the issue concerning ReLU networks, proving that the most reasonable explanation for such failures resides in the discontinuity of the related first order derivative in the axis origin.
- Tests 6 and 7 have been useful to study the behavior (and the potentialities) of our basic PINN over two classical time-dependent PDEs, with the latter that presents a particularly stiff target profile. Their related outcomes are more than positive, even though for the Burger's problem we did not manage to obtain straightforward results concerning the performance trends with respect to the structural hyper-parameters of the models (probably due to the stiffness of the exact solution).

3.3. Convergence Analysis

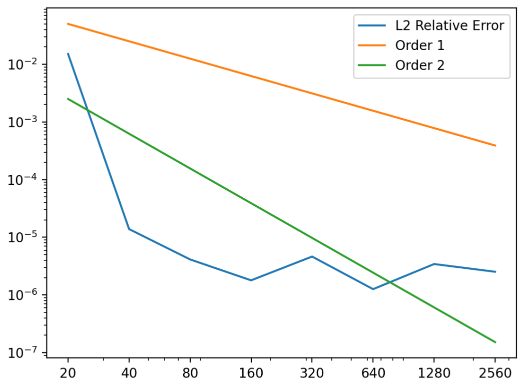
The aim of this subsection consists in presenting the experimental convergence properties, with respect to an increasing number of residual points, of the basic PINN structure, by testing the latter on three different types of solution targets: single sinusoid, multi frequency function and a generic bell-like profile, all implicitly imposed through the solution of the one-dimensional Poisson problem with homogeneous boundary conditions (1). Some related formal results and computational examples can be found in [60]. For this study, rather than their absolute performances (measured through the relative L^2 error), we are much more interested in the asymptotic behavior shown by the accuracy presented by the trained models (all embedded with the hyperbolic tangent activation function).

3.3.1 Single-Scale

For this test we employed the medium frequency solution that we already encountered in the single-scale sensitivity analysis of sub-subsection 3.2.1. The structure of our basic PINN is fixed, embedded with a single hidden layer and 100 neurons. Three identical attempts (with a different random initialization seed) were performed for all architectural specifications, involving a total of 25000 ADAM iterations followed by a maximum of 100000 LBFGS steps. In order to exploit the full potential of these networks and accordingly focus uniquely on their convergence properties, a sufficiently large amount of learning iterations have been set and performed. The number of uniformly distributed residual points inside the domain $[-1, 1]$ ranges in the following list of values: $\{20, 40, 80, 160, 320, 640, 1280, 2560\}$.



(a) Average trend plot.



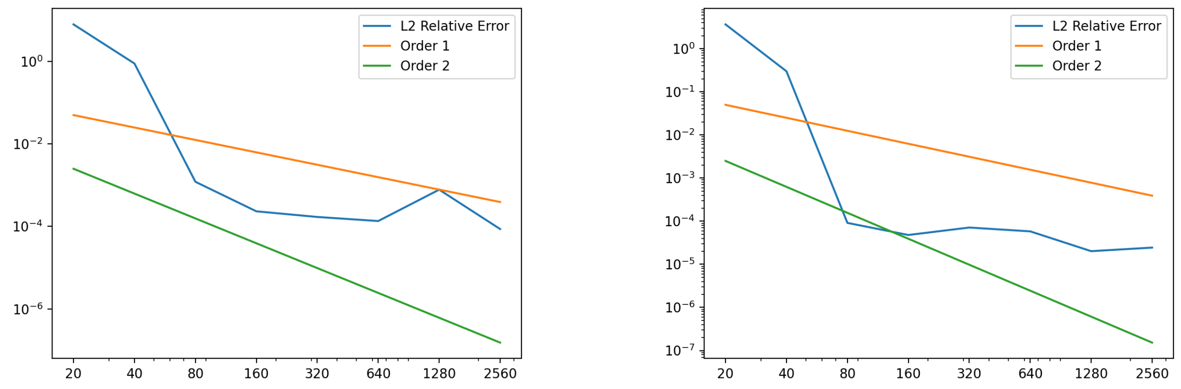
(b) Best trend plot.

Figure 35: Average (left) and best (right) error plots for the basic PINN single-scale convergence analysis (with respect to the number of uniformly distributed residual points employed) visualized in semi-logarithmic scale. The lines associated to the linear and quadratic orders lines are represented.

Figure 35 represents the average (left) and best (right) relative L^2 error trends for this experiment in semi-logarithmic scale. As we can see the two profiles are very similar to each other, confirming that the networks had enough time to fully exploit their approximation features. It is clearly visible that 40 residuals represent the threshold to obtain an extremely reliable emulation of the solution, with a relative error that stays well below 1%. Exploiting a higher number of training points leads to a level of saturation in the order of $1e-5/1e-6$.

3.3.2 Multi-Scale

For this experiment we reuse the multi-scale solution to (1) that we already exploited in the first test case of the homonym sensitivity analysis of sub-subsection 3.2.2 (for the same differential problem, that is the one-dimensional Poisson equation with homogeneous boundary conditions). The structure of our basic PINN is kept fixed to an architecture with a single hidden layer constituted by 100 neurons. Also in this case we performed three attempts for every set of architectural specifications, involving a total of 25000 ADAM iterations followed by a maximum of 100000 LBFGS steps. In order to exploit the full potential of these networks and accordingly focus uniquely on their convergence properties, a sufficiently large amount of learning iterations have been set and performed. The number of uniformly distributed residual points inside the domain $[-1, 1]$ ranges in the following list of values: $\{20, 40, 80, 160, 320, 640, 1280, 2560\}$.



(a) Average trend plot. (b) Best trend plot.

Figure 36: Average (left) and best (right) error plots for the basic PINN multi-scale convergence analysis (with respect to the number of uniformly distributed residual points employed) visualized in semi-logarithmic scale. The lines associated to the linear and quadratic orders lines are represented.

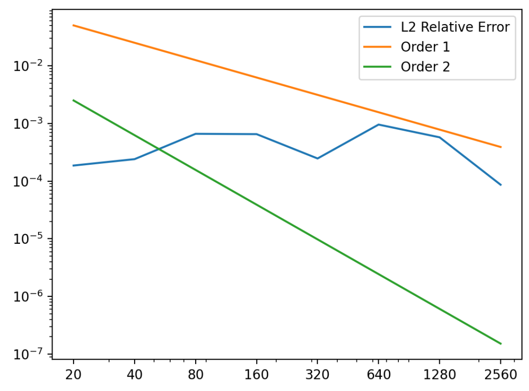
Figure 36 reports the similar evolution for the average (right) and best (left) relative L^2 error trends for this experiment. Similarly to the previous study, we appreciate a generically increasing performance in correspondence of an augmented number of residual points employed. Even these networks, however, reach a saturation level for their accuracy (close to $1e-4$) when the amount of training points exceeds 80. The performances obtained with a lower number of them proved to be thoroughly unsatisfactory.

3.3.3 Generic

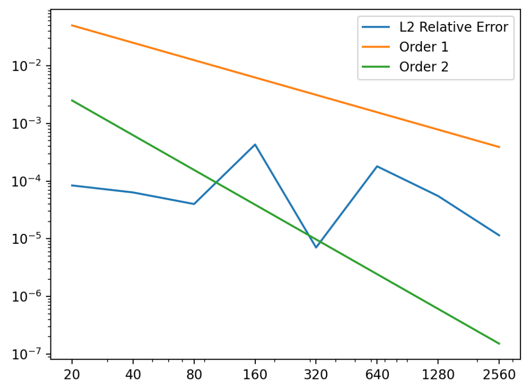
We conclude this brief analysis with another trial, where we impose the bell-like profile (taken from [35]) that we considered for Test 5 in sub-subsection 3.2.3 as the solution to the differential problem (1):

$$u(x) = x\left(e^{-100(x-1/3)^2} - e^{-400/9}\right)$$

The structure of the following networks is kept fixed to an architecture with two hidden layers composed by 80 neurons each, belonging to the settings that showed the best results in the aforementioned test. As usual we performed three attempts for every set of architectural specifications, involving a total of 25000 ADAM iterations followed by a maximum of 175000 LBFGS steps. In order to express the full potential of these networks and accordingly focus uniquely on their convergence properties, a sufficiently large amount of learning iterations have been set and performed. The number of uniformly distributed residual points inside the domain $[0, 1]$ ranges in the following list of values: $[20, 40, 80, 160, 320, 640, 1280, 2560]$.
Figure 37 shows a similar evolution for the average (left) and best (right) relative L^2 error patterns for this experiment. Differently from the other tests, we ascertain a constant trend for the models performances with respect to the total number of training points employed, with a value of the error that hovers around $1e-4$.



(a) Average trend plot.



(b) Best trend plot.

Figure 37: Average (left) and best (right) error plots for the basic PINN generic convergence analysis (with respect to the number of uniformly distributed residual points employed) visualized in semi-logarithmic scale. The lines associated to the linear and quadratic orders lines are represented.

The best models obtained for the approximation of the solutions to these convergence tests show a functional representation that coincides with the related targets. Their respective plots are essentially identical to the ones we already presented in section 3.2 for their counterpart sensitivity experiments (Figures 9 (a), 14 (a), 31 (a)). The most important conclusion related to the present analysis resides in the confirmation of the experimental evidence shown in [60], where the relative L^2 error trend linked to the approximation of a set of regular solutions through the one-dimensional Poisson problem reaches a saturation level with respect to the number of uniformly distributed residual points employed. Our work exhibits and repeats the same pattern for three different types of target profiles for the same differential system embedded with homogeneous boundary conditions (1).

The analyses and considerations made so far all belong to the original scope of this work, whose aim is to give a general overview of the basic PINN behavior through a series of classical test cases that are ubiquitous in literature. The main focus of the previous sections was to study and linger on the architectural features of the models trained to approximate the target functions that are passed, in a more or less implicit manner (depending on the case), through the imposition of a Partial Differential Equation. Provided with a sufficiently long learning procedure, the mentioned architectural characteristics are essentially represented by the following list of parameters: number of hidden layers, amount of neurons per layer and number of internal residual points. The latter, in particular, have been uniformly sampled inside the computational domain of the problems treated in the previous sections. This choice has been made for several reasons, among which the most important ones concern consistency and reproducibility of our experiments. In this context we transversely evaluated the performances shown by different activation functions, discovering that the presence of effective differential terms in (presumably) any PDE leads to favor the indefinitely regular profiles over the ones that present critical points. An example with the former characteristics is represented by the hyperbolic tangent function, which turned out to be extremely reliable in this respect. On the contrary, ReLU, that presents a discontinuity for it first derivative in the origin, shows poor abilities whenever any differential term appears in the equation governing the phenomenon under study. Such irregularity evidently impacts on the complexity of the cost functional landscape, that becomes excessively difficult to be explored by our optimization procedure. These conclusions were drawn after a proper investigating for the reasons behind such failure in one of the previous tests. After a complete sensitivity analysis where we studied the influence of the aforementioned parameters on the relative networks performances, we experimentally investigated the PINNs convergence properties over a set of simple cases. Concerning this topic, we eventually observed an interesting saturation pattern that is present in all our experiments. This trend profoundly differs from the results coming from other computational methodologies employed for the resolution of PDEs, such as FEMs. These, indeed, are embedded with strong theoretical foundations and a set of formal results regarding their convergence features which have also been experimentally verified (see [3]). Due to a lack of tangible theorems in the field of Artificial Neural Networks, we can consider our tests to be valid only in an experimental sense.
In some of our trials we observed better performances related to shallow networks, while in other tests we saw that multi-layered models were able to provide more accurate results. Even if with wider layers we typically obtained better approximations, we cannot classify this observation as generally valid: in some situations, in fact, too many neurons lead to architectures with a larger generalization error.

Continuing our discussion, we can certainly regard the total amount of learning iterations performed as another key variable for obtaining a satisfactory accuracy from our models. This fact should come with no surprise: indeed, the stiffer a solution target, the longer we expect our learning procedure to take in order to understand and emulate it correctly.
On the basis of all the previous considerations, we will dedicate the next section to the description and testing of a new computational tool for the numerical resolution of PDEs with a new adaptive algorithm for PINNs.

4. Adaptive PINN

In this chapter we propose an alternative version of the basic PINN endowed with a series of adaptive features, based on a set of heuristical assumptions, aiming at providing a substantial help in determining the proper structure to be used for the resolution of the related problem. This procedure is therefore intended to select the right number of hidden layers, their relative size and the amount of residual points to be employed, along with their location. It is needless to say that such an algorithm requires a stopping criterion which is expected to end in a proper configuration for the network. Also remember that, as general as an adaptive algorithm might be, it will always certainly need a set of hyper-parameters to be manually prompted with. Before passing to the detailed description of our proposal, we present a non-exhaustive list of adaptive techniques already presented in literature, accompanied by a short description.

- Residual Adaptive Refinement ([37]): an iterative technique used not only to adaptively increment the number of internal training points, but also to pinpoint their convenient location inside the domain.
- Time Adaptive Approaches ([67]): a pair of useful methods to address the resolution of time-dependent PDEs by sampling the residuals in specific spacetime portions during the learning phase.
- Loss Terms Adaptive Weights ([64, 66]): introduction of a set of adaptive weights multiplying the terms composing the cost functional expression.
- Individual Loss Terms Adaptive Weights ([39]): introduction of a set of adaptive weights multiplying the single individual terms of the cost functional.
- Adaptive Activation Function ([24]): technique that endows the network's activation profile with an adaptive coefficient varying in the learning phase of the model.

In the following we will construct an adaptive procedure that only focuses on the network architectural features. The latter will combine the renowned Growing Method (see for instance [12, 14, 23, 31, 34, 40, 55]), developed in the general field of Machine Learning, and a variation of the just mentioned Residual Adaptive Refinement technique. A first justification for their combined use resides in the fact that, despite being two different concepts, they are strictly correlated with each other. It is actually intuitive to assume that, in front of a stiffer solution, we will consequently need more residual points and, at the same time, a larger architecture, whether this means adding more neurons or increasing the number of hidden layers. All the related implementative details will be discussed in the next subsection, after which we shall test the proposed procedure through a campaign of significant experiments. Before entering into the aforementioned expositions, a crucial premise is in order. Indeed, we must specify that the following results are to be intended as trials of our newborn algorithm, and that all the consequent considerations must be accordingly taken with caution. It is important to contextualize their significance in terms of mere initial experimental attempts in a new (and possibly promising) direction for the development of adaptive PINNs, that nonetheless still remains unpretentious.

