uc3m | Universidad **Carlos III** de Madrid

Master Degree in Computational and Applied Mathematics

Academic Year (e.g. 2021-2022)

Master Thesis

"Using Artificial Neural Networks To Solve PDEs in regions with non-trivial boundaries."

Santiago Morales Saldarriaga

Pedro González Rodríguez

Leganés, 2022

SUMMARY

In this work, we explored a computational implementation of the method described in [1] to solve PDEs using Physics-informed Neural Networks. We show how to use the state-of-the-art tools available in TensorFlow to build and train the PINN, and also, to approximate the differential operator including automatic differentiation. Using this, we solve the advection and diffusion equation in one dimension to illustrate the method, and then we do it in a complex domain in two dimensions in order to test its capacity to solve PDEs in this kind of domain.

Finally, we analyze the results by comparing the computed value with the real one and looking at the error evolution through the PINN training. From this, we make a brief review of the method, its major benefits and disadvantages and the possible work-to-do.
Keywords: Physics-informed Neural Networks (PINN), Artificial Intelligence, Machine Learning, Partial Differential Equations (PDE), Python, TensorFlow

DEDICATION

A mis padres, Carlos y Cristina.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1. Motivation

In real applications, it is very common to find PDEs that can not be solved analytically, and thus they must be solved numerically. Most numerical methods are based in a discretization of the domain, that is, the domain is covered with a grid, and the method approximates the solution over the grid's nodes. How this discretization is being done is called the meshing problem. These methods are, usually, very efficient in regular geometries, in which the meshing problem is trivial or already solved, but in more complicated geometries, where the meshing problem is not trivial these methods lose their efficiency since the meshing could be as complicated as solving the PDE itself.

In order to solve PDEs in this kind of geometries, it is convenient to use methods that not require solving the meshing problem. This kind of methods are called meshless methods. One of the techniques used to develop meshless methods, and the one used in this work, is to reduce the dimension of the solution by writing the infinite - dimensional solution as a finite - dimensional one, that will be defined by a certain amount of parameters. In this case, this will be done be assuming that the solution will have the form of an Artificial Neural Networks (ANN).

Then, since the solution will be in a finite - dimensional domain, we will require that it fulfills the equation in a finite amount of point. Using these points and the solution there, we can build a system of equation. In this case this system will be non-linear since the equations contains the output of a PINN. So, we will reformulate the problem in order to solve an optimization problem instead.

With this technique, we use a set of random points in the domain, and approximate the solution by solving the problem only in this set, instead of using also the relation between the points as is done in mesh methods. The major benefit of using these methods is that generating the points turns to be a very simple task, it could be done by using Monte Carlo Methods to generate points near the area of interest and just check if the point is inside or not.

Therefore, to summarize, a set of collocation points generated using Monte Carlo Methods inside an arbitrary geometry will be used to solve a PDEs by using ANN. ANN are a very useful tool in order to solve PDEs since, if an analytical smooth function is used as activation function (like sigmoid), they are also analytical, smooth functions. This technique where a deep learning tool is used to solve problems involving PDEs is called Physics-Informed Neural Networks (PINNs).

PINNs is a method that uses NN whose input is a point in the domain and its output, once it is trained, give a estimation of the solution of the PDE in that point. The principal novelty of PINNs is that this technique use physical information of the problem in order to turn it into a non-supervised problem, so is not required to label data to solve it [2]. In this way is possible to resume the two major benefits of using PIINs for solving PDEs as:

- Is a mesh-free technique, therefore is possible and computationally viable to solve PDEs in non-trivial geometries in which constructing the mesh is not computationally efficient.
- It is a non-supervised technique, so it is no necessary to know anything about the solution before trying to solve it.

The use of PINNs techniques to solve PDEs could be traced back to [3] when Dissanayake and Phan-Thien introduced the concept of including the physical information about the problem into a machine learning algorithm. Since then, many other methods have been introduced to use the deep learning in the solution of PDEs. With the exponential growth of computational power in the early 2000s and the open collaboration in the deep learning area, models with more parameters and more sophisticated structures were now available to be used and be trained, this leads to new open-source tools like TensorFlow [4] and the implementation of Automatic differentiation in it, which is based in the backpropagation algorithm, an specific kind of Automatic differentiation for neural networks.

In [1] it is shown a PINN technique in which the result of the ANN is used to obtain the result of the Boundary Value Problem (BVP) by using an ansatz that includes the physical information of the problem.

The following work will show the implementation using Python and state-of-the-art resources in it for managing ANN like TensorFlow (including automatic differentiation), NumPy, Keras, and Shapely.

1.2. Theoretical Framework

1.2.1. Collocation points

Collocation methods are used to solve boundary value problems in one dimension or in higher dimensions. These methods are fundamentally based in representing the solution to a problem in a finite - dimensional approximation space. These finite - dimensional spaces are usually defined by linear bases like polynomials (or splines).

In this work, we will use a slightly different approach. In this case we will not use a finite basis to approximate a function of an infinite dimensional space, instead we will use an ANN, which is, intrinsically, a non-linear approach. Although the system is not built on a linear basis, still is possible to take it as a finite - dimensional one. This is because we are representing a function, that is in an infinite dimensional space, y, as a function that is defined by a finite amount of parameters $y^L(x; w, b)$, this implies that, although in a complex and non-linear way, the space in which $y^L(x; w, b)$ belongs is finite - dimensional.

When the collocation methods are used with polynomials (or any linear basis) the solution is going to be a linear combination of the vectors of the basis that span the finite-dimensional space. This is not our case because we are dealing with a non-linear representation, but still in both cases the problem turns to find the parameters that define the representation.

To do this, in the linear case, we would use $N - M$ collocation conditions in order to obtain a $n \times n$ system of linear equations with M boundary conditions [5] and solve it with the usual Linear Algebra methods. In our case, we can not use this since we do not have a linear system. The non-linearity of the problem needs to be tackled differently. For this reason, we will introduce in sections 2 and 3 an optimization approach to solve this problem.

1.2.2. Artificial Neural Networks (ANN)

ANN were originally designed to replicate the function of our brain and how the brain learns from experience. In the beginning, started as a purely academic topic with few to none applications in real life problems due to the high need for resources necessary to train them.

In the end of the 90s, and the beginning of the 00s, with the development of new hardware with many more resources, and new open-source implementations (like TensorFlow itself) ANN became very popular because it performed better than the models of the time for supervised tasks, where the goal is to predict an output from an input by looking other example of the same prediction task. Since then, ANN is established as the state-of-the-art way to solve this kind of problems. In this work it will be shown that it also works for non-supervised problems if we include extra information, in this case using the physical information about the PDE (PINNs), but before it is important to understand what is the general structure of an ANN.

Since the idea is to replicate the way the brain behaves, an ANN is also composed of neurons, also called perceptrons. In the same way as in the brain cells the artificial perceptron receives a stimulation (called input) and in response to that stimulation the perceptron is activated or not (i.e. the output could be 1 or 0, respectively) following determined rules (activation function). This can be observed in Fig 1.1, where the inputs are the Xs values and a bias (the 1 on top), that, when they entered the perceptron, are multiplied, each one, by a different weight, the sum of the resultant values is compared to a threshold to determine if the perceptron output is 0 or 1. To summarize, the perceptrons are composed of an activation function, the same for everyone, that is defined a priori, and a series of variables, weights, whose values will be calculated so they minimize a loss function (training) that will depend on the task we want the ANN to fulfill.

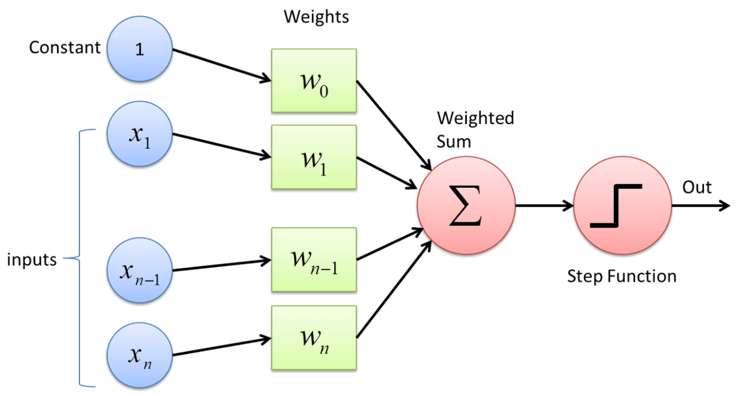


Fig. 1.1. Graphical description of a perceptron. Image taken from: *What the hell is a perceptron.* [6]

In a fully connected neural network, the layer i is composed of N_i perceptrons and all of them receive as inputs the N_{i-1} outputs of the layer $i-1$. The first layer will receive the inputs of the model and the last layer will output the output of the model, that could be just one value if is a prediction task or M values if is a classification task, in this case each value represents the probability of the input being each class. This work is using a "prediction" task to solve PDEs, therefore it will be done by using multilayer ANN with only one output.

1.2.3. PINNs method using an ansatz[1]

As explained before, in order to apply PINNs to solve PDEs it is necessary to include the physical information of the problem in the model. In this work, this information will be included using the method introduced in [1].

First, it is important to understand the form of the problems that are going to be solved In this work in order to construct the ansatz function. The problems that are going to be solved are stationary PDEs that have a general form:

$$Lu = f \quad x \in \Omega \tag{1.1}$$

$$Bu = g \quad x \in \Gamma \subset \partial\Omega \tag{1.2}$$

where L is a differential operator, B the boundary operator that is the identity operator if Dirichlet conditions are being considered, f is a forcing function, g the boundary data, $\Omega \subset \mathbb{R}$ the domain of interest and Γ is the part of its boundary ($\partial\Omega$) where the boundary conditions are imposed.

From (1.1) and (1.2) it is possible to rewrite the problem as an optimization problem in which the solution of the PDE is the u that minimize:

$$|Lu - f|$$
$$|Bu - g|$$

This would be a very hard problem to solve, since u could be of any form. So, in order to be able to solve this using an optimization approach, we use, as is done in [1] that the solution u can be written using the following ansatz:

$$\hat{u} = G(x) + D(x)y^L(x; w, b) \tag{1.3}$$

where $D(x)$ is a smooth extension of the distance function defined as

$$d(x) = \min_{x_b \in \Gamma} ||x - x_b|| \tag{1.4}$$

and $G(x)$ is a smooth extension for the boundary function (for the one dimensional case $G(x)$ is directly the boundary condition)

The calculation of these smooth extension function will be done by using small, regular, ANNs that use $g(x)$ and $d(x)$ as the labels for the supervised learning task, this ensures smoothness and because of the form of $\hat{u}$ it also ensure that the boundary will be trained using the boundary (known) data.

The other term, y^L, will be the output of the ANN that is going to be used for the PINN method, observe that the physical information is included elegantly in the ansatz function by multiplying y^L by the distance function, and therefore giving more strength to the values of the boundary near this. How the ANNs are being calculated is going to be described in the methodology section.

By using this ansatz, we are separating the solution in such a way that the information about the boundary conditions is, exclusively, in $G(x)$ since at those points we have that $D(x) < \epsilon$ for ϵ an arbitrary small number. In this way, we are training the PINN (y^L) to solve the equation only for $x \in \Omega$, and we do not care what values take $y^L \in \Gamma$ since there $D(x)$ is 0. Therefore, the ansatz is including, implicitly, the information about the boundary conditions and is not necessary to include that in our loss function.

Therefore, the optimization problem can be rewritten as finding the parameters b^*, w^* such that:

$$b^*, \vec{w}^* = argmin_{b,\vec{w}} \{||L\hat{u} - f||_2\} \tag{1.5}$$

Observe that, since the L2-norm is non-negative, the smaller value of the right side of the former expression is reached when $L\hat{u} = f$, that is, joined with the boundary conditions included in the ansatz, exactly the solution of the original problem.

1.3. Goals

The major goal of this work will be to show the implementation of the PINN method 1.2.3 in Python, and use it to solve arbitrary diffusion and advection problems in 1D and in 2D. This will be done by using state-of-the-art tools included in Python library TensorFlow [4] like automatic differentiation (using Gradient Tapes) and optimization of loss function (using backpropagation, also in Gradient Tapes).

Furthermore, in this work will be shown how to use libraries such Shapely [7] to represent the solution in non-trivial geometries, in this case star will be taken as an example of a non-trivial shape, but following this, another arbitrary shape like country maps could be implemented directly.

Finally, using this non-trivial geometry as domain, we will solve diffusion and advection problems by using the PINN method that is described in this work, and compared the results with the real (known) value, in order to evaluate the performance of the method.

2. STATE OF THE ART

The beginning of the study of PINNs to solve PDEs could be traced back to the last years of the former century when Lagaris et al use neural networks to determine some unknown parameters used (along with some constant terms) to solve differential equations [8]. Although this was something novel by the time, it does not solve one of the principal disadvantages of the former methods, since this cannot be used to solve problems in non-trivial geometries. After that, they extend the method in [9] in order to solve also problems in more complicated geometries, but it was not until the development of more powerful processors and the inclusion of graphic processing units (GPU) in the decades of 2000s and 2010s that more complex ANNs could be implemented.

The sophistication of the software, the understanding of better ways to train neural networks using GPUs and TPUs and the open-source movement impulse in a big way by the liberation of TensorFlow and the creation of new tools like Pytorch cause an exponential increase of works in this area. This explosion is described in a more detailed way in [2] where is also shown how this derived in the application of PINNs to other kind of equations such as integrodifferential equations (IDEs) or stochastic differential equations (SDEs).

Also, it is explained in [2] how PINNs may be used to solve inverse problems, which consist in finding the parameters that produced the data, that is usually obtained by observation. Some examples of inverse problems are characterizing fluid flows or temperatures obtained using physical sensors.

Finally, in [1] it is shown one particular implementation that includes the boundary conditions information of the problem by using the ansatz described in 1.2.3. They propose an approach in which the calculations of the derivatives are done from scratch, coding the full backpropagation for each derivative. This is, probably, the faster implementation but is difficult to scale for more complex derivatives or even just for different second-order problems. On the other hand, we propose to calculate the derivatives using the tools that are already implemented (and extensively tested) in TensorFlow. This allow us to implement it in a more robust, understandable and "Pythonic" way, which could help to other researchers to replicate and extend the ideas for more general projects.