4.1. Implementation

We are finally ready to present the outline of our adaptive algorithm, which is intended to be the building block for the construction of a properly sized and highly performing PINN. As we already remarked, our aim is to combine the well-known Growing Method and a variation of the so-called Residual Adaptive Refinement technique, focusing our attention on the related architectural aspects. A first fundamental assumption underneath this procedure consists in considering the network's physical structure and the number of residual points to be employed as interconnected with each other. In fact, we might heuristically link stiffer solutions to the need of larger models and a higher number of training points. It is for this reason that, in our adaptive procedure, the operative steps responsible for these two enhancements intervene at the same time. In the following we present the exhaustive list of additional hyper-parameters considered for our new optimization procedure, which is still based on the combined ADAM - LBFGS learning technique, already exploited to construct our basic PINNs.

- ***Pools_Residuals_Size***: cardinality of the test set $\mathcal{T}_t$, over which we perform accurate evaluations of our PINN between consecutive learning procedures. Even more importantly, it is also exploited as a uniform and dense pool of points where we seek the candidates that should be added to the training set.

- ***Max_ Number_ Residuals***: maximum amount of residual points that can be employed.
- ***Max_ Hidden_ Layers***: maximum amount of hidden layers for the current PINN structure.
- ***Min_ Neurons_ Per_ Layer***: minimum amount of neurons inside each hidden layer.
- ***Max_ Neurons_ Per_ Layer***: maximum amount of neurons inside each hidden layer.

The operative step of our process, which is responsible for the architectural evolution of our PINN, bases its decisions upon the value of a fundamental target variable, represented by the *target ratio* ($T.R.$) between the PDE residual evaluated on the uniformly dense test set ($C1$ for short) and over the actual pool of training points that is currently in use ($C2$ for short) (see subsection 2.2 for the adopted notation):

$$T.R.\left(\boldsymbol{W};\mathcal{T}_t,\mathcal{T}_f\right)=\frac{\mathcal{L}_f\left(\boldsymbol{W};\mathcal{T}_t\right)}{\mathcal{L}_f\left(\boldsymbol{W};\mathcal{T}_f\right)}=\frac{C1}{C2}$$

Since our networks are trained upon the latter, we generally expect this value to be greater than one. We also provide the definition of another important variable which is extensively exploited inside our algorithm, the so-called *improvement factor* ($I.F.$). Assuming n represents the index of the last learning cycle performed, the related value for the n-th stage of our algorithm is defined as the factor of improvement of the cost functional during the last training phase:

$$I.F.\left(\boldsymbol{W};\mathcal{T}_t,\mathcal{T}_b,n\right)=\frac{\mathcal{L}\left(\boldsymbol{W};\mathcal{T}_t,\mathcal{T}_b,n\right)}{\mathcal{L}\left(\boldsymbol{W};\mathcal{T}_t,\mathcal{T}_b,n-1\right)}$$

We may now continue with the exposition of the list of additional hyper-parameters and jointly explain the heuristical justifications underlying the strategies of our adaptive algorithm.

- ***NoAction_ Threshold***: if the improvement factor $I.F.$ is greater than this threshold, it means that the optimization procedure is pursuing a promising path that does not show the need of any intervention from our adaptive toolbox. Optimization proceeds with no changes.
- ***RAR_ Threshold***: if the target ratio $T.R.$ stays above this threshold, we assume our model to be significantly biased towards a better performance over the actual pool of residual points. This basically means that the PINN, under these circumstances, is thought to be presenting difficulties in generalizing the knowledge acquired over the training set to the whole domain. Under this condition for the target ratio, we increase the number of residual points according to a criterion that will be later explained.
- ***GRW_ Threshold***: if the target ratio stays below this threshold, we assume our model to be performing in one of the following regimes: either the accuracy is satisfactory over both the test set and the training set, or it is poor for all of them. It is therefore natural, when we find ourselves in this situation, to assume that we are experiencing the worst case scenario, attributing the cause of such similar performances to a poor architecture which is not able to give accurate predictions neither on the training points nor, consequently, over the test set. Thus, in this case, we operate by increasing (growing) the size of the network in a proper manner, which will be conveniently explained later.

Another important feature concerns the number of training iterations to be performed at each learning cycle. Such parameter, inside our adaptive algorithm, conveniently depends on: the number of neurons currently embedded in the PINN, the amount of hidden layers employed and the training set cardinality at present time. The higher these parameters, the larger the needed amount of learning iterations we reasonably assume our model should perform. The explicit dependency from each of the latter is not reported here: it suffices to clarify that such relationships were purely experimentally designed. Related to this subject, we eventually introduce:

- ***Force_ First_ Iterations***: represents the minimum number of training cycles that have to be mandatorily performed at the beginning of our adaptive algorithm. In Figure 38, *Forcing Left* is a decreasing counter (initialized to *Force_ First_ Iterations*) that expresses how many learning cycles still have to be unconditionally executed.
- ***Learning_ Iterations_ Multiplier***: expresses a user-defined multiplying coefficient for tuning the learning steps to be performed at each training cycle.

It must be pointed out that, by choice, our adaptive algorithm does not include any feature operating on the boundary points of the domain, that remain fixed throughout the entire optimization process. It is therefore recommended to provide them in a sufficient number when the PINN is initialized. Along with the starting structure of our network and the initial residuals embedded in the architecture, the boundary points might therefore be considered as special hyper-parameters.
Prior to the complete illustration of our adaptive algorithm, it is crucial to remark that these newly-developed PINNs present a specifically designed architectural Feed-Forward skeleton for the interconnection of their neurons. The latter consists in the presence of a special neuron on top of each hidden layer, which applies the identity function to its input (while the other neural units are embedded with the hyperbolic tangent profile).

Continuing our discussion, in the following diagrams we represent the salient steps and the most important features characterizing our innovative adaptive learning procedure.

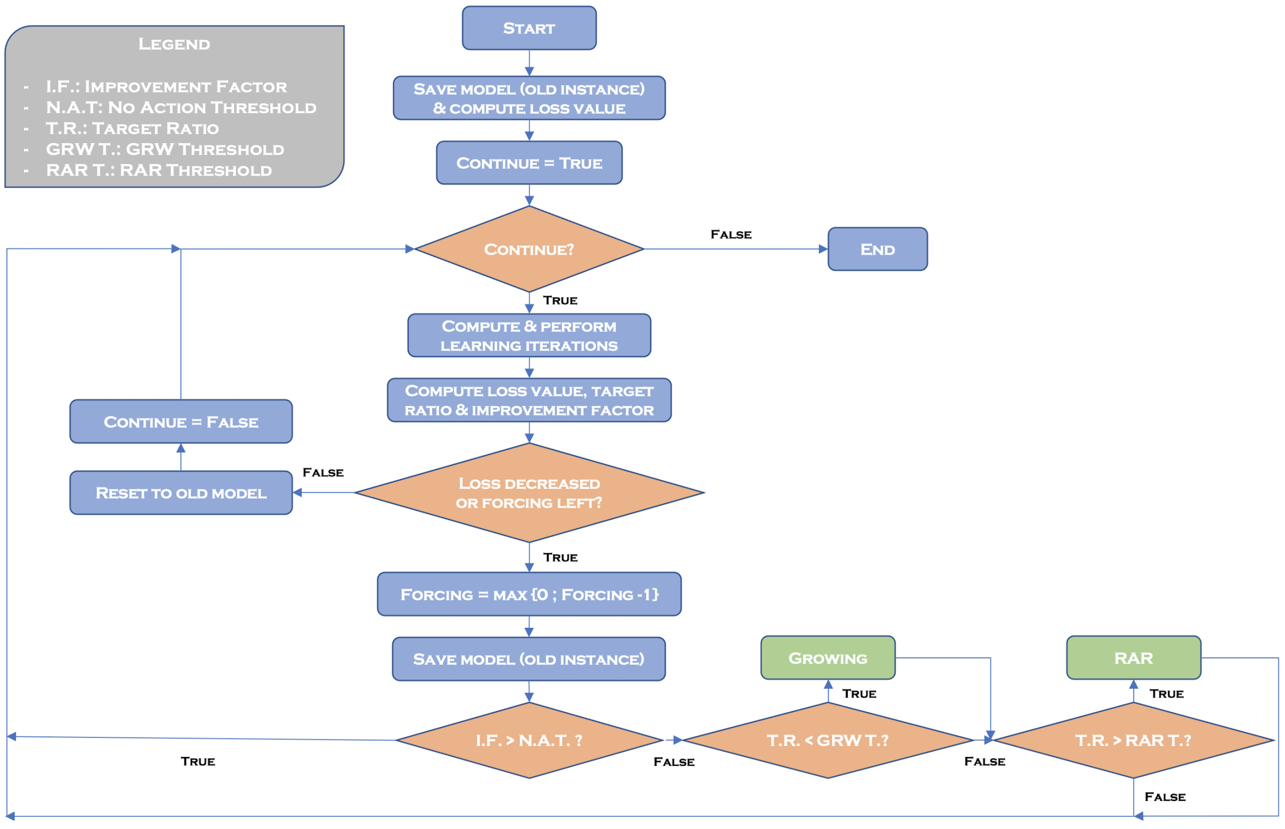

Legend
- I.F.: Improvement Factor
- N.A.T: No Action Threshold
- T.R.: Target Ratio
- GRW T.: GRW Threshold
- RAR T.: RAR Threshold
Start
Save model (old instance) & compute loss value
Continue = True
Continue?
False
End
True
Compute & perform learning iterations
Compute loss value, target ratio & improvement factor
Continue = False
Reset to old model
False
Loss decreased or forcing left?
True
Forcing = max {0 ; Forcing -1}
Save model (old instance)
Growing
RAR
True
True
I.F. > N.A.T. ?
T.R. < GRW T.?
T.R. > RAR T.?
True
False
False
False

Figure 38: Flow chart for the adaptive algorithm.

In the flow chart represented in Figure 38 we illustrate the general steps of our procedure. First of all, we save the initial instance of the model and we enter in the main loop. Secondly, we compute the number of learning iterations that have to be performed by means of the experimental expression that we previously mentioned, which takes into account the total number of neurons, the amount of hidden layers and the training set cardinality that characterize the current version of the PINN. Then, after the completion of a new learning cycle, we compute the updated values of the loss function, alongside the aforementioned target ratio ($T.R.$) and improvement factor ($I.F.$). Notice that the expression of the cost functional involves an evaluation of the PDE residuals over the test set, instead of the training set. At this point, if the general performance has improved (or if we are still enforcing a possibly positive number of training cycles), we proceed by updating the number of steps that are yet to be forced, subsequently overwriting the saved model with the current version of the PINN. Now, if the improvement factor is larger than the *NoAction_ Threshold*, we turn back to the beginning of the loop without taking any further action. Otherwise, if the related entering conditions are satisfied, the Growing Method and the RAR technique come into play and accordingly modify the PINN structure. Eventually we go back to the beginning of the cycle, and start it over until we reach the end of the algorithm. Our stopping criterion is met whenever the minimum number of learning cycles have been performed and the computation of the loss function provides a worse value than the previously saved model. At that point, before exiting the adaptive procedure, we definitively reset our PINN architecture to the last (better) version stored in memory.

In Figures 40 and 41 we visualize the specific diagrams for the inner functioning concerning the Growing Method and our version of the Residual Adaptive Refinement technique. Starting from the former, as soon as we enter the algorithm it is immediately checked whether there is enough space left in the last hidden layer for the addition of new neurons. If this is the case, we double their amount (in the specified layer) and we randomly initialize the correspondent weights by means of a zero-mean probability distribution. Otherwise, if the *Max_ Neurons_ Per_ Layer* threshold has been reached, we try to increment the number of hidden layers for our network: if this is possible, namely if the latter has not reached its limit value (*Max_ Hidden_ Layers*), we insert a new layer embedded with *Min_ Neurons_ Per_ Layer* units. In Figure 39 we illustrate this mechanism.

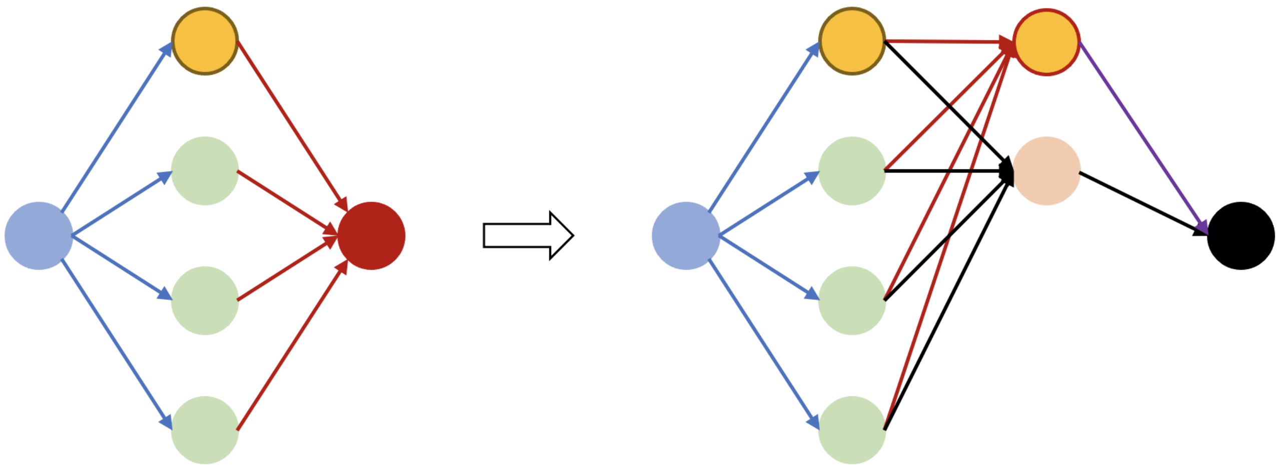
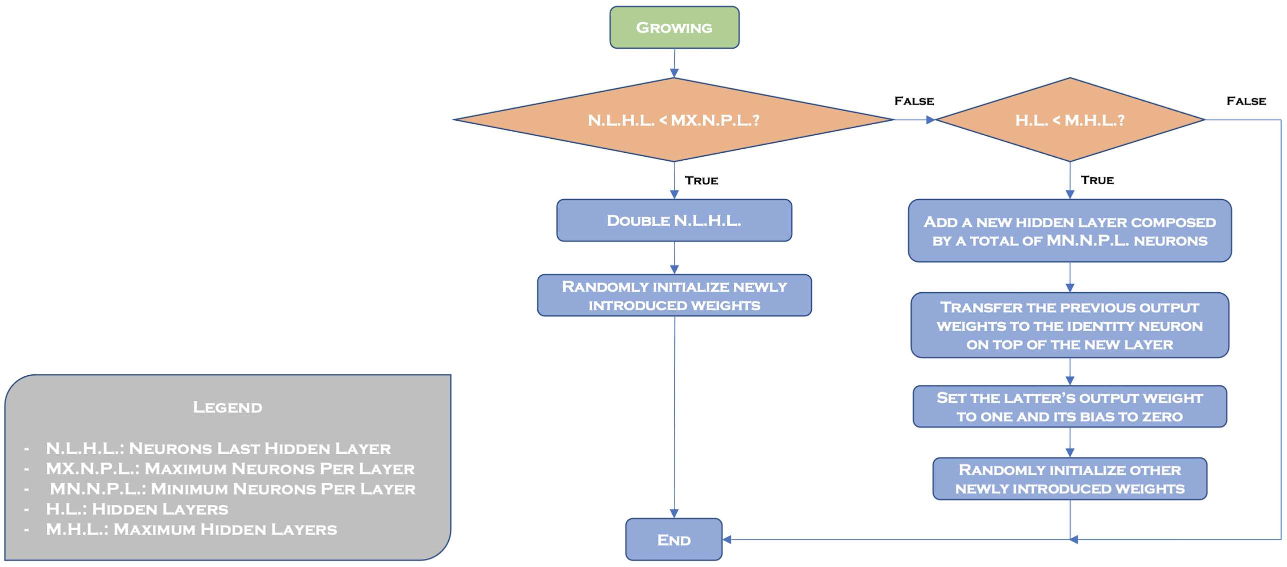


Figure 39: Addition of a new hidden layer for the Growing procedure.

First of all we remark that the first neuron of every hidden layer (depicted in yellow), in these architectures, is embedded with the identity activation function: this implies that, when we add a new layer at the end of our structure, simply by feeding the previous output weights (the arrows in red) to the mentioned newly-created neuron (yellow with red borders) and by initializing its relative output connection to one (purple arrow) and its bias to zero, and setting all the other new connections (black arrows) with a random zero-mean distribution and their biases to zero, we are able to retain (on average) the complete information coming from the previous version of the network. It is crucial to notice that such a property is exclusively made possible by the use of an identity activation for the mentioned special neurons. Such a procedure can be generalized to a generic n-dimensional output DNN employing n neurons with these characteristics on top of every hidden layer. To sum up, following the two described procedures to enlarge our network (neuron-wise or layer-wise), the expected (in a statistical sense) output of our model remains unchanged. It should be needless to say that, if our architecture has already grown to its maximum size, the algorithm immediately returns without performing any further action. Concerning the application of the Growing Method, we still need to clarify an important concept that was willingly neglected in the previous pages. As we anticipated, whenever the target ratio $T.R.$ stays below the user-defined *GRW_ Threshold* it is assumed that the related PINN, that consequently performs similarly over the training and test sets, presents a poor general accuracy. Our claim is that the mentioned assumption does not compromise the possible success attached to the adaptive algorithm. In fact, even if we wrongly suppose our PINN to express bad performances, the structure of our procedure prevents the model to take a misleading direction: in case the performance worsens, the algorithm actually stops during the successive training cycle, retrieving the last (better) network that has been stored in memory (see Figure 38).



Growing
N.L.H.L. < MX.N.P.L.?
False
H.L. < M.H.L.?
False
True
True
Double N.L.H.L.
Randomly initialize newly introduced weights
Add a new hidden layer composed by a total of MN.N.P.L. neurons
Transfer the previous output weights to the identity neuron on top of the new layer
Set the latter's output weight to one and its bias to zero
Randomly initialize other newly introduced weights
End
Legend
- N.L.H.L.: Neurons Last Hidden Layer
- MX.N.P.L.: Maximum Neurons Per Layer
- MN.N.P.L.: Minimum Neurons Per Layer
- H.L.: Hidden Layers
- M.H.L.: Maximum Hidden Layers

Figure 40: Flow chart for the Growing Method.

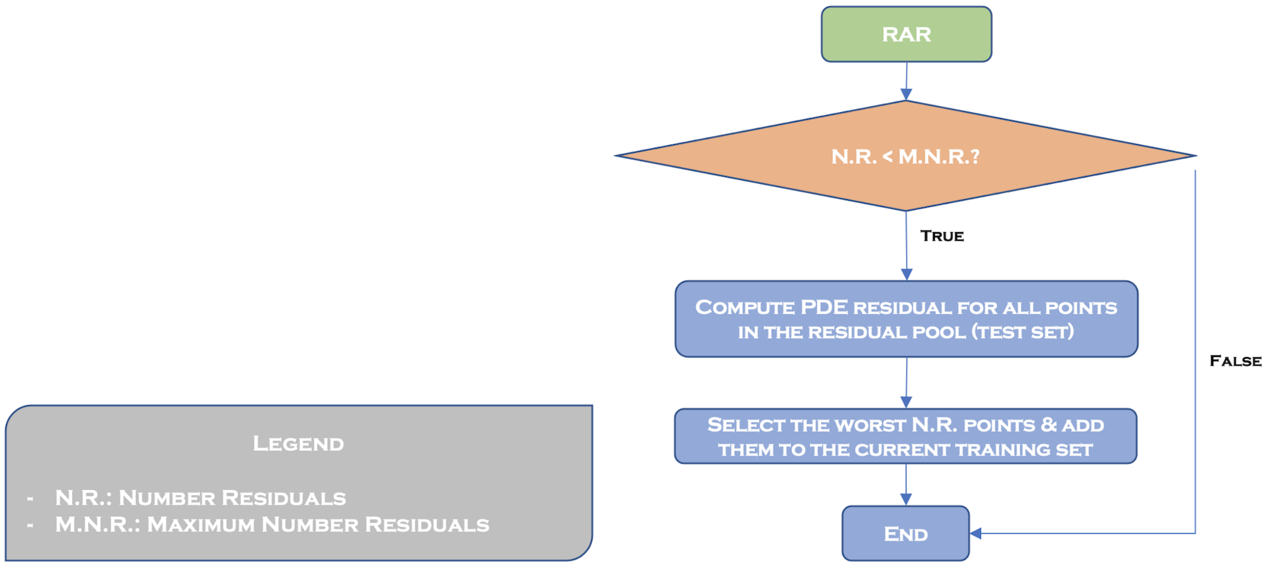

RAR
N.R. < M.N.R.?
True
False
Compute PDE residual for all points in the residual pool (test set)
Select the worst N.R. points & add them to the current training set
End
Legend
- N.R.: Number Residuals
- M.N.R.: Maximum Number Residuals

Figure 41: Flow chart for the RAR technique variation used in our work.

In Figure 41 we represent the simple flow chart concerning the application of our revisited version of the so-called Residual Adaptive Refinement technique, introduced for the first time in [37]. The unique check present in this algorithm controls whether there is the possibility to further increase the number of residual points embedded in the current architecture. If so, the PDE residual is evaluated over the entire test set (alternatively called residual pool) and the points that present the worst results are aggregated to the current training set. Assuming N to be the number of residual points prior to the application of this technique, at the end of the algorithm our network will possess $2N$ of them (we actually add to the training set those N points belonging to the residuals pool which present the highest PDE residual values). Notice that, since the test set remains unmodified during the whole optimization phase, there are no restrictions regarding the location of the points that can be added: some spots might actually be repeatedly inserted in the training set (in different applications of RAR). Such eventuality does not represent an issue: if this happened, it would simply mean that the repeated points have assumed a relatively higher importance with respect to the others, given by the fact that in the mentioned spot the PINN struggles in providing good performances.

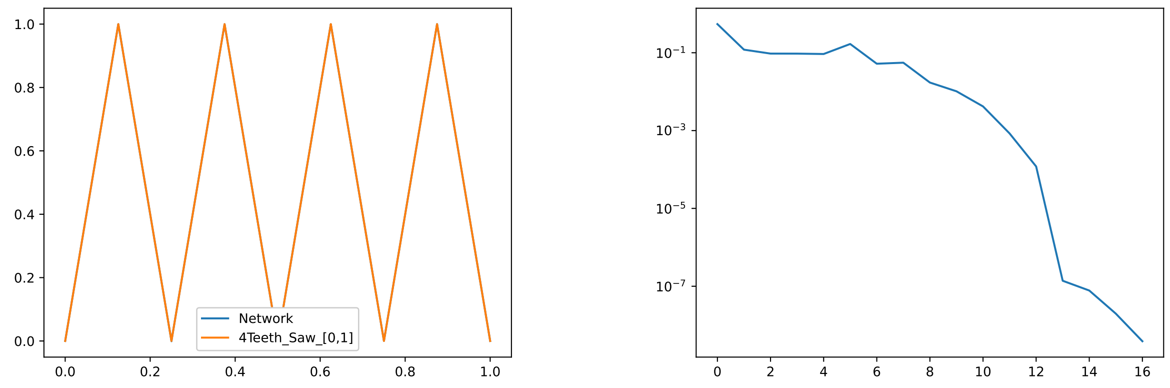
Before passing to the analysis of the results related to our experimental tests for the described framework, we shall conclude the current discussion with a series of useful considerations. First of all, let us provide a comment regarding the chosen features of our Growing Method. The operative decision that consists in prioritizing the growth for the number of embedded neurons in the last hidden layer (before increasing the number of layers) has been made because slimmer networks are generally much computationally cheaper than deeper architectures. Therefore, we accordingly tried to obtain satisfactory performances with fewer hidden layers before accepting to increase the number of the latter. Notice that, by construction, our final versions of the networks do not necessarily have the same amount of neurons in each layer, providing a more elastic and wider range of possibilities for our models. Our second comment concerns the fact that, operatively, the hyper-parameters *GRW_ Threshold* and *RAR_ Threshold* are themselves adapted during the execution of the algorithm, changing in a way that makes it more and more difficult to fall in the relative procedural branch every time the latter is entered. This has been done in order to avoid the occurence of an extremely and unnecessarily heavy learning procedure whenever these two thresholds are badly tuned for the problem at hand. We finally remark that many of the design choices that have been made for the construction of this new adaptive algorithm might be clearly revisited for possible further developments, that may also depend on the application that it is intended to be addressed. Despite the fact that we certainly created an elastic, generic and portable technique, however still belonging to its first experimental phase, we must acknowledge that, unfortunately, we have not been able to eliminate the necessity of many hyper-parameters that need to be tuned by the user.

4.2. Results

The contents of the following adaptive tests take inspiration from a restricted subset of cases formerly analyzed in the basic PINN sensitivity study. A total of five experiments were performed in this framework. For the sake of simplicity, the related learning settings will not be entirely reported in our discussion.

Test 1

In the first test case we consider the pure approximation problem (2) for the 4-saw-teeth wave, which was the subject of our study in the first test of the basic PINN generic sensitivity analysis. In order to attenuate the effects linked to random initialization, we trained three independent models with the same initial specifications, all embedded with the ReLU activation function.
All these adaptive networks have noticeably explored a multi-layered architecture. In two cases (out of three) we have even obtained the exact same structure, composed by 80 neurons in the first hidden layer and 10 of them in the second. It is worth specifying, at this point, that we set the *Min_ Neurons_ Per_ Layer* value to 10 and, accordingly, the *Max_ Neurons_ Per_ Layer* hyper-parameter to 80 for this test. The obtained relative L^2 errors are all satisfactory and two of them are particularly worthy, with a value of about 0.1% in one case and 0.01% in the other. The least performing model got evidently stuck in a slightly worse local minimum of the loss function, presenting a final value for the error that is close to 3%. Figure 42 shows the basically overlapping plots for the exact solution and the best model that we achieved (on the left), alongside the corresponding loss function evolution (on the right). Here we recall that, for all these tests, the lastly mentioned quantity is evaluated by exploiting the dense residual pool (other than the usual fixed boundary points) at the end of every adaptive cycle. For the illustrated model, a total of 16 learning periods were actually performed.
Summing everything up, we have obtained satisfactory results in this very first experimental tranche of adaptive models. Indeed, even comparing the best networks observed in the counterpart trial performed during the basic PINN sensitivity analysis (see Test 1 of sub-subsection 3.2.3) with the best architecture constructed here, we appreciate a better accuracy in the latter. For completeness, it must be noticed that the final number of residuals for our best network amounts to 2560 (starting from 10), which turns out to be a much greater number than the training set cardinalities used for the basic PINN counterparts. Notice that a minimum of 10 initial learning cycles were forced for this experiment.

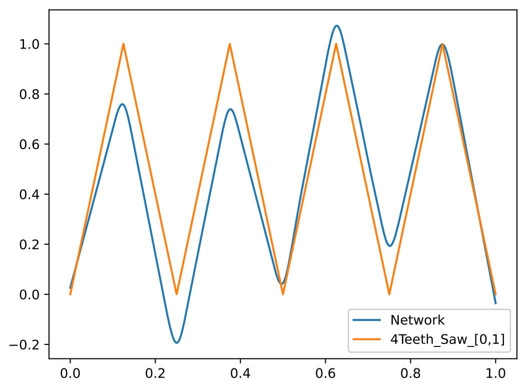


(a) Target solution and model approximation. (b) Loss function evolution.

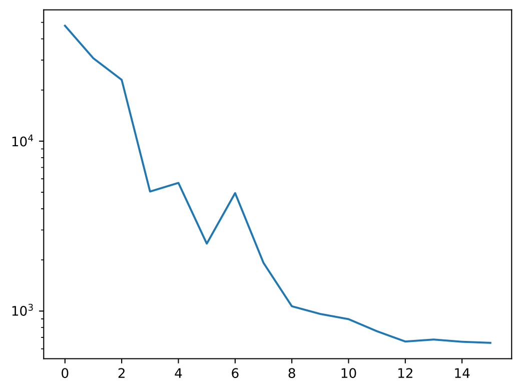
Figure 42: Best approximation plot of the target solution by the second trial adaptive PINN (left) and its related cost functional evolution (right) in semi-logarithmic scale.

Test 2

In this test case we look, through the implicit solution of the one-dimensional Poisson equation with homogeneous boundary conditions (1), for an approximation of the 4-saw-teeth target function, in the identical fashion already explored in Test 3 for the basic PINN generic sensitivity analysis (see sub-subsection 3.2.3). In order to attenuate the effects linked to random initialization, we trained three independent models with the same initial specifications, all embedded with the hyperbolic tangent activation function. The salient features for the starting configuration concern: the number of initial (uniformly distributed) residual points set to 80, an initial structure composed by one hidden layer of 10 neurons (that is also the value given to the hyper-parameter *Min_ Neurons_ Per_ Layer*), a maximum number of residuals set to 10240 and the variable *Max_ Neurons_ Per_ Layer* tuned to 80. A total of 15 learning cycles were forced for all the networks, with the *Learning_ Iterations_ Multiplier* augmented to 1.5.



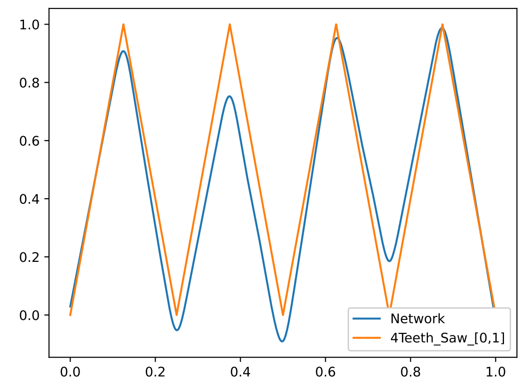
(a) Target solution and model approximation.