3. IMPLEMENTATION

3.1. Tools, Algorithms and Methodology

For this project, all the code was written using Google Colab [10] in order to take advantages of the computational resources that are offered and to make more agile the communication and the debugging. Also, by using Colab in this work it is ensured that all the results are easily reproduced since all the code is being run in a custom virtual machine.

3.1.1. Tools

Being so, as one of the most important things in science is the reproducibility of the results all the versions of the software use are described next:

- Python (version 3.7.13) [11]
- TensorFlow (version 2.8.2) [4]: Package for building, training and deploying deep learning models in Python.
- Keras (version 2.8.0) [12]: TensorFlow subpackage for simplifying the managing of standard deep learning models.
- Numpy (version 1.21.6) [13]: Package for calculations using multidimensional arrays (tensors) in Python.
- Matplotlib (version 3.2.2) [14]: Package for create visualizations
- Shapely (version 1.8.4) [15]: Package for manipulation and analysis of geometric objects in the Cartesian plane.

All this software is running in a Linux-based virtual machine provided by Google for free.

3.1.2. Algorithms

In this work, we will use the Automatic Differentiation algorithm (AutoDiff) to calculate the derivatives inside the training loop. AutoDiff is not near to be a novel mathematical development since it could be traced back to the decade of 1950s ([16] for example) as an algorithm in the area of Differential Algebra, but its use in Deep Learning state-of-the-art applications just since the decade of 2010s (inside the training loop) when the big boom in computational power allow training big models that includes derivatives as one part of the loss function [17].

One of the more important cases of use of the AutoDiff algorithm is Backpropagation. Backpropagation, is a well establish state-of-the-art algorithm to train deep learning models using labeled data, and is also used in [1].

Next, both algorithms are explained, being more exhaustive in the explanation of the AutoDiff algorithm since Backpropagation is just the special case when AutoDiff is used in the calculation of the gradients of an ANN.

In this work, there have been used two major algorithms in order two use the PINNs in the way described in [1]. The first one, is Backpropagation, which is a well establish state-of-the-art algorithm to train deep learning models using labeled data, and is also used in [1]. The second is Automatic Differentiation (AutoDiff). AutoDiff is not near to be a novel mathematical development since it could be traced back to the decade of 1950s ([16] for example) as an algorithm in the area of Differential Algebra, but its use in Deep Learning state-of-the-art applications just since the decade of 2010s since the big boom in computational power allow training big models that includes AutoDiff as one part of the loss function [17]. Next, both algorithms are explained, being more exhaustive in the explanation of the AutoDiff algorithm since Backpropagation used AutoDiff as its main step.

Automatic Differentiation (AutoDiff) [18]

To define what AutoDiff is, first is important to understand what AutoDiff is not.

AutoDiff is not numerical differentiation. For numerical differentiation is necessary to make a discretization of the space, and for this is its complexity increases with the dimension n of the space that is going to be discretized (increase like $O(n)$). The effiency of AutoDiff, on the other hand, does not depend on the dimensionality of the problem. This makes that AutoDiff look like a reasonable technique for implementations in which the space is not restricted to be low-dimensional.

In particular, in the scope of this project, the derivatives are not being calculated with respect the variables of the physical space, but instead with respect the parameters of the ANN, so the increase in complexity could be very problematic, even making it impossible to calculate the derivatives since n in this case could be millions or billions (which are already the numbers of parameters in state-of-the-art deep learning models). Although this is not exactly the case in this work, since AutoDiff does not increase in complexity with dimensionality, then it is not a problem to extrapolate to larger models.

AutoDiff is not symbolic differentiation. In symbolic differentiation, the value of the derivative is calculated by applying transformations on the symbolic representation of the original expression, leading first to the result, also, in symbolic terms. This could be helpful to identify optimal points where derivative's value is 0 and avoid unnecessary calculations.

The bad part about symbolic differentiation, and also what make it useless for machine learning purposes, is that calculating the value of a derivative expression is not even a NP problem but an exponential one, which means that the calculation of an expression derivative could be exponentially larger than the original expression.

A clear example of this is the calculation of the derivative of a multiplication, if $f(x) = g(x)h(x)$ then $f'(x) = g(x)h'(x) + g'(x)h(x)$, but $g(x)$ and $g'(x)$ are very likely to have common terms, common terms that are being duplicated, this happened with each multiplication. This problem is called expression swell

To get rid of this problem, it is possible to reduce the complexity by storing the values of intermediate expressions. In this way it is not necessary to recalculate them each time they appear, propagating only its values along the calculation of the more complex terms. This technique is known as Forward Accumulation, and established the basis for AutoDiff. This is showed with more detail later in the example of the Forward Accumulation technique. There we can see that the result calculated by AutoDiff is not a symbolic expression.

Now, to understand what AutoDiff is, it is also important to understand that AutoDiff is not a unique technique, but a set of such, that use the chain rule to exploit the fact that every computational calculation is based on elementary operations for which derivatives are known [19]. Therefore, by applying chain rule repeatedly to a function, it is possible to calculate derivatives of an arbitrary order.

For this to be done, it is needed to save all the calculations done in the function which its derivative are going to be calculated. By doing this, it is possible to rewrite an arbitrary function as a composition of N elementary functions, as follows:

$$F(x) = (f_N \circ f_{N-1} \circ ... \circ f_2 \circ f_1)(x)$$

Then, if we have:

$$\begin{aligned} f_0(x) &= y_0 \\ f_1(y_0) &= y_1 \\ f_2(y_1) &= y_2 \\ &\vdots \\ f_{N-1}(y_{N-2}) &= y_{N-1} \\ f_N(y_{N-1}) &= y_N = F(x) \end{aligned}$$

Therefore, it is possible to calculate the first derivative of $F(x)$ as:

$$\frac{dF(x)}{dx} = \frac{df_N(y_{N-1})}{dy_{N-1}} \frac{df_{N-1}(y_{N-2})}{dy_{N-2}} \cdots \frac{df_2(y_1)}{dy_1} \frac{df_1(y_0)}{dy_0} \frac{df_0(y_0)}{dy_0}$$

Where $\frac{df_0(y_0)}{dy_0} = 1$.

There are two ways of how to do the calculation of the chain rule, depending on how the elements are traversed:

1. Forward accumulation:

   The forward accumulation is the simplest one. Then, to calculate the derivative of $F(x, y)$ it starts by associating each intermediate variable, y_i, with its derivative $\partial y_i / \partial x$.

   Then, rewriting the function as a composition of elementary function, it is possible to construct the Forward Primal Trace, which could be used to calculate the derivative of each intermediate variable (because all the derivatives of elementary functions are all known), with this value it is possible to calculate the evolution of the derivative value, also called the Forward Tangent Trace. Therefore, by evaluating both the primals and the derivatives, it is possible to calculate the derivative of the target function that should be equivalent to calculate the derivative of the last primal value.