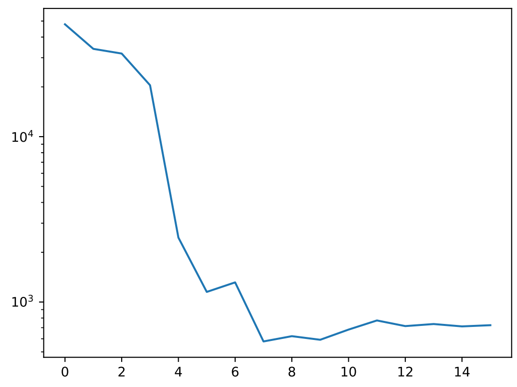
(b) Loss function evolution.

Figure 43: Best approximation plot of the target solution by the second trial adaptive PINN (left) and its related cost functional evolution (right) in semi-logarithmic scale.

Figure 43 shows the graphical view of the best instance alongside the related cost functional evolution. As we can notice from the left plot, even the most performing architecture has not been able to reach a high degree of accuracy for its approximation. Nevertheless, we must highlight that our network has at least grasped the essential features of the exact solution. Interestingly, given such starting configuration, all the models have actually evolved towards an identical final overall structure with respect to the number of hidden layers (1), the amount of neurons (80) and the training set cardinality (5120). Under the mentioned circumstances, these final architectures remained shallow and embedded with the maximum number of neurons that we formerly imposed. Evidently, once they reached this structural configuration, either they did not satisfy the explained conditions to perform a further growing step or they managed to enter the Growing Method obtaining worse results. For this reason, we made a further attempt that consisted in training three additional models by leaving every setting equal to the described case, except for the *Max_ Neurons_ Per_ Layer* parameter, decreased to 40.



(a) Target solution and model approximation.



(b) Loss function evolution.

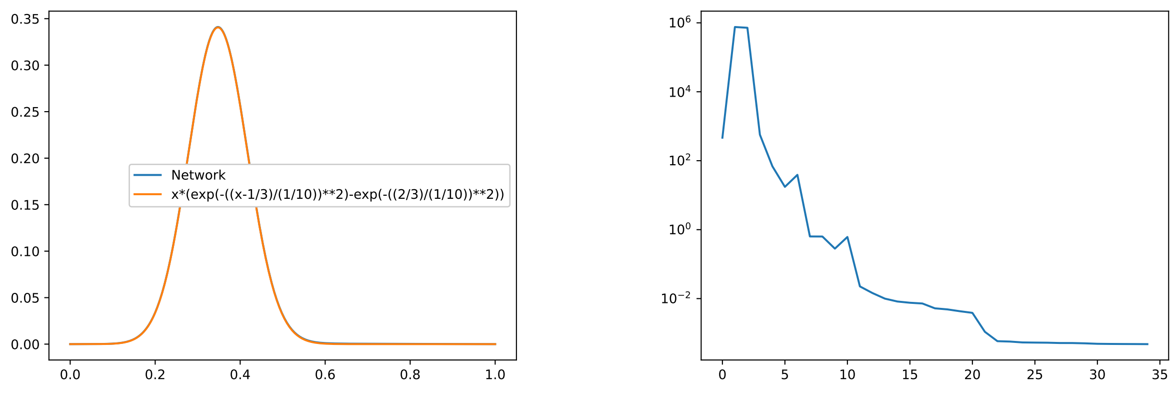
Figure 44: Best approximation plot of the target solution by the second additional trial adaptive PINN (left) and its related cost functional evolution (right) in semi-logarithmic scale.

Figure 44 represents the best instance among the networks trained for our second attempt of the current test case. Despite a visibly improved performance, we have not been able to obtain a satisfactory level of accuracy yet. Noticeably, also here all the models converged to the same final structure, constituted by 40 neurons in the first hidden layer and 10 of them in the second. Looking at the plot concerning the evolution of the loss function, we can appreciate that its minimum was reached after seven iterations: recall that the algorithm, however, could not stop at that point because we forced it to perform at least 15 learning cycles for this problem. In light of these considerations, we prompted a third trial for this test, in order to investigate the final performances when the networks are forced to execute a lower number of learning cycles. Here, *Force_ First_ Iterations* is set to 5.

Given also these last results, we decided to spare the visual representation of the plots related to the third tranche of models. This phenomenon seems difficult to be fully interpreted by our adaptive optimization procedure. The latter, nonetheless, proves to be able in providing a consistent approximation framework, since the models typically converge towards a common architectural structure and many target features are often learned successfully. We therefore conclude the discussion for the current test case by pointing out a new possible applicative path for the mentioned algorithm, that consists in using the latter to identify a proper structural guess for the architecture that should be exploited to obtain a reliable approximation of the target solution. Nevertheless, we must acknowledge that performing a limited number of attempts with our adaptive algorithm (as we did in this case) does not always guarantee a successful level of accuracy. The experimental proof of this statement has just been provided by the results of this trial.

Test 3

For this experiment we select the bell-like target solution, already considered in Test 5 of our basic PINN generic sensitivity analysis (in sub-subsection 3.2.3), imposing it implicitly by means of the one-dimensional Poisson equation with homogeneous boundary conditions (1). As usual, in order to attenuate the effects linked to random initialization, we trained three independent models with the same initial specifications, all embedded with the hyperbolic tangent activation function. Our structures start with 10 uniformly distributed residual points and one hidden layer composed by 10 neurons (that is also the value attached to $Min\_Neurons\_Per\_Layer$). The hyper-parameter $Max\_Neurons\_Per\_Layer$ is set to 80, while a total number of 10 learning cycles are forced. The performances are all very similar, with a relative L^2 error that hovers around 0.3%-0.4%. One more time, all three models converged to the same shallow structure formed by 20 neurons. In particular, two of them also ended up with the same cardinality for their training sets. As we can also appreciate from Figure 45, the best trained network matches the exact solution very reliably. Nevertheless, the PINNs belonging to the basic counterpart of this test reached better results (also the ones embedded with the same structure), even with a much smaller number of residual points. It is in light of all these comments and considerations that we can attribute to the current experiment only a partial degree of success.



(a) Target solution and model approximation.

(b) Loss function evolution.

Figure 45: Best approximation plot of the target solution by the second trial adaptive PINN (left) and its related cost functional evolution (right) in semi-logarithmic scale.

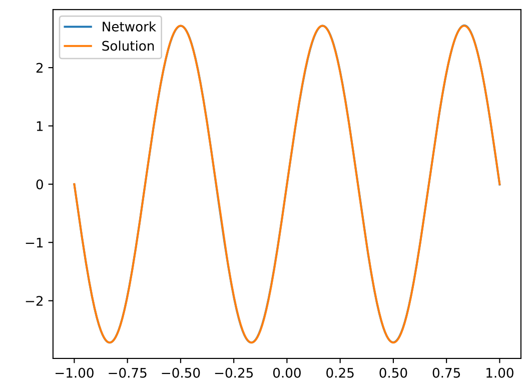
Test 4

This experiment takes inspiration from Test 7 of our basic PINN generic sensitivity study (in sub-subsection 3.2.3), where we imposed a sinusoidal target with exponentially decaying magnitude for the Heat problem (6):

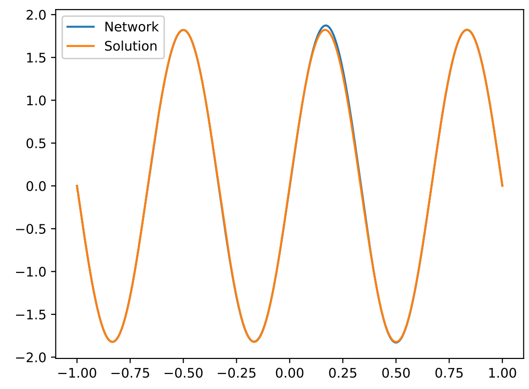
$$g(x,t) = \exp(-t)\sin(3\pi x)$$

Also in this case we attenuated the effects of random initialization by training three independent models with the same specifications, all embedded with the hyperbolic tangent activation function. Our networks start with 40 uniformly distributed residual points and one hidden layer composed by 10 neurons (which is also the value attached to $Min\_Neurons\_Per\_Layer$). The hyper-parameters $Max\_Neurons\_Per\_Layer$ and $Max\_Number\_Residuals$ are respectively set to 80 and 5120, while a total of 10 learning cycles are forced. The performances shown by all models are very similar, with a relative L^2 error that hovers around 2%-3.5%.

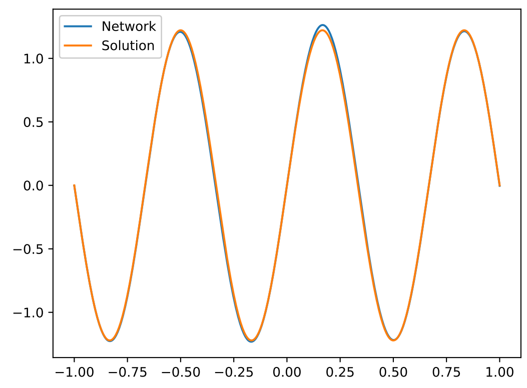
Once again all models converged to the same shallow structure, here formed by 40 neurons. Rather uniquely, they all present the same number of residuals (2560) as well. Although we have obtained satisfactory results for this trial, we can still observe better performances from the networks produced during the counterpart test executed for the basic PINN sensitivity analysis, where we even employed a lower number of training spots. Figure 46 shows the illustration of a six time-frame evolution of the best network's graph next to the sought solution. As we can see, the represented curves basically overlap for small time values (close to the initial condition) while they progressively detach as time goes by, especially close to the second peak of the target.



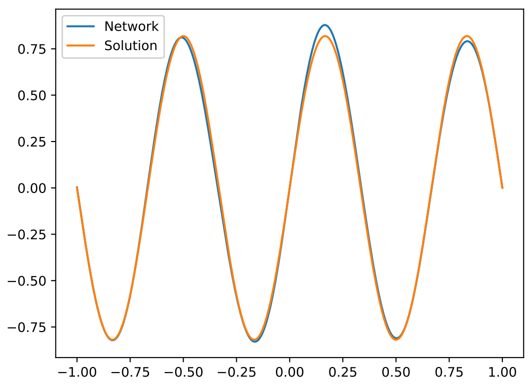
(a) Target solution and network approximation, $t = 0$.



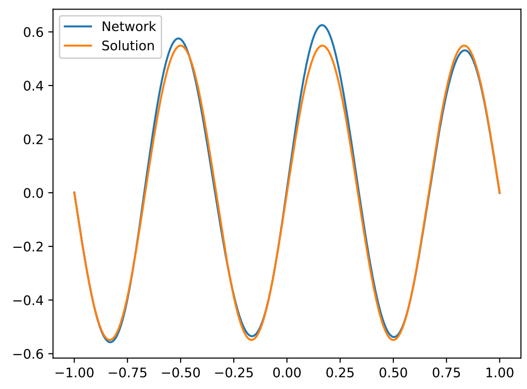
(b) Target solution and network approximation, $t = 0.2$.



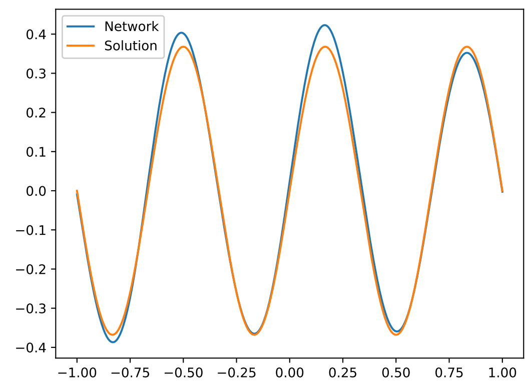
(c) Target solution and network approximation, $t = 0.4$.



(d) Target solution and network approximation, $t = 0.6$.



(e) Target solution and network approximation, $t = 0.8$.

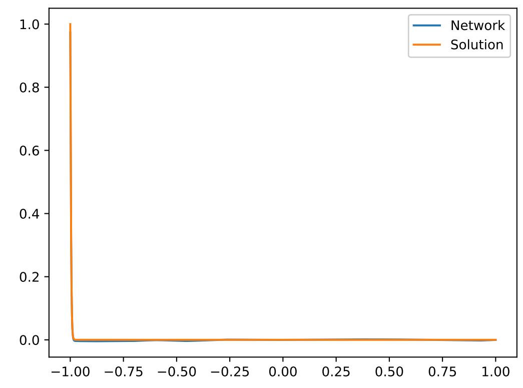


(f) Target solution and network approximation, $t = 1$.

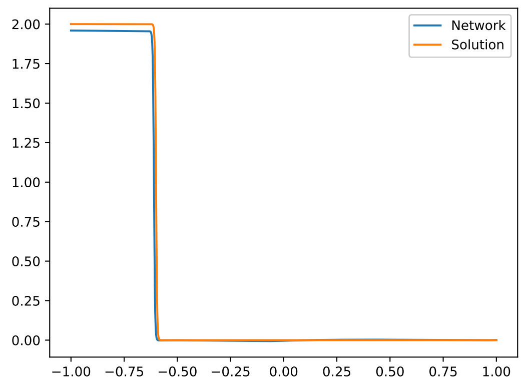
Figure 46: Best approximation plots, at six uniformly-spaced time steps, of the target solution by the first network (trained with the hyperbolic tangent activation function).

Test 5

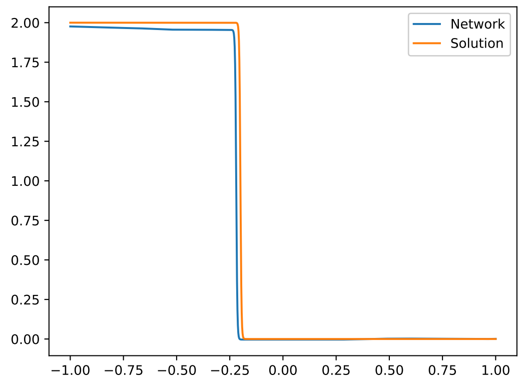
Our last experiment consists in re-analyzing, using the newly-developed adaptive learning scheme, the same solution profile seen in Test 6 for the basic PINN generic sensitivity analysis (in sub-subsection 3.2.3) through the Burger's equation (5). We recall that one of the target's main features is represented by the sharp (but still indefinitely regular) interface that emerges over the spacetime line $x = t$ in the domain $[0, 1]^2$. As always, we attenuated the effects of random initialization by training three independent models with the same specifications, all embedded with the hyperbolic tangent activation function. All networks were prompted with 40 uniformly distributed residual points and one hidden layer composed by 10 neurons (which is also the value set for *Min_ Neurons_ Per_ Layer*). The hyper-parameters *Max_ Neurons_ Per_ Layer* and *Max_ Number_ Residuals* are respectively set to 160 and 5120, while a total of 10 learning cycles are forced.
Two important considerations are in order: concerning the performances shown by the models we definitely cannot consider ourselves fulfilled, since the best observed relative L^2 error stays above 20%. Secondly, all networks converged to different shallow structures. The one that presents the best result is, curiously, also the smallest among all. At this point, we can usefully recall that the best model obtained during the analysis of the counterpart basic trial corresponds to a structure composed by two hidden layers embedded with a few neurons each. In order to dig deeper into the problem, also by considering our last observations, we decided to prompt another tranche of networks: these were initialized identically to the former except for the hyper-parameter *Max_ Neurons_ Per_ Layer*, which was halved to 80. The results obtained with these additional attempts slightly improve the scenario, that however remains not ideal, with a best relative L^2 error close to 14%. It is worth noticing that our modification of the initial configurations did not produce any appreciable result, because no instances actually explored a multi-layered architecture successfully. Anyway, it must be said that the imposed solution represents a very stiff target, and that with a totality of just six attempts we can appreciate at least a network that, graphically, went not that far from a satisfactory approximation.



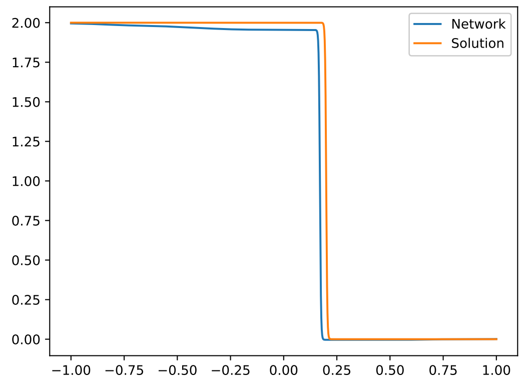
(a) Target solution and network approximation, $t = 0$.



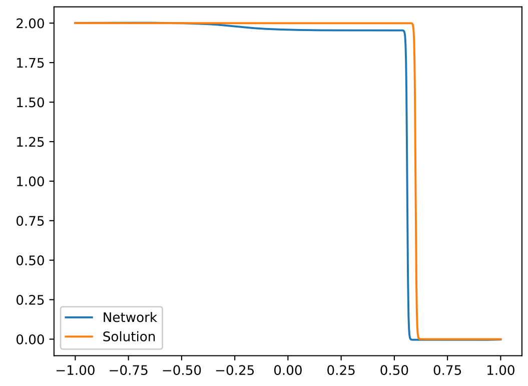
(b) Target solution and network approximation, $t = 0.2$.



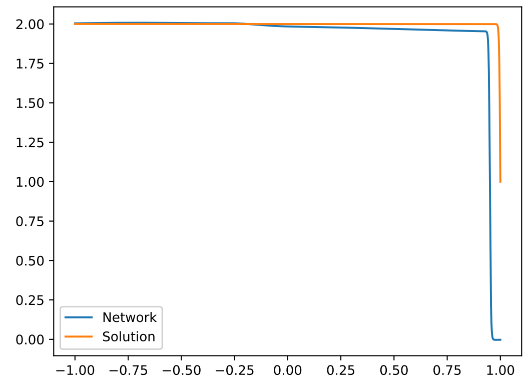
(c) Target solution and network approximation, $t = 0.4$.



(d) Target solution and network approximation, $t = 0.6$.



(e) Target solution and network approximation, $t = 0.8$.



(f) Target solution and network approximation, $t = 1$.

Figure 47: Best approximation plots, at six uniformly-spaced time steps, of the target solution by the first additional network (trained with the hyperbolic tangent activation function).

In Figure 47 above, we report a graphical view for the comparison between the best model approximation obtained in this experiment and the related target solution. Similarly to what we were able to observe for the previous test case, also here we appreciate a progressive detachment of the network curve from the objective function. Close to the initial condition (obviously imposed in a Dirichlet fashion) the trends basically overlap with each other, but they move away from each other as time increases.
The heaviest architecture (in terms of computational cost) trained for this experiment (which turns out to be the best in terms of performance) took more than two hours to complete its learning phase while, on average, the six models produced here have run for about one hour.

5. Conclusions

In this work we have explored the performances of PINNs over a set of predefined test cases with the aim of assessing their sensitivity (measured through the relative L^2 error) with respect to, on one hand, the complexity of the target solutions underlying the studied problems and, on the other, the choices concerning the architectural properties of our networks. First of all, we saw that it is not necessarily true that deeper models perform better than shallow networks. Secondly, we highlighted that some of the most common trends (which, nevertheless, are not valid in general presumably due to effects linked to the so-called generalization error) consist in the improvement of performances when we increase the number of neurons per layer or the amount of residual points inside the computational domain. We remark that, in some cases, no clear patterns emerged from the gathered outcomes of our experiments (as for Test 6 in sub-subsection 3.2.3). Along the way we also discovered that indefinitely regular activation functions prove to be much more reliable than irregular profiles such as ReLU, conjecturing and verifying a possible explanation for the failure associated to the latter (see Test 5 in sub-subsection 3.2.3). Moreover, increasing the maximum order of derivative appearing in the PDE seemingly leads to stiffer and more unstable learning procedures. Overall we can acknowledge success for the presented tool, because for all the trials mentioned up to now we have always been able to find successful instances that provided satisfactory levels of accuracy.
The second part of this project has been dedicated to the introduction and testing of an adaptive version of the PINN, whose aim, other than seeking for a reliable approximation of the solution of the differential problem at hand, consists in searching for the "optimal" architecture for the interpretation of the mentioned target function. In order to do so we employed a revisited version of the Residual Adaptive Refinement technique (introduced in [37]) alongside the Growing Method, all applied to a simple but innovative modification of the usual Feed Forward Neural Network structure. At the very end of this work, we made the first experimental tests of this adaptive algorithm over a pool of test cases already encountered in the previous analyses. Some of them turned out to be completely successful, while other still show clear signs of improvements to be made. With that being said we must ascertain that, after all, thanks to the smaller computational cost attached to our new scheme, we are left with promising margins for potential improvements. In order to avoid possible misunderstandings, we must recognize that even in this preliminary version of the algorithm, whenever we did not achieve completely satisfactory performances, our scheme has nonetheless shown to be partially useful in understanding a convenient structure that could be embedded in a PINN to achieve appreciable results.

Still on this subject, the path drawn by our basic PINN analyses should motivate the need of an alternative road for the resolution of differential problems, if we intend to pursue the usage for this category of techniques. The latter (basic) framework, in fact, generally requires a potentially enormous amount of simulations to be able to reach, if this is even possible, a good model for the description of the phenomenon under study. Indeed, every time a new architectural parameter is considered, the number of trials that must be performed to explore all the possible settings, by varying singularly each parameter in its own range, grows exponentially fast. As an immediate consequence, this framework cannot be considered scalable at all.

5.1. Further Developments

Possible extensions of the present work are:

- Exploit the so-called Pruning techniques (thoroughly presented in Appendix B), for which the main utilities are already included in our library, for the adaptive PINN strategy. This proposal should follow the dual heuristical reasoning that has been pursued in the present work, starting from a very large architecture (that should exhibit overfitting) and progressively cutting the redundant connections of the network (according to a proper criterion that must be chosen) until an optimal performance is reached.
- Modify or enhance our adaptive algorithm with new features such as the adaptive activation functions (see [24]) or the introduction of adaptive weight multiplying the loss function's single terms (as in [39]).
- Exploit PINNs for the resolution of inverse problems or, as shown in [37], to solve integro-differential equations with the features that have been exposed here.

References

[1] Mark Ainsworth and Justin Dong. Galerkin Neural Networks: A Framework for Approximating Variational Equations with Error Control. *CoRR*, 2021.

[2] Mark Ainsworth and Yeonjong Shin. Plateau Phenomenon in Gradient Descent Training of ReLU networks: Explanation, Quantification and Avoidance. *CoRR*, 2020.

[3] Quarteroni Alfio. *Numerical models for differential problems.* Springer, 2020.

[4] Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. Automatic differentiation in machine learning: a survey. *CoRR*, 2015.

[5] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization Methods for Large-Scale Machine Learning, 2018.

[6] Zhiqiang Cai, Jingshuang Chen, and Min Liu. Self-adaptive deep neural network: Numerical approximation to functions and PDEs, 2021.

[7] Giovanna Castellano, Anna Fanelli, and Marcello Pelillo. An iterative pruning algorithm for feedforward neural networks. *IEEE transactions on neural networks*, 1997.

[8] Eric C. Cyr, Mamikon A. Gulian, Ravi G. Patel, Mauro Perego, and Nathaniel A. Trask. Robust Training and Initialization of Deep Neural Networks: An Adaptive Basis Viewpoint. *CoRR*, 2019.

[9] Wolfgang Dahmen, Min Wang, and Zhu Wang. Nonlinear Reduced DNN Models for State Estimation, 2021.

[10] Chenguang Duan, Yuling Jiao, Yanming Lai, Xiliang Lu, and Zhijian Yang. Convergence Rate Analysis for Deep Ritz Method, 2021.

[11] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning. *CoRR*, 2017.

[12] Scott Fahlman and Christian Lebiere. The Cascade-Correlation Learning Architecture. In D. Touretzky, editor, *Advances in Neural Information Processing Systems.* Morgan-Kaufmann, 1990.

[13] Luca Formaggia. *Applicazioni ed esercizi di modellistica numerica per problemi differenziali.* Springer, 2005.

[14] Marcus Frean. The Upstart Algorithm: A Method for Constructing and Training Feedforward Neural Networks. *Neural Computation - NECO*, 1990.

[15] Steve Gallant. Perceptron-based learning algorithms. *IEEE transactions on neural networks*, 1990.

[16] Yue-Seng Goh and Eng-Chong Tan. Pruning neural networks during training by backpropagation. In *Proceedings of IEEE Annual International Conference on: 'Frontiers of Computer Technology'*, 1994.

[17] Ingo Gühring, Mones Raslan, and Gitta Kutyniok. Expressivity of Deep Neural Networks. *CoRR*, 2020.

[18] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both Weights and Connections for Efficient Neural Networks. *CoRR*, 2015.

[19] Harun-Or-Roshid, M. Zulfikar Ali, and Md. Rafiqul Islam. Explicit and Exact Traveling Wave Solutions of Cahn Allen equation using MSE Method, 2016.

[20] B. Hassibi, D.G. Stork, and G.J. Wolff. Optimal Brain Surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, 1993.

[21] Juncai He. Relu Deep Neural Networks and Linear Finite Elements. *Journal of Computational Mathematics*, 2020.

[22] Qingguo Hong, Jonathan Siegel, and Jinchao Xu. Rademacher Complexity and Numerical Quadrature Analysis of Stable Neural Networks with Applications to Numerical PDEs, 2021.

[23] Guang-Bin Huang, Lei Chen, and Chee Siew. Universal Approximation Using Incremental Constructive Feedforward Networks With Random Hidden Nodes. *IEEE transactions on neural networks*, 2006.

[24] Ameya D. Jagtap, Kenji Kawaguchi, and George Em Karniadakis. Adaptive activation functions accelerate convergence in deep and physics-informed neural networks. *Journal of Computational Physics*, 2020.

[25] P.P. Kanjilal and D.N. Banerjee. On the application of orthogonal transformation for the design and analysis of feedforward networks. *IEEE Transactions on Neural Networks*, 1995.

[26] E.D. Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks*, 1990.

[27] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2017.

[28] R. Kozma, M. Sakuma, Y. Yokoyama, and M. Kitamura. On the accuracy of mapping by neural networks trained by backpropagation with forgetting. *Neurocomputing*, 1996.

[29] Tin-Yan Kwok and Dit-Yan Yeung. Objective Functions for Training Units in Constructive Neural New Hidden Networks. *Neural Networks, IEEE Transactions on*, 1997.

[30] Yann Lecun, John Denker, and Sara Solla. Optimal Brain Damage. 1989.

[31] M. Lehtokangas. Modeling with Constructive Backpropagation. *Neural Networks*, 1999.

[32] Asriel U. Levin, Todd K. Leen, and John E. Moody. Fast Pruning Using Principal Components. 1993.

[33] Huimin Li, Marina Krcek, and Guilherme Perin. A Comparison of Weight Initializers in Deep Learning-based Side-channel Analysis. 2020.

[34] Derong Liu, Tsu-Shuan Chang, and Yi Zhang. A constructive algorithm for feedforward neural networks with incremental training. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions*, 2003.

[35] Min Liu and Zhiqiang Cai. Adaptive Two-Layer ReLU Neural Network: II. Ritz Approximation to Elliptic PDEs, 2021.

[36] Min Liu, Zhiqiang Cai, and Jingshuang Chen. Adaptive Two-Layer ReLU Neural Network: I. Best Least-squares Approximation, 2022.

[37] Lu Lu, Xuhui Meng, Zhiping Mao, and George Karniadakis. DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, 2021.

[38] Stefano Markidis. Physics-Informed Deep-Learning for Scientific Computing. 2021.

[39] Levi D. McClenny and Ulisses M. Braga-Neto. Self-Adaptive Physics-Informed Neural Networks using a Soft Attention Mechanism. *CoRR*, 2020.

[40] Marc Mezard and Jean-Pierre Nadal. Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A: Mathematical and General*, 1999.

[41] Hrushikesh N. Mhaskar and Tomaso A. Poggio. Deep versus shallow networks : An approximation theory perspective. *CoRR*, 2016.

[42] Piotr Minakowski and Thomas Richter. Error Estimates for Neural Network Solutions of Partial Differential Equations, 2021.

[43] Siddhartha Mishra and Roberto Molinaro. Estimates on the generalization error of Physics Informed Neural Networks (PINNs) for approximating PDEs, 2021.

[44] Sidharth Mishra, Uttam Sarkar, Subhash Taraphder, Sanjoy Datta, Devi Swain, Reshma Saikhom, Sasmita Panda, and Menalsh Laishram. Principal Component Analysis. *International Journal of Livestock Research*, 2017.

[45] Soha Abd El-Moamen Mohamed, Marghany Hassan Mohamed, and Mohammed F. Farghally. A New Cascade-Correlation Growing Deep Learning Neural Network Algorithm. *Algorithms*, 2021.

[46] J.O. Moody and P.J. Antsaklis. The dependence identification neural network construction algorithm. *IEEE Transactions on Neural Networks*, 1996.

[47] M. Mozer and Paul Smolensky. Using Relevance to Reduce Network Size Automatically. *Connection Science*, 1989.

[48] Maryam Najafabadi, Taghi Khoshgoftaar, Flavio Villanustre, and John Holt. Large-scale distributed L-BFGS. *Journal of Big Data*, 2017.

[49] Christian Petersen. Neural Network Theory. 2020.