   Example:

   Next, it is shown step-by-step the calculation of both the primal and the derivative trace for the function:

   $$F(x_1, x_2) = \exp(x_1) + \cos(x_2) + x_1 x_2$$

   and its derivative $\partial F / \partial x_1$, respectively. Using the values $x_1 = 1$, and $x_2 = 2$

Forward Primal Trace	Forward Derivative Trace
$y_0 = x_1 = 1$	$\dot{y}_0 = \partial x_1 / \partial x_1 = 1$
$y_1 = x_2 = 2$	$\dot{y}_1 = \partial x_2 / \partial x_1 = 0$
$y_2 = \exp y_0 = \exp 1$	$\dot{y}_2 = \dot{y}_0 \exp y_0 = 1 \times \exp 1 = 2.718$
$y_3 = \cos y_1 = \cos 2$	$\dot{y}_3 = -\dot{y}_1 \sin y_1 = 0 \times \sin 0 = 0$
$y_4 = y_0 y_1 = 1 \times 2$	$\dot{y}_4 = y_0 \dot{y}_1 + \dot{y}_0 y_1 = 1 \times 0 + 1 \times 2 = 2$
$y_5 = y_2 + y_3 = 2.718 - 0.416$	$\dot{y}_5 = \dot{y}_2 + \dot{y}_3 = 2.718 + 0$
$y_6 = y_5 + y_4 = 2.302 + 2$	$\dot{y}_6 = \dot{y}_5 + \dot{y}_4 = 2.718 + 2$
$F(x_1, x_2) = y_6 = 4.302$	$\partial F(x_1, x_2) / \partial x_1 = \dot{y}_6 = 4.718$

   By reading this example, it is clear why AutoDiff is different to both, numeric and symbolic, differentiation. Further, it is possible to understand why is so powerful in the scope of machine learning and specifically for PINNs.

   AutoDiff never use the nature of the space or how this is being discretized, so in a manner could be seen as a more generic tool than numeric differentiation. But also, although is clearly based in the logic of symbolic differentiation, AutoDiff solves the complexity problem by focusing only in the value of the terms and save them to avoid useless recalculations that can not be avoided in symbolic differentiation.

2. Reverse accumulation:

   In this case, as in forward accumulation, every intermediate variable is complementing with an auxiliary variable such that, for an intermediate variable, y_i the auxiliary variable is:

   $$\bar{y}_i = \frac{\partial F}{\partial y_i}$$

   Where F is the output function and this term represents how a change in the intermediate variable y_i would affect the output function.

   The first thing to remark is that in this case, all the values of the intermediate variables are needed to start the calculation of the auxiliary terms, instead of doing it at the same time as in the forward accumulation.

   In the case of the backpropagation algorithm used to train ANNs the intermediate values are the weights of each layer, therefore by calculating the derivative of the cost function with respect to each weight it is possible (assuming the function is convex) to find its minima.

   Although, at first hand, this algorithm seems to be more complex than the forward implementation since it has to traverse the trace twice instead of once. And this is true if just one derivation is needed, but this is hardly the case, and specifically is not the case in the scope of ANNs where it is necessary to calculate the derivative with respect of the N parameters of the network, and, as it was established, this value of N could go to millions or even billions.

   In these cases where N is a large number, clearly the reverse accumulation will be much more efficient than the forward one because in just two phases (advancing forward to calculate the intermediate values and backwards to calculate the auxiliary values) the algorithm can calculate the N derivatives instead of requiring N forward runnings to calculate each derivative, which will be the case for the forward accumulation algorithm.

   Next, this will be shown by calculating the derivative of the same example used before, but now by running the reverse accumulation algorithm instead of the forward one.

   To read the following example, it is important to understand the flow in which the calculations are being done. First, we calculate the Forward Primal Trace, as usual, obtaining the values for each intermediate variable. Then it comes the tricky part, for the calculation of the Reverse Derivative Trace we calculate the derivatives from the output to the input by checking step-by-step the calculations done in the Forward Primal Trace backwards. In the next example is possible to observe this better:

$F(x_1, x_2) = \exp(x_1) + \cos(x_2) + x_1 x_2$ for $x_1 = 1$, and $x_2 = 2$

Forward Primal Trace

$$
\begin{aligned}
y_0 &= x_1 = 1 \\
y_1 &= x_2 = 2 \\
y_2 &= \exp y_0 = \exp 1 \\
y_3 &= \cos y_1 = \cos 2 \\
y_4 &= y_0 y_1 = 1 \times 2 \\
y_5 &= y_2 + y_3 = 2.718 - 0.416 \\
y_6 &= y_5 + y_4 = 2.302 + 2
\end{aligned}
$$

$$F(x_1, x_2) = y_6 = 4.302$$

Reverse Derivative Trace

$$\bar{y_6} = \bar{F}(x_1, x_2) = \partial F / \partial F = 1$$

$$
\begin{aligned}
\bar{y_5} &= \bar{y_6} \partial y_6 / \partial y_5 = 1 \times 1 = 1 \\
\bar{y_4} &= \bar{y_6} \partial y_6 / \partial y_4 = 1 \times 1 = 1 \\
\bar{y_3} &= \bar{y_5} \partial y_5 / \partial y_3 = 1 \times 1 = 1 \\
\bar{y_2} &= \bar{y_5} \partial y_5 / \partial y_2 = 1 \times 1 = 1 \\
\bar{y_1} &= \bar{y_4} \partial y_4 / \partial y_1 = 1 \times y_0 = 1 \\
\bar{y_0} &= \bar{y_4} \partial y_4 / \partial y_0 = 1 \times y_0 = 2 \\
\bar{y_1} &= \bar{y_1} + \bar{y_3} \partial y_3 / \partial y_1 \\
&= 1 - 1 \times \sin y_1 = 1 - 0.909 \\
&= 0.091 \\
\bar{y_0} &= \bar{y_0} + \bar{y_2} \partial y_2 / \partial y_0 \\
&= 2 + 1 \times \exp y_0 = 2 + 2.718 \\
&= 4.718
\end{aligned}
$$

$$
\begin{aligned}
\partial F / \partial x_1 &= \bar{x_1} = \bar{y_0} = 4.718 \\
\partial F / \partial x_2 &= \bar{x_2} = \bar{y_1} = 0.091
\end{aligned}
$$

For understanding better the algorithm, let's describe how is calculated the former example:

First, the Forward Primal Trace is calculated, starting from the values of x_1, x_2, passing through the calculation of the elementary functions, and ending with the value of F.

Then, we start from the output, i.e we use the fact that $\partial F / \partial F = 1$ and $y_6 = F(x_1, x_2)$. So, the first derivative is calculated.

To compute the following derivatives, we go to the Forward Primal Trace and observe what variables are used to calculate y_6. In this case, those variables are y_5 and y_4.

We take one of those variables, let's start with y_5, and calculate $\bar{y_5} = \partial F / \partial y_5$. To do this, we have to observe that we could see y_6 as a function of y_5 (and maybe other variables) or look it in the "adjoint" way, which is to see y_5 as a function of y_6 (and maybe other variables). Taking that into account, we can calculate $\bar{y_5}$ as $\bar{y_6}\partial y_6/\partial y_5$ by applying the chain rule. And continue doing this to each variable on the right side term of each step in the Forward Primal Trace.

To include the influence of the other variables, it is important to save the value of each derivative calculated. For example, y_1 appears in the definition of y_4 so we calculate $\bar{y_1} = \bar{y_4}\partial y_4/\partial y_1$, but also appears in the definition of y_3, so in this case we need to sum both contributions, as it is done after when we do $\bar{y_1} = \bar{y_1} + \bar{y_3}\partial y_3/\partial y_1$.

By looking at both examples, the forward and the reverse accumulation, it is clear that one flow of the reverse method is more complex than one flow of the forward method, but it calculates both $\partial F/\partial x_1$ and $\partial F/\partial x_2$ at the same time, while the forward method will require to rerun the method again to calculate $\partial F/\partial x_2$.

Backpropagation

The backpropagation algorithm is how the gradient is calculated to train a multi-layer neural network. Once the gradient is known, it could be used in any iterative optimization algorithm to search for the values (in the weights and bias of every layer) that minimize the loss function (The loss function used in this work is specified in section 3.1.3.

The basic idea from which the algorithm is based in the calculation of the reversed derivative trace of the AutoDiff as it was explained before. Here the parameters are the weights of the ANNs and the traversing is done through the layers from the output to the input using the information given by the loss function -L-. As in regular optimization algorithms, the gradient is composed of the partial derivatives of the loss function with respect to the vector of parameters.

To summarize, the backpropagation algorithm will start by calculating the gradient of the loss function with respect to the output of the ANN last layer (N layer). Then, this gradient is used to calculate the gradient with respect to the last layer activation functions. After this, the algorithm, again, calculate the gradient with respect to the output of the $N - 1$ layer using the gradient calculated with respect to the activation function. Then, this continues until the algorithm ends, traversing from the output to the input all the layers in the ANN.

A more explicit description of the algorithm could be found already in textbooks like [20].

3.1.3. Description of the Methodology

In this section is presented, in a more detailed way, how to use the tools introduced before to solve PINNs, specifically how to solve the advection and the diffusion problem. First, it is done in one dimension in order to show more explicitly how the algorithm works, and then both are solved in a non-trivial geometry (a star) in a two-dimensional space, to show the real utility of the technique.

First, it is necessary to select the collocation points that are going to be used in the PINN method. To do this, it is necessary to define the domain in which the problem is going to be solved, once the domain is defined it should be trivial to select the N collocation points that will be used to solve the PINN.

After that, it is important to define both D and G, that are the smooth distance and boundary functions, respectively. For the one dimensional problems, G is not a problem since is just one value for the advection problem and two for the diffusion, so this could be defined directly. Also, for the advection problem D is also trivial since d is just a straight line. For the diffusion problem, in the other hand, since the distance is defined as in 1.4, d is not smooth, therefore is necessary to train a small ANN to calculate D as a smooth version of d.

One of the major advantages of using PINNs is that its resources could be easily escalated to greater dimensions, and this could be seen in the smooth distance function, as it is shown next where is defined a general pipeline where the only thing that is going to change for the two-dimensional version is the number of perceptron in the ANN layers.

Smooth Distance Function

To define a general pipeline to calculate the Smooth Distance Function, D, first let's define a general distance function that calculate the distance of an arbitrary point to the closest point in the boundary. Next, it is shown how this function is being defined:

```python
#Computation of non-smooth distance function d:
def d(xs, xbs):
  #xs: List of collocation points
  #xbs: List of boundary conditions
  xs, xbs = np.array(xs), np.array(xbs)
  ds = [min([np.linalg.norm(x - xb) for xb in xbs]) for x in xs]
  return ds
```

Here `np` is the reference name of NumPy library, which is used to calculate the norm. In this way, the distance is calculated as the l2-norm, no matter what the dimension the problem is.

After having the general distance function, it is necessary to build the dataset that is going to be used to train the small ANN whose output will be D. To do this, the library `random` is used to randomly select n points from the N collocation points. Then, the distance is calculated for these points, and, these pairs are used as labeled data to train a regular ANN that is defined as follows:

```python
def build_model_distance(NUM_PERCEPTRONS):
  model_distance=tf.keras.Sequential([
    tf.keras.layers.Dense(NUM_PERCEPTRONS, activation='sigmoid',
                            input_shape=[None, 1], dtype='float64'
                            ),
    tf.keras.layers.Dense(NUM_PERCEPTRONS,
                            activation='sigmoid'
                            ),
    tf.keras.layers.Dense(1,
                            dtype='float64'
                            )
  ])

  optimizer = Adam(learning_rate=0.01, beta_1=0.9, beta_2=0.99)#tf.
                                                   keras.optimizers.RMSprop(0.001
                                                   )

  model_distance.compile(loss=custom_loss,
                     optimizer=optimizer,
                     metrics=['mae', 'mse'])
  return model_distance
```

Using this function, it is possible to build a general (general for this work since it is not required more than 2 hidden layers) model to calculate the smooth distance function. For the one dimensional problem, the model is defined with `NUM_PERCEPTRONS` = 10 and for the two-dimensional one with `NUM_PERCEPTRONS` = 20.

An important remark in this point is that the sigmoid [1] is used as the activation function. This is because the smoothness of the function defined by the model depends on the smoothness of the activation function, so if another non-smooth function, like the ReLU for example, is used as the activation function the result is not smooth and therefore is useless for the purposes of this work.

Then, to train the model, just run the built-in method `train` passing the selected points as the x values and the distances calculated using d as the y values. In this work, the train was done in 1500 epochs.

Smooth Boundary Function

Although for one dimensional problems the construction of the G function is trivial, in general this is a fundamental part of the PINN method, and is done similarly as is done for the construction of the smooth distance function, but in this case the values of $g(x)$ are already known since the method is used to solve a boundary value problem.

[1]Sigmoid function:

$$S(x) = \frac{1}{1 + \exp(-x)}$$

So, to define the smooth distance function, just build a model in the same way as is done in `build_model_distance` but now train it using the values of $g(x)$ that are defined in the problem statement.

Advection Problem

First rewrite the advection problem using the expression for the ansatz 1.3:

$$\begin{aligned} L\hat{u} &= \frac{d\hat{u}}{dx} = f(x) \\ L\hat{u} &= \frac{dG(x)}{dx} + D(x)\frac{dy^L}{dx} + y^L\frac{dD(x)}{dx} = f(x) \end{aligned}$$

Observe that the ansatz includes the information about the boundary.