[50] Tomaso A. Poggio, Andrzej Banburski, and Qianli Liao. Theoretical Issues in Deep Networks: Approximation, Optimization and Generalization. *CoRR*, 2019.

[51] P.V.S. Ponnapalli, K.C. Ho, and M. Thomson. A formal selection and pruning algorithm for feedforward artificial neural network optimization. *IEEE Transactions on Neural Networks*, 1999.

[52] Guruprasad Raghavan and Matt Thomson. Neural networks grown and self-organized by noise. *CoRR*, 2019.

[53] Maziar Raissi, Paris Perdikaris, and George E. Karniadakis. Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. *CoRR*, 2017.

[54] Maziar Raissi, Paris Perdikaris, and George E. Karniadakis. Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations. *CoRR*, 2017.

[55] Thomas F. Rathbun, Steven K. Rogers, Martin P. DeSimio, and Mark E. Oxley. MLP iterative construction algorithm. In Steven K. Rogers, editor, *Applications and Science of Artificial Neural Networks III*. International Society for Optics and Photonics, SPIE, 1997.

[56] Francesco Regazzoni. An Introduction to Machine Learning, 2020. `https://web.microsoftstream.com/video/02a3ece9-718b-42a3-a971-9d8c8988ac6d?list`.

[57] Youcef Saad. Preconditioning techniques for nonsymmetric and indefinite linear systems. *Journal of Computational and Applied Mathematics*, 1988.

[58] José Santos, Guilherme Barreto, and C.M.S. Medeiros. Estimating the Number of Hidden Neurons of the MLP Using Singular Value Decomposition and Principal Components Analysis: A Novel Approach. 2010.

[59] R. Setiono and L.C.K. Hui. Use of a quasi-Newton method in a feedforward neural network construction algorithm. *IEEE Transactions on Neural Networks*, 1995.

[60] Yeonjong Shin. On the Convergence of Physics Informed Neural Networks for Linear Second-Order Elliptic and Parabolic Type PDEs. *Communications in Computational Physics*, 2020.

[61] Jonathan W. Siegel and Jinchao Xu. High-Order Approximation Rates for Shallow Neural Networks with Cosine and ReLU^k Activation Functions, 2021.

[62] Jocelyn Sietsma and Robert J.F. Dow. Creating artificial neural networks that generalize. *Neural Networks*, 1991.

[63] Matus Telgarsky. Representation Benefits of Deep Feedforward Networks, 2015.

[64] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient pathologies in physics-informed neural networks. *CoRR*, 2020.

[65] Sifan Wang, Hanwen Wang, and Paris Perdikaris. On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks, 2020.

[66] Sifan Wang, Xinling Yu, and Paris Perdikaris. When and why PINNs fail to train: A neural tangent kernel perspective. *CoRR*, 2020.

[67] Colby L. Wight and Jia Zhao. Solving Allen-Cahn and Cahn-Hilliard Equations using the Adaptive Physics Informed Neural Networks, 2020.

[68] Wikipedia. Two-Way Backtracking Line Search. `https://en.wikipedia.org/wiki/Backtracking_line_search`.

[69] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical Evaluation of Rectified Activations in Convolutional Network. *CoRR*, 2015.

[70] Zhi-Qin John Xu, Yaoyu Zhang, Tao Luo, Yanyang Xiao, and Zheng Ma. Frequency Principle: Fourier Analysis Sheds Light on Deep Neural Networks. *CoRR*, 2019.

[71] Dmitry Yarotsky. Error bounds for approximations with deep ReLU networks. *CoRR*, 2016.

[72] Jacek M. Zurada, Aleksander Malinowski, and Shiro Usui. Perturbation method for deleting redundant inputs of perceptron networks. *Neurocomputing*, 1997.

A. Library Structure

The aim of this short appendix is to provide a general overview of the online repository dedicated to this project, which is publicly accessible at `https://github.com/patropolimi/Thesis`. Its content can be essentially subdivided into the following main categories, for which we will supply the most salient details:

1. Basic PINN results.
2. Adaptive PINN results.
3. Python libraries.

The items related to the first and second categories are grouped in the ***Basic*** and ***Adaptive*** folders, respectively, while the internal machinery that constitutes the newly-developed Python libraries is gathered inside the ***Library*** folder. The latter is further subdivided into two subfolders (***Library/Basic*** and ***Library/Adaptive***), each clearly corresponding to the specific type of networks that can be built exploiting the related code. Other than these two, inside the ***Library*** folder we also find the *PINN_ Utilities.py* file, which contains all the useful functionalities that are needed by both of the former library parts. Each of these is then formed by the following source files, that individually carry the models applicative features in a hierarchical structure:

- *PINN_ Grounds.py*, containing the classes that implement the functional evaluation of the networks.
- *PINN_ Wrappers.py*, endowed with the mathematical functions that are used to: compute the PDE residual values, calculate the boundary conditions loss, represent the network plot and, for the related library, exploit the adaptive features of the model. Any kind of PINN wrapper (only implemented for scalar problems, in our case) contains the network's functional formulation for its evaluation through the hierarchical derivation from a class contained in the previously described source file.
- *PINN_ Problems.py* derives all the utilities provided by a wrapper class (implemented in *PINN_ Wrappers.py*) and specifies the differential problem that is being considered for the underlying model.
- *PINN_ Resolutors.py* implements the optimizer used during the learning phase of our networks, combining the ADAM method and the LBFGS technique for our specific cases. All its other functionalities are derived from a class that represents the differential problem of our interest (from *PINN_ Problems.py*).

Follows the extensive list of scalar differential problems that have already been implemented in our library:

1. Pure approximation (through the identity operator).
2. Ordinary Differential Equation including only the first order derivative (ODE).
3. n-dimensional Poisson problem.
4. Advection-Diffusion-Reaction problem.
5. One-dimensional Burger's equation.
6. n-dimensional Heat equation.
7. n-dimensional Wave equation.
8. Allen-Cahn problem.
9. Helmholtz equation.

The interested user is allowed to properly create a set of classes for the representation of any other differential problem, simply emulating the main rules and features illustrated in the classes that have already been coded. Continuing with our description of the repository, we shall explain the content of the folder named ***Adaptive***. The latter gathers a total of five test subfolders, each containing: a proper *Launch_ Script.py* file, which first has to be tuned and subsequently run in order to prompt the training phase of the models for the related test, and a series of object files that group the final state information of the model instances to which they refer. Inside the former folder we also find the *Inspect_ Main.py*, which can be executed to visualize the results and plots connected to a selected set of instances for any adaptive test. While executing, the needed instructions for its use are printed on the screen and must be followed.
Finally, we provide a brief description of the ***Basic*** folder, reporting all the results related to the prime version of the PINNs that we have employed throughout our work. Inside we find two principal subfolders, each gathering all the tests performed for the related analysis: ***Basic/Sensitivity*** and ***Basic/Convergence***. These are further subdivided into three additional categories: ***Single-Scale***, ***Multi-Scale*** and ***Generic***, which group all tests depending on the characteristics of their solutions. Each of these contains the collections of all tests performed within that framework, every one corresponding to a properly enumerated subfolder. Inside any of the latter, we find the related *Launch_ Script.py* and all the network instances that have been saved for that specific trial. Back to the previously mentioned folders (that provide, as we said, the framework characterization for their tests), we also find some utility files needed to organize the results and represent them properly.

In particular, inside the subfolders contained in ***Basic/Sensitivity*** we include:

- *Inspect_ Main.py*, which must be run following the instructions printed on the screen during its execution (as for the analogous file found inside the ***Adaptive*** folder) to visualize the plots and results related to a user-definable set of models that have already been successfully stored.
- *Organize_ Main.py*, which, as the name suggests, is a helper file that can be run to organize the results, measured through the L^2 relative error, of a specific set of networks decided by the user at run-time. This organization process simply consists in the creation of an additional target file that contains the error tables for the selected trained models.
- *Result_ Main.py*. After the successful instantiation of the result tables by running our *Organize_ Main.py*, we can actually visualize and inspect the latter by means of a simple execution of this script.

Lastly, the helper files contained inside the subfolders gathered in ***Basic/Convergence*** are:

- *Inspect_ Main.py*, whose analogous functioning has already been explained above.
- *Plot_ Main.py*, which can be run to visualize the patterns followed by the average and best relative L^2 error trends with respect to a varying number of uniformly distributed residual points.

B. Pruning Methodologies

As previously anticipated, this appendix is essentially dedicated to the presentation of a selected group of pruning techniques that are publicly available in literature. For each method, that will be treated individually, we will provide a brief mathematical motivation and a rigorous description of the underlying assumptions. More in general, we will dedicate our focus on the most relevant advantages and drawbacks that we should expect to observe for each technique. All these procedures follow the same working principle, which consists in progressively reducing the size of the network by iteratively deleting some of its free parameters, precisely the ones that are considered to be the least *salient* according to a selected working criterion. From now on, for the sake of simplicity, all the variables of the model (including the activation thresholds) will be referred to as weights. The latter are obviously treated as the independent variables of the cost functional J.

B.1. Optimal Brain Damage [30]

Optimal Brain Damage (OBD) is a pruning method that operates during the learning phase of the Neural Network. We will see that, in order to estimate the saliency of each weight, an approximate computation of some second order derivatives of J is performed at each step. Diversely from other kinds of procedures, OBD does not rely on the raw assumption that the *magnitude* of a weight necessarily corresponds to its importance in the network. After every iteration, the weight that is considered to be the least salient is definitively *pruned*.

B.1.1 Background & Assumptions

We are now ready to dig into the mathematical assumptions that will eventually lead us towards a reasonable expression for the saliency of the parameters. This approach starts by considering the Taylor expansion of the variation of J for a small perturbation (δu) of the weights:

$$\delta J \;=\; \sum_i g_i \delta u_i \;+\; \frac{1}{2}\sum_i h_{ii}\delta u_i^2 \;+\; \frac{1}{2}\sum_{i\neq j} h_{ij}\delta u_i \delta u_j \;+\; O(||\delta u||^3)$$

In the expression above, g_i represents the i-th component of ∇J, while h_{ij} is the ij-th element of its hessian. Assuming the cost functional to be indefinitely regular, we may consider this formula to be valid in general. Whenever OBD is applied, we suppose the following statements to be true:

- The network is close to a local minimum of the cost functional.
- The off-diagonal terms of the hessian of J are negligible.
- Locally, the cost functional assumes an approximately quadratic shape.

Thanks to these simplifications, we may write:

$$\delta J \;=\; \frac{1}{2}\sum_i h_{ii}\delta u_i^2$$

The core idea of OBD can be expressed as follows: the saliency (s) of a weight is quantifiable with the increase that the cost functional would undergo if we pruned it. In other words, the relevance of a weight corresponds to the worsening of the model caused by its elimination. Through a simple set of assumptions, we see that we have been able to come up with a valid criterion to determine the saliency of each parameter. In practice, we will eventually need to compute, at the end of every iteration, the quantities:

$$s_i = \frac{1}{2} h_{ii} u_i^2$$

Notice that, without the introduced approximations, we would need to compute the magnitude of δJ (consequent to the elimination of each single weight) by definition. This would require, in every pruning iteration, the evaluation of our model for as many times as the number of available weights. The expensiveness of such a procedure clearly makes it computationally unaffordable.

B.1.2 Algorithm & Features

In order to calculate a cheap approximation of the generic term h_{ii}, which appears in the expression of s_i, we can rely on a procedure that makes use of the backpropagation technique. For more details and further explanations, [30] can be consulted. Concerning our interests, we limit ourselves to highlighting the fact that the numerical burden associated to this step has the same order of magnitude with respect to the computation of ∇J by means of automatic differentiation. Nevertheless, having the necessity to compute all h_{ii} terms at each iteration, we cannot expect this method to be ranked as one of the fastest. Even more importantly, we generally do not have any guarantee about the validity of the several assumptions that we made.

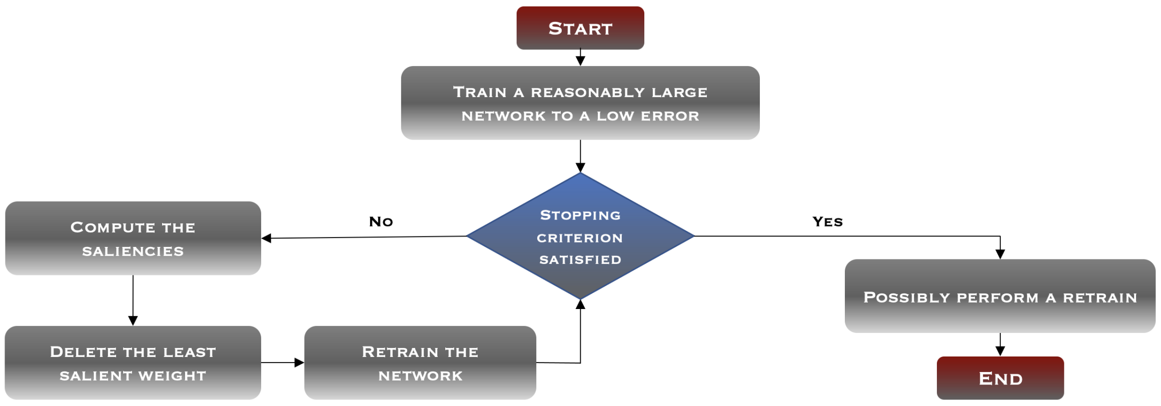

START
TRAIN A REASONABLY LARGE NETWORK TO A LOW ERROR
STOPPING CRITERION SATISFIED
NO
YES
COMPUTE THE SALIENCIES
DELETE THE LEAST SALIENT WEIGHT
RETRAIN THE NETWORK
POSSIBLY PERFORM A RETRAIN
END

Figure 48: OBD algorithm.

B.2. Optimal Brain Surgeon [20]

Contrarily to OBD, Optimal Brain Surgeon (OBS) comes on stage only after the end of the training phase of the model. Similarly to the former method, however, it involves the computation of the second order derivatives of the cost functional at each pruning iteration. As we will see, the main mathematical idea behind these two techniques is very similar, especially in the formal derivation of the expression for the saliency associated to each parameter. The latter scheme, nonetheless, avoids the rough diagonal approximation for the hessian of the cost functional, which has no heuristical interpretation. For a complete and thorough treatment of this subject and the relative framework, it is advisable to consult [20].

B.2.1 Background & Assumptions

Analogously to OBD, we define the saliency of each weight as the relative increase of the cost functional that is caused by its deletion. Our formal derivation starts again by considering the Taylor expansion of δJ for a small variation of the model parameters, δu:

$$\delta J = \nabla J^{\mathsf{T}} \cdot \delta u + \frac{1}{2} \delta u^{\mathsf{T}} \cdot H \cdot \delta u + O(||\delta u||^3)$$

In this more compact expression, H clearly stands for the hessian matrix of the cost functional.
As it has already been partially anticipated, whenever OBS is employed we implicitly hypothesize that:

- The network lies near a local minimum of the cost functional.