Therefore, using the function $f(x)$ is possible to define a cost function as:

$$C = \sum_{\text{x} \in \text{collocation points}} (L\hat{u} - f(x))^2$$

Finally, using the `GradientTape` method from TensorFlow is possible to calculate the derivative of $D(x)$ as follows:

```
x_variable = tf.Variable(x_train, dtype='float64')
with tf.GradientTape() as g:
    g.watch(x_variable)
    Dx = D(x_variable)
dD_dx = g.gradient(Dx, x_variable)
D_pred = D(x_variable)
```

And, using this code, but replacing D by G . Also replacing D by y^L and including it into the training cycle is possible to calculate $\frac{dy^L}{dx}$ and using these three items the ANN is trained as follows:

```
opt = Adam(learning_rate=0.01, beta_1=0.9, beta_2=0.99)
ff = tf.constant(f(x_variable))
n_train_steps = 10000

for step in range(n_train_steps):
    # we need to convert x to a variable if we want the tape to be
    # able to compute the gradient according to x
    with tf.GradientTape() as model_tape:
        with tf.GradientTape() as y_tape:
            y_tape.watch(x_variable)
            y_pred = model(x_variable)
        dy_dx = y_tape.gradient(y_pred, x_variable)
        y_pred_old = y_pred.numpy()
        lu = dD_dx.numpy()*y_pred + D_pred.numpy()*dy_dx
```

```
            loss = tf.reduce_sum(tf.math.squared_difference(lu, ff))
        grad = model_tape.gradient(loss, model.trainable_variables)
        opt.apply_gradients(zip(grad, model.trainable_variables))
```

After the training is finished, $\hat{u}$ could be calculated as in 1.3 and in this way the solution of the BVP is approximated.

Diffusion Problem

As it was done for the advection problem, first start by rewriting the diffusion problem using the expression for the ansatz 1.3:

$$
\begin{aligned}
L\hat{u} &= \frac{d^2\hat{u}}{dx^2} = f(x) \\
L\hat{u} &= \frac{d^2D(x)}{dx^2}y^L + 2\frac{dD(x)}{dx}\frac{dy}{dx} + D(x)\frac{d^2y}{dx^2} = f(x)
\end{aligned}
$$

So, the only difference now are the second order derivatives terms, to handle this terms it is necessary to use two nested `GradientTape`. In the following code is exemplified how to do it for the smooth distance function, extend it to G is trivial and extend it to y^L is just include it in the training cycle as it was shown for the advection problem:

```
x_variable = tf.Variable(x_train, dtype='float64')

with tf.GradientTape() as d_tape:
    d_tape.watch(x_variable)
    with tf.GradientTape() as d2_tape:
        d2_tape.watch(x_variable)
        y = D(x_variable)
    dD_dx = d2_tape.gradient(y, x_variable)
d2D_dx2 = d_tape.gradient(dD_dx, x_variable)

D_pred = D(x_variable)
```

3.2. Results

In this section, some examples of solutions obtained using the methods explained in this work are presented and compared with the known solutions in order to test how good the approximations are and how the method behaves in non-trivial geometries.

3.2.1. Advection 1D

The ansatz introduced in 1.2.3 required the calculation of the smooth distance function distance. Therefore, before solving the PDE we need to calculate the ANN for $D(x)$.

Next, is plotted both $d(x)$ and $D(x)$:

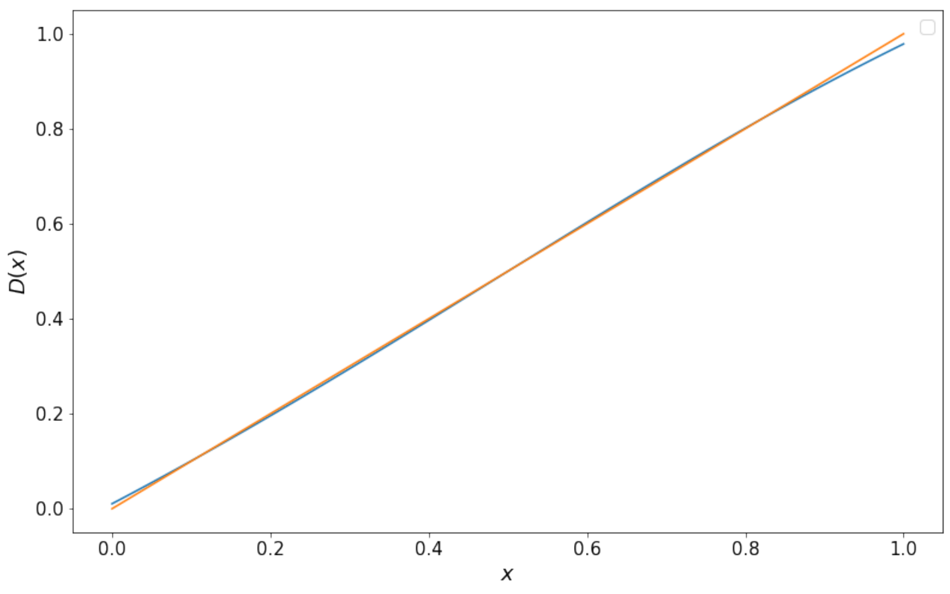


Fig. 3.1. Smooth distance function $D(x)$ compared with the original $d(x)$.

Although, in this case, the distance function is trivial, by calculating it using the ANN we are making sure that this works for arbitrary functions, and approximate the values in a good way, in order to use it for the diffusion problem and the 2D problems.

To check the method for the advection equation in 1D, let's start from a known analytical function and build the problem from that in order to have something to compare the result with.

Taking the analytic solution as follows:

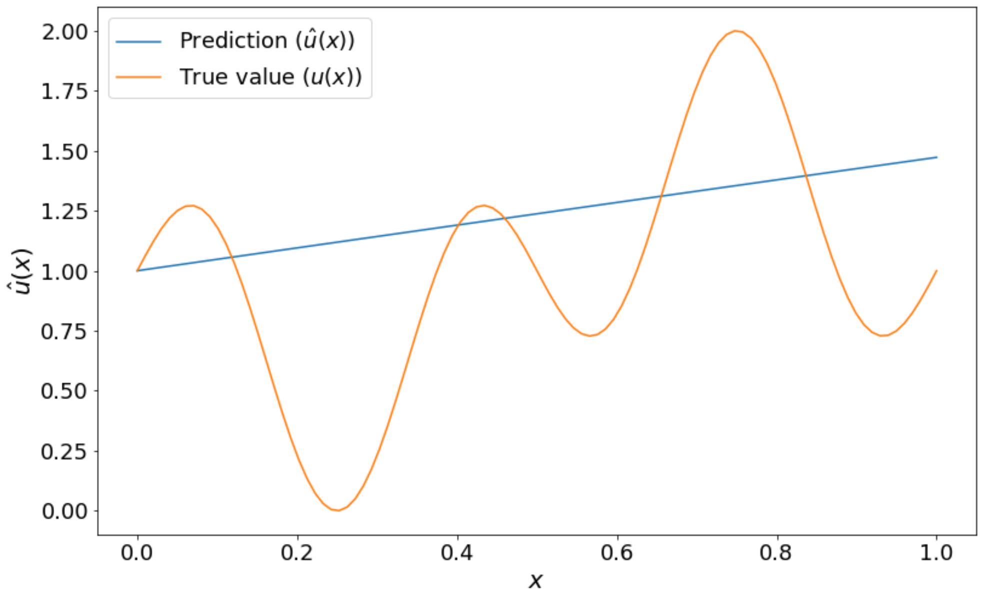
$$u = sin(2\pi x)cos(4\pi x) + 1$$

Therefore, the problem is defined by

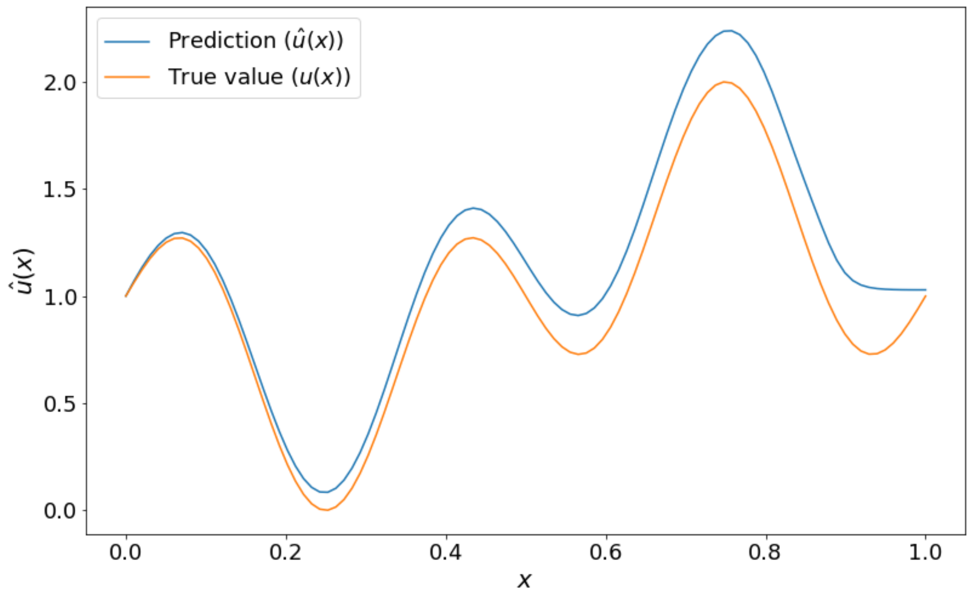
$$\begin{aligned} f(x) &= 2\pi(\cos(2\pi x)\cos(4\pi x) - 2\sin(2\pi x)\sin(4\pi x)) \\ g_0 &= 1 \end{aligned}$$

Using this, and plotting the result for $\hat{u}$ using the PINN as it was trained (each 2000 training steps), the following results are obtained:

$\hat{u}$ for $n_{steps} = 0$:



$\hat{u}$ for $n_{steps} = 2000$:



$\hat{u}$ for $n_{steps} = 50000$:

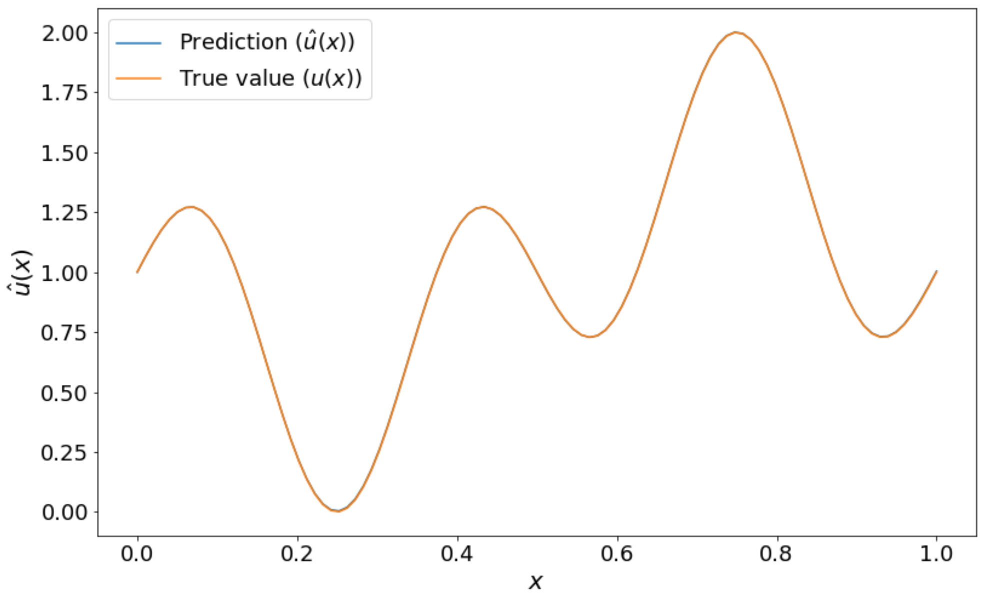


Fig. 3.2. Evolution of the 1D Diffusion problem solution with training

3.2.2. Diffusion 1D

The case of the distance function for the diffusion problem in one dimension is the simplest example of how is used a small ANN to obtain a smooth function from a not smooth one. Next are plotted both $d(x)$ and $D(x)$ to exemplify the situation:

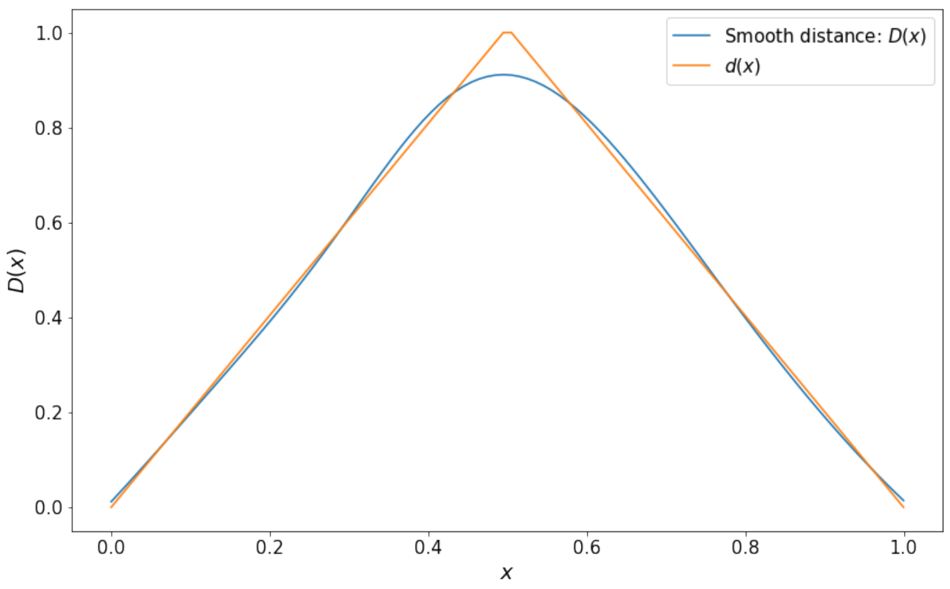


Fig. 3.3. Smooth distance function $D(x)$ compared with the original $d(x)$.

In here is important to remark that the exact values of $D(x)$ are not important as long as we make sure that:

$$|D(x)| < \epsilon, \forall x \in \Gamma$$

This is because the values inside the domain will be "compensated" in the training of the PINN y^L.