- Locally, J assumes an approximately quadratic profile.

These assumptions drastically simplify the expression written above, resulting in:

$$\delta J \;=\; \frac{1}{2}\delta u^{\mathsf{T}} \cdot H \cdot \delta u$$

It is also important to notice that, under these hypotheses, we expect H to be a (symmetric) positive definite matrix. This necessarily implies that the hessian is invertible and has all its diagonal elements strictly greater than zero. We may now proceed by finding out the index (q) of the weight whose elimination causes the least increase in the value of J, along with the correspondent adjustment (δu) that has to be applied in order to minimize such an increment. This problem actually consists in finding the couple $(q,\, \delta u)$ which realize:

$$\begin{cases} \arg\min_{\delta u \in \mathcal{S}_q} \frac{1}{2}\delta u^{\mathsf{T}} \cdot H \cdot \delta u \\ \mathcal{S}_q = \{\delta u : \delta u_q \;+\; u_q \;=\; 0\} \end{cases}$$

To this aim, we introduce the Lagrangian functional:

$$\mathcal{L} \;=\; \frac{1}{2}\delta u^{\mathsf{T}} \cdot H \cdot \delta u \;+\; \lambda(e_q^{\mathsf{T}} \cdot \delta u \;+\; u_q)$$

This constrained optimization problem admits the unique solution:

$$\delta u \;=\; -\frac{u_q}{[H^{-1}]_{qq}} H^{-1} \cdot e_q$$

In the last equation, q is the index of the weight that minimizes its relative saliency, given by:

$$\mathcal{L}_q \;=\; \frac{u_q^2}{2[H^{-1}]_{qq}}$$

B.2.2 Algorithm & Features

By its exact and rigorous nature, which does not involve any a priori assumption on the structure of the hessian of J, we expect OBS to be much more reliable and accurate in the choice of the weights that have to be pruned, at least with respect to OBD. We can basically consider the former as a more evolved and sophisticated version of the latter, in which the risk of eliminating a relevant weight for the network is minimized. Another fundamental advantage stems from the fact that, after every pruning iteration, OBS automatically (and optimally) corrects the values of all the other parameters of the network in order to recover the original performance. For all these reasons, we could be naively led to account OBS as a much more worthy technique. Unfortunately, things are not that straightforward. As a matter of fact, the high theoretical accuracy of this method carries around a major drawback: an immense computational burden. Indeed, even neglecting the cost of calculating the full hessian of the cost functional for several times, it must not be forgotten that the computation of the saliencies require, at each pruning iteration, the resolution of as many linear systems as the number of available parameters in the model. When we employ one of the celebrated factorization techniques in the attempt of resolving the issue, this limitation might still represent an insurmountable obstacle in terms of memory and time that would be needed to complete such an expensive task.

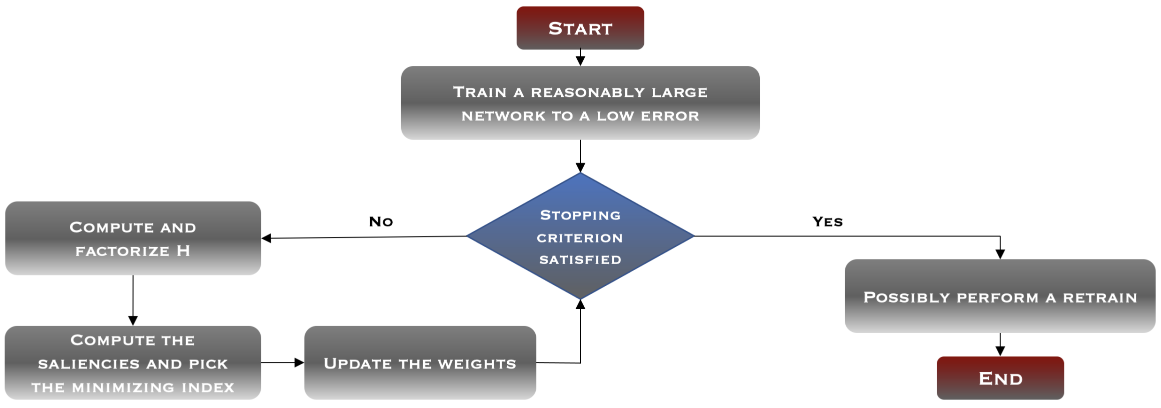

Start
Train a reasonably large network to a low error
Stopping criterion satisfied
No
Yes
Compute and factorize H
Compute the saliencies and pick the minimizing index
Update the weights
Possibly perform a retrain
End

Figure 49: OBS algorithm.

B.3. Local Relative Sensitivity Index [51]

Local Relative Sensitivity Index (LRSI) is a pruning method that, similarly to OBD, acts during the learning phase of the Neural Network. This procedure generalizes Karnin's idea (read [26]) by improving the criterion for the selection of the weights that deserve to be deleted at each iteration. The evolutionary step brought by LRSI (see [51]) consists in categorizing the parameters of the model in different subsets, grouping them according to the role they play inside the model. This approach should supposedly result in a much more qualitative and fair comparison between the weights, facilitating the job of the pruning algorithm.

B.3.1 Background & Assumptions

As we mentioned earlier, the mathematical foundations of LRSI find their roots in Karnin's approach for the computation of the saliencies. Also in this case, the relevance of each parameter is considered to be equal to the *sensitivity* of the cost functional with respect to its exclusion. However, differently from the other techniques that we have analyzed so far, Karnin derived the approximated expression of these quantities in a completely alternative way. Indeed, the magnitude of the variation δJ that follows the elimination of the k-th weight can be computed as:

$$S_k = \left| \sum_{n=0}^{N-1} \frac{\partial J}{\partial u_k}(n) \Delta u_k(n) \left(\frac{u_k^f}{u_k^f - u_k^i} \right) \right|$$

In the expression above, that arises from the discretization of an integral quantity which provides the exact estimation of δJ for this case, we recognize that all the factors can be computed and stored during the back-propagation steps of the learning phase. In the mentioned formula, N is the pruning period that we chose, n enumerates the current training step, while u_k^i and u_k^f respectively represent the initial and final values of the k-th parameter, at the beginning and at the end of the learning process.

B.3.2 Algorithm & Features

We now proceed by presenting the innovations brought by the LRSI technique, providing a heuristical justification for their usage. The essential problem that Karnin overlooked in his groundwork resides in the fact that all the parameters of the model are compared through an overall rank, regardless of their location and the role they play inside the network. This seemingly irrelevant detail might actually constitute a decisive feature in the process of selecting the correct weights to prune. Karnin's procedure, indeed, does not take into account the fact that different values of sensitivity might be uniquely related to the role that is played by distinct weights in the network. In other words, it may well be that what appears to be globally irrelevant actually turns out to be a very important cog in a local substructure of the model, possibly describing a peculiar characteristic of the phenomenon under analysis. In this case, its elimination would determine a leak of useful information for the network that we are constructing, resulting in a great and undesirable loss of precision for the model.
The last point that needs to be addressed concerns the subdivision of the parameters in smaller subcategories. Regarding it we firstly observe that, during each learning step, provided that we are executing the training phase by exploiting a gradient based technique, for all the incoming weights attached to a certain neuron (belonging to either a hidden layer or the output layer) the related variations depend only on the output coming from the unit of the previous layer at the other end of the connection. This consideration inspires the choice of the aforementioned *fair groups*, within which we can perform a proper comparison of the saliencies. For all fixed neurons (except the ones in the input layer), we will consider the sets of their incoming weights as such categories. Notice that every parameter of the model belongs to one (and only one) of these. Guided by intuition, we measure the relative importance of the generic k-th weight (belonging to the C_k-th group) through the following value, also called *Local Relative Sensitivity Index*:

$$LRSI_k = \frac{|S_k|}{\sum_{i \in C_k} |S_i|}$$

This quantity simply represents the fraction carried by the k-th parameter to the total sensitivity of the C_k-th subcategory. Among all the possible advantages of this constructive technique, it is worth highlighting that the latter does not rely on any assumption a priori. Furthermore, except for a little memory overhead, this procedure does not pay any additional price with respect to the computation of the gradient of the cost functional (needed for the learning iterations). Finally, we have the opportunity (and, unfortunately, the relative burden) to set the local thresholds that are needed for the pruning procedure.
A final comment is in order: notice that, contrarily to OBD and OBS, which normally crop only one weight at a time, LRSI may prune an arbitrary number of parameters (even zero) at each learning iteration.

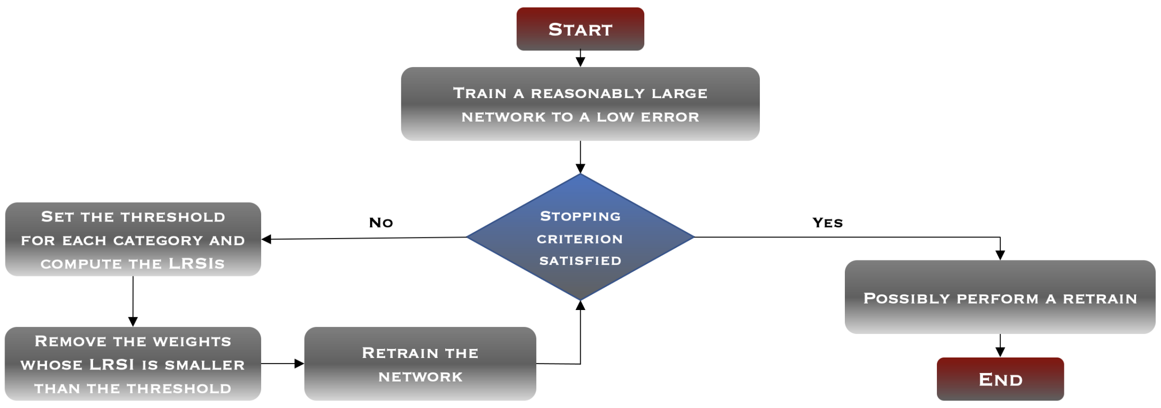

Start
Train a reasonably large network to a low error
Stopping criterion satisfied
No
Yes
Set the threshold for each category and compute the LRSIs
Remove the weights whose LRSI is smaller than the threshold
Retrain the network
Possibly perform a retrain
End

Figure 50: LRSI algorithm.

B.4. Pruning Using Relevance [47]

Pruning Using Relevance (PUR) is a technique that, analogously to OBD and LRSI, interferes with the learning phase of the network. Known for its versatilty, we can identify at least three different versions of this method, all working by following the same underlying principle. We anticipate that such procedure formally substitutes the concept of saliency with the (analogous) term *relevance*.

B.4.1 Background & Assumptions

Let us begin our discussion with the presentation of the original framework where this method was developed. The first version of PUR, instead of reducing the size of the model by pruning a single weight at a time, is characterized by the elimination of an entire neuron for every iteration, meaning that all free parameters which are directly linked to that particular unit are simultaneously deleted. In order to apply this procedure, we temporarily need to substitute the idea of relevance for a single parameter with the concept of relevance for an entire unit. Similarly to the mathematical concept of saliency, we define the relevance of the k-th neuron as the variation of the cost functional with respect to the elimination of all its connections:

$$\rho_k \;=\; J_{-k} \;-\; J$$

Unfortunately, this expression is ill-posed and cannot be exploited for the computation of ρ_k. Indeed, in order to evaluate the relevance of all the units of the network with this definition, we would firstly need to build as many architectures as the number of available neurons, and then perform an evaluation of the related cost functional for each of them. In general, this represents a worthless cost. Therefore, in order to avoid such a heavy numerical burden, we employ a smart trick: for every unit k, define the quantity $\alpha_k \in [0,1]$ as its *attentional strength*. This real number acts as a *gating coefficient* for the original output (a_k) of the relative neuron, resulting in the gated output:

$$o_k \;=\; \alpha_k a_k$$

Equivalently rewriting the expression of ρ_k in terms of the gating coefficients and assuming the cost functional to be indefinitely regular, we can take advantage of its Taylor expansion to attain a cheap approximation for the relevance. Neglecting the higher order terms, we finally obtain:

$$\rho_k \;=\; -\left.\frac{\partial J}{\partial \alpha_k}\right|_{(\alpha_i=1\ \forall i)}$$

B.4.2 Algorithm & Features

Since this method exclusively uses the first order derivatives of J with respect to the gating coefficients, it is enough to employ the tools of automatic differentiation for the computation of all the neural relevances. It is worth remarking that the attentional strength is nothing more than just a useful notational parameter, which does not influence the structure of the Neural Network nor its training phase (more details can be found in [47]). With this idea in mind and no more theoretical effort to be made, we might repeat the same identical procedure simply by substituting the weights of the model with the neurons of the network. In this second version of PUR, the attentional strength has to be analogously introduced for all the parameters. In all cases, in every iteration we prune the least relevant among all the units (weights). The final objective of this technique consists in finding a way to eliminate those neurons (weights) that slow down the learning process of the model, compromising or reducing the quality of its predictive power at the end of the training phase. A possible issue that we might encounter using this pruning method concerns the notorious *vanishing gradient problem*. Such a drawback would probably result in an highly *unbalanced* network, in which most of the pruned neurons (weights) belong to the first layers. Another viable approach is to combine the two methodologies described above, pruning *neuron-wise* for a certain number of iterations and eventually proceed by performing *weight-by-weight* eliminations. This mechanism can be regarded as a *coarse to fine* pruning scheme.

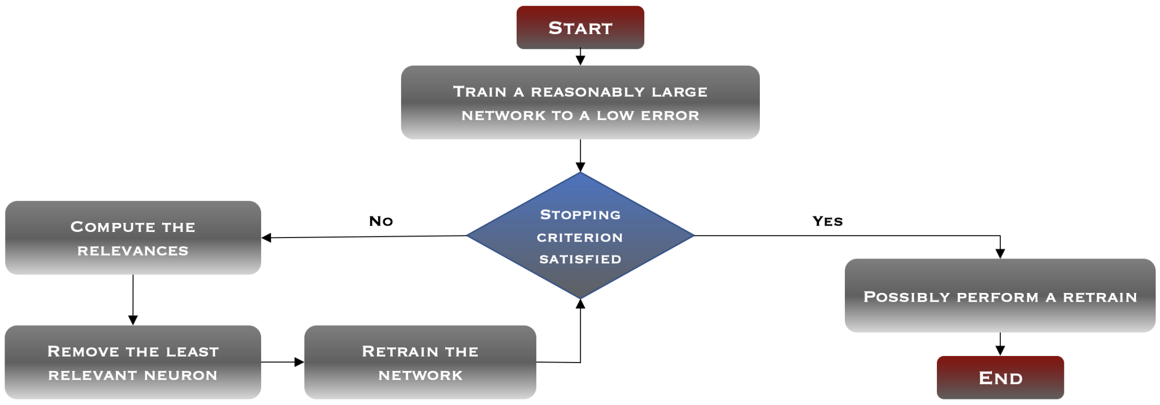

Start
Train a reasonably large network to a low error
Stopping criterion satisfied
No
Yes
Compute the relevances
Remove the least relevant neuron
Retrain the network
Possibly perform a retrain
End

Figure 51: PUR algorithm.

B.5. CGPCNE [7]

The last method that we describe is known as CGPCNE (short for *Conjugate Gradient for Pre Conditioned Normal Equations*). Similarly to PUR, it comes in at least three different versions that reflect the same basic mathematical derivation (see [7]). Moreover, along with OBS, it operates once and for all at the end of the learning phase of the models, adaptively correcting the parameters of the network in the attempt of recovering the original performance after each iteration. CGPCNE's essential operative idea consists in iteratively selecting an optimal candidate to be pruned and subsequently solving a *least squares* system to *repair*, as much as possible, the *damages* caused by the elimination of the chosen unit (weight). The goal is to minimize the overall difference in the input stimuli that each neuron receives before and after every pruning step, reducing the performance loss of the whole model. Noticeably, only the parameters that are directly or indirectly involved in these cuts undergo this procedure. In the following, we will mainly analyze the first version of this sophisticated technique, which runs with a neuron-wise pruning approach. Eventually, we will underline the operative differences that have to be taken into account for the application of the weight-by-weight and the coarse to fine alternative procedures. As the method's name suggests, a variant of the Conjugate Gradient scheme is exploited throughout the entire work (read [57] for all the details).

B.5.1 Background & Assumptions

First of all, let us introduce some useful notation:

- V denotes the set of neurons in the Neural Network.
- $E \subseteq V \times V$ is the set of (directed) connections of the model.
- $w : E \to \mathbb{R}$ represents the parameters function.
- w_{ji} represents the weight connecting neuron j to neuron i, accordingly directed.
- $P_i = \{j : (i, j) \in E\}$ is the projective field of the i-th neuron.
- $R_i = \{j : (j, i) \in E\}$ is the receptive field of the i-th neuron.
- M is the cardinality of the training set.
- f is the (differentiable) activation function of the Neural Network.

The generic i-th unit (except the ones in the input layer) receives the signal:

$$\xi_i^{(\mu)} = \sum_{j \in R_i} w_{ji} y_j^{(\mu)}$$

Clearly, $y_j^{(\mu)}$ here indicates the output of the j-th neuron for the μ-th training sample:

$$y_j^{(\mu)} = f(\xi_j^{(\mu)})$$

Ignoring (for the time being) the procedure for the selection of the appropriate neuron to be pruned, assume we need to cut off the h-th unit from the network. Pursuing the aim declared beforehand, we would like to adjust all the weights in the *receptive field* of every neuron in the *projective field* of our candidate unit, in order for the former to receive the closest possible stimulus, for every training sample, to the one they had before its elimination. In the ideal case in which we achieve a perfect matching, the model maintains unchanged its output for every training sample, consequently leaving its performance unvaried. Considering $i \in P_h$, after the removal of the h-th neuron, the i-th unit will receive its input from $R_i \setminus \{h\}$:

$$\xi_i^{(\mu)} = \sum_{j \in R_i \setminus \{h\}} w_{ji} y_j^{(\mu)}$$

Our goal is to determine the *corrective* variation vector $\bar{\delta}$ such that:

$$\sum_{j \in R_i} w_{ji} y_j^{(\mu)} = \sum_{j \in R_i \setminus \{h\}} (w_{ji} + \delta_{ji}) y_j^{(\mu)} \quad \forall \mu \in \{1, \dots, M\}$$

Equivalently, we need to solve:

$$\sum_{j \in R_i \setminus \{h\}} \delta_{ji} y_j^{(\mu)} = w_{hi} y_h^{(\mu)} \quad \forall \mu \in \{1, \dots, M\}$$

These conditions give rise to $M \cdot p_h$ linear equations, where p_h indicates the cardinality of P_h. Denoting with r_i the cardinality of R_i, the number of unknowns to be determined is given by:

$$\kappa_h = \sum_{i \in P_h} (r_i - 1)$$

In order to obtain a compact formulation, let $i \in P_h$ and define:

$$\bar{y}_i = [y_i^{(1)}, y_i^{(2)}, \dots, y_i^{(M)}]^T \in \mathbb{R}^M \quad , \quad Y_{i,h} = [\bar{y}_{j_1}, \bar{y}_{j_2}, \dots, \bar{y}_{j_{r_i - 1}}] \in \mathbb{R}^{M \times (r_i - 1)}$$

In the expression of $Y_{i,h}$, the index j_k varies in the set $R_i \setminus \{h\}$. Also introduce the quantities:

$$\bar{\delta}_i = [\delta_{j_1 i}, \dots, \delta_{j_{r_i-1} i}]^T \in \mathbb{R}^{(r_i - 1)} \quad , \quad \bar{z}_{i,h} = w_{hi} \bar{y}_h \in \mathbb{R}^M$$

Solving the aforementioned problem is equivalent to finding a solution to the following:

$$Y_{i,h} \, \bar{\delta}_i = \bar{z}_{i,h} \quad \forall i \in P_h$$

Fancying the notation even further, set:

$$\bar{\delta} = [\bar{\delta}_{i_1}^T, \dots, \bar{\delta}_{i_{p_h}}^T]^T \ , \ Y_h = \begin{pmatrix} Y_{i_1,h} & & \\ & \ddots & \\ & & Y_{i_{p_h},h} \end{pmatrix} \ , \ \bar{z}_h = [\bar{z}_{i_1,h}^T, \dots, \bar{z}_{i_{p_h},h}^T]^T$$

We are now ready to ultimately group these p_h independent linear systems into:

$$Y_h \, \bar{\delta} = \bar{z}_h$$

B.5.2 Algorithm & Features

As a matter of fact, the final system that we retrieved is practically always overdetermined. This leaves us with no other choice other than solving it using the least squares approach. However, due to its potentially enormous size, it may well happen that Y_h is not full rank, resulting in a multiplicity of solutions for our vectorial equation. Since no particular meaning is given to the latter, we are ready to accept any of these solutions, regardless of the way in which they are obtained. As we already remarked, the resolution of our system will be performed iteratively, by means of an efficient variant of the celebrated Conjugate Gradient method. Concerning the criterion for the choice of the unit to be pruned at each iteration, we may solve:

$$h = \arg\min_{q \in V_H} ||\bar{z}_q - Y_q \, \bar{\delta}_f(q)||^2$$

With V_H we indicate the set containing all available hidden neurons in the Neural Network, while $\bar{\delta}_f$ represents the numerical solution reached by the execution of CGPCNE. The rationale behind this choice is pretty straightforward: we seek for the unit whose elimination causes the minimum residual in the correction of the weights. This procedure, by construction, ensures that the model will suffer from the minimal impact concerning its loss of performance over the training set. Unfortunately, however, an important issue immediately arises. The mentioned problem resides in the fact that, in order to proceed with this mechanism and find all the solutions $\bar{\delta}_f$ necessary for the computation of the residuals, at each step we would need to complete a number of full CGPCNE cycles that is equal to the cardinality of V_H. Since this would be computationally overwhelming, in the expression above we substitute $\bar{\delta}_f$ with a rough approximation, which is represented by the null vector. Hence, in the end, we pinpoint the index of the unit to be pruned by solving:

$$h \;=\; \arg\min_{q \in V_H} ||\bar{z}_q||^2$$

Writing this formulation in an equivalent way (exploiting the expression for $\bar{z}_q$):

$$h \;=\; \arg\min_{q \in V_H} \sum_{i \in P_q} w_{qi}^2 ||\bar{y}_q||^2$$

Let us now provide a heuristical interpretation to the quantity:

$$\sum_{i \in P_h} w_{hi}^2 ||\bar{y}_h||^2$$

Considering the h-th unit of the model, we can label the expression written above as its *synaptic activity*, a simple but meaningful measure of its importance. This observation justifies our heuristic pruning criterion: at each step, we delete the least active neuron.

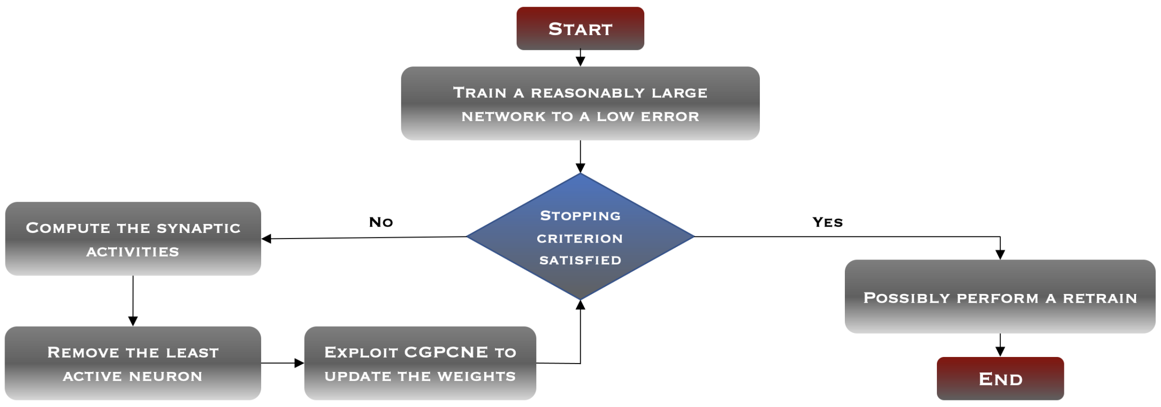

Start
Train a reasonably large network to a low error
Stopping criterion satisfied
No
Yes
Compute the synaptic activities
Remove the least active neuron
Exploit CGPCNE to update the weights
Possibly perform a retrain
End

Figure 52: CGPCNE algorithm.

As we previously anticipated, this method might be also exploited in its alternative form where we prune a single parameter at a time. Just by repeating all the conceptual steps that we formerly described, we end up with an analogous mathematical formulation in which we actually solve a sub-problem of the original scheme. If w_{hi} is the chosen weight to be pruned, we simply resolve:

$$Y_{i,h} \; \bar{\delta}_i \;=\; \bar{z}_{i,h}$$

Analogously, we replace the former selection procedure with the problem of finding:

$$(h, j) \;=\; \arg\min_{(q,i) \in E} w_{qi}^2 ||\bar{y}_q||^2$$

As an alternative route with respect to the two versions of the algorithm that we just described, we can decide to reduce the size of the network using a coarse to fine approach, in which we initially prune neuron-wise and then proceed by refining the model with a weight-by-weight elimination process (as for PUR).
Unlike all the other pruning techniques that we presented, CGPCNE does not require the computation of any kind of derivative. Even more importantly, for this procedure we can appreciate a very strong (but nonetheless extremely easy) mathematical foundation. On the other hand, a liability that we might expect to encounter, especially in presence of highly overdetermined systems, concerns the loss of performance caused by the elimination of the units (weights). Under those circumstances, in fact, we cannot have any guarantee that the corrections brought by this technique will be able to contrast the negative effects of pruning. We finally recall that, in order to use this method, we need to properly tune the so-called *relaxation parameter*, that plays a crucial role in the resolution of the least squares system.

Abstract in lingua italiana

Oltre che per una moltitudine di branche in cui intervengono tecniche di matematica applicata, i Deep Neural Networks sono recentemente emersi anche per applicazioni connesse con lo studio e l'analisi delle Equazioni Differenziali alle Derivate Parziali, diventando noti in questo contesto come Physics-Informed Neural Networks. In questo lavoro focalizzeremo la nostra attenzione sulle relative performance nell'approssimare la soluzione esatta per un generico insieme di equazioni differenziali scalari. Durante la nostra esposizione ci aspettiamo di trovarci ripetutamente di fronte alle loro qualità e, allo stesso tempo, riscontrarne i noti difetti che tuttora limitano il loro sfruttamento in molte applicazioni reali. In questo senso, uno dei più noti svantaggi ad essi connesso consiste nel noto problema della scelta di una conveniente architettura per la risoluzione del problema considerato. Dopo un'analisi approfondita dei risultati riguardanti la versione base del PINN, dove studieremo la relazione tra l'accuratezza dei modelli ottenuti e le loro caratteristiche strutturali (numero di hidden layers, quantità di neuroni per layer e cardinalità del training set), daremo una overview generale di alcune tecniche che vengono accompagnate al loro utilizzo in letteratura. Successivamente proporremo una giustificazione euristica e la relativa implementazione di un nuovo schema adattivo per i PINN, che affianca il noto metodo di Growing alla tecnica conosciuta come Residual Adaptive Refinement. Lo strumento tecnologico che verrà sfruttato per la costruzione di tutti i modelli di questo progetto si basa su una semplice e nuova libreria Python, sviluppata per la risoluzione di Equazioni Differenziali alle Derivate Parziali su domini iper-rettangolari mediante una procedura di apprendimento basata sull'impiego combinato degli ottimizzatori ADAM e LBFGS. Il materiale di questo lavoro è integralmente e pubblicamente accessibile seguendo il link `https://github.com/patropolimi/Thesis`.

Parole chiave: Intelligenza Artificiale, Reti Neurali, Equazioni Differenziali alle Derivate Parziali, Adattività

Acknowledgements

Un carissimo e sentito ringraziamento è innanzitutto dedicato al Professor Marco Verani ed al Professor Edie Miglio, i quali mi hanno accompagnato negli ultimi mesi dell'Università con il loro appoggio e tanti preziosisissimi consigli. Vi sarò sempre grato per il vostro aiuto, conservando il bellissimo ricordo di due persone meravigliose, ancor prima che eccellenti insegnanti.

Un ringraziamento altrettanto speciale per i miei genitori: Mamma Claudia e Papà Stefano. Mi avete dato la possibilità di crescere e mettermi alla prova in questo lungo e tortuoso percorso che, ancor prima che l'inizio della mia carriera professionale, ha rappresentato la più grande lezione della mia vita fin qui. Grazie per il vostro infinito affetto, il vostro costante sostegno e per avermi sopportato in questi anni complicati.

Un altro ringraziamento che viene dal cuore è dedicato a Cecilia e Federica. Le parole, purtroppo, sono un mezzo di comunicazione troppo freddo per esprimere quello che rappresentate per me. Per questo motivo mi limito ad avvisarvi del fatto che voi, più di chiunque altro, dovrete sopportarmi ancora per un bel pò.

Ultimo ma non ultimo, tengo a salutare affettuosamente la mia Professoressa di Matematica delle scuole superiori, Maria Alberta Montruccoli. Senza la sua passione, i suoi insegnamenti e la determinazione che ha acceso in me in quegli anni, difficilmente avrei potuto apprendere quale fosse il percorso universitario giusto per me. Probabilmente non lo sa, ma per me rappresenta un grande ed indelebile esempio.