For the diffusion equation in 1D, let's take the analytic solution as follows:

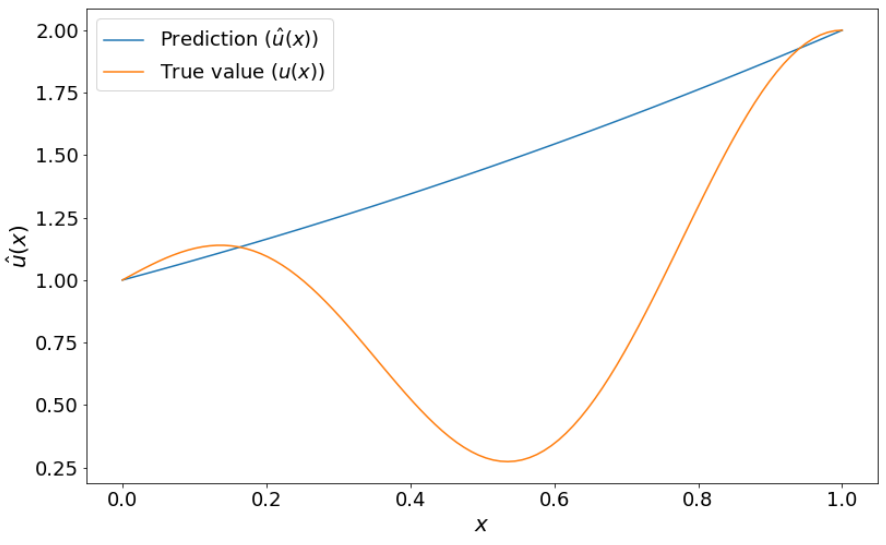
$$u = sin\left(\frac{\pi x}{2}\right)cos(2\pi x) + 1$$

Therefore,

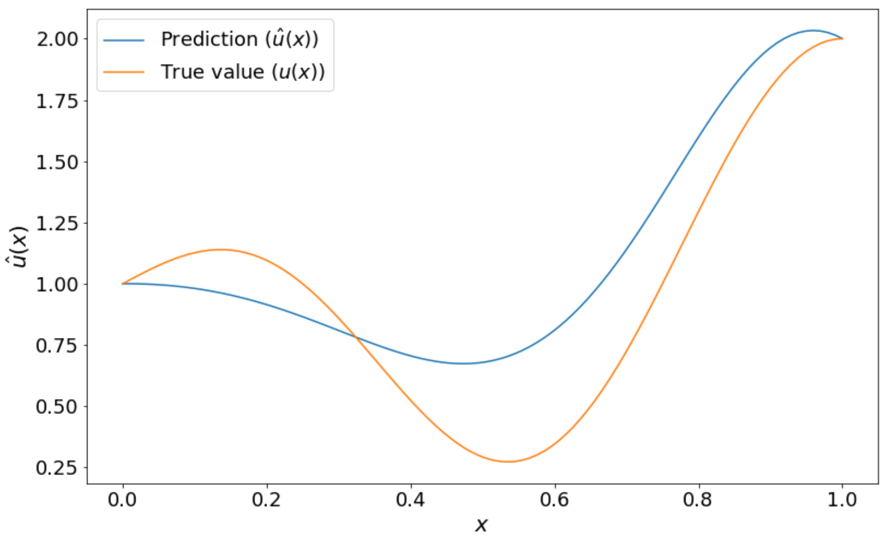
$$f(x) = -\frac{1}{4}\pi^2\left(17cos(2\pi x)sin\left(\frac{\pi x}{2}\right) + 8cos\left(\frac{\pi x}{2}\right)sin(2\pi x)\right)$$
$$g_0 = 1$$

Using this, and plotting the result each 10000 training steps, the following results are obtained:

$\hat{u}$ for $n_{steps} = 0$:



$\hat{u}$ for $n_{steps} = 20000$:



$\hat{u}$ for $n_{steps} = 50000$:

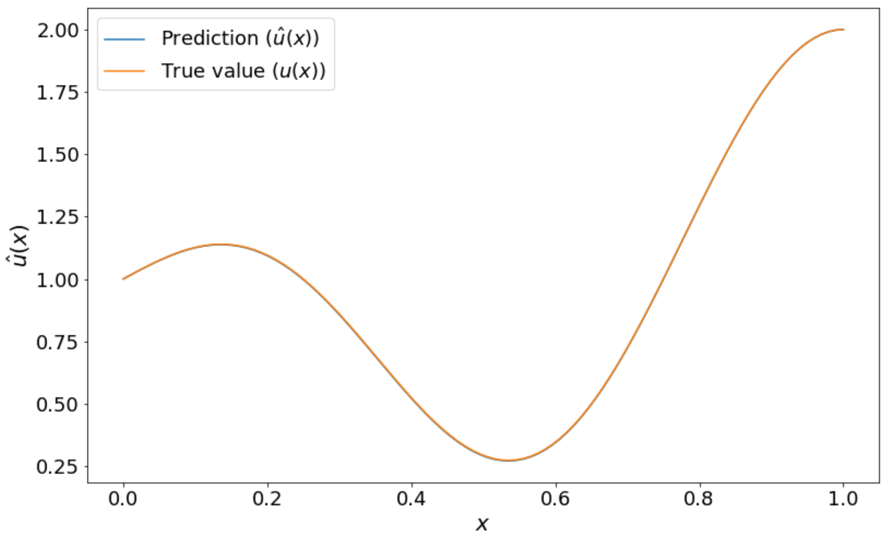


Fig. 3.4. Evolution of the 1D Diffusion problem solution with training

Two-dimensional examples

The most important purpose for this subsection is to show how good this method is to face non-trivial geometries, since this is impossible to show using one-dimensional examples.

For this reason, first, it is important to define the geometry that is going to be used, and this is done with the help of the Python package `shapely.geometry`. Using this package is possible, using the code in Appendix, to define an object, that not only allow to plot a polygon, but also to obtain additional information about it, like checking if an arbitrary point is inside the polygon or generating random boundary points.

Therefore, using this package, a six-pointed star is created as follows:

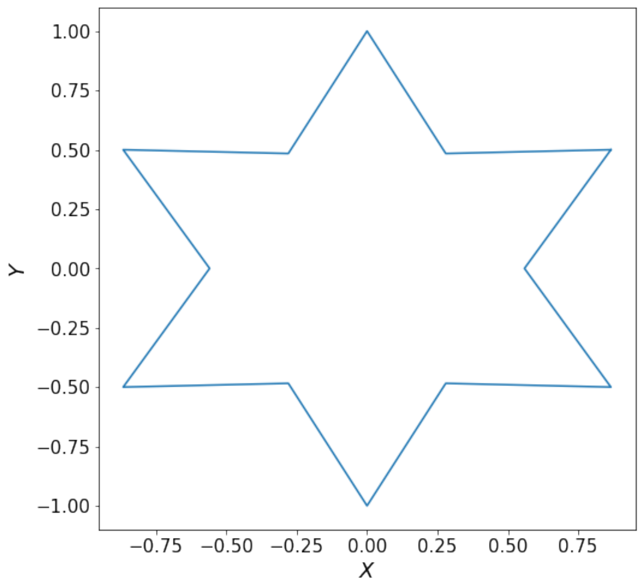


Fig. 3.5. Non-trivial domain in which the PDEs are going to be solved

Then, M boundary points and N internal points are generated using this polygon.

Advection Problem

If the Advection Problem has the following form:

$$Lu = a\frac{\partial u}{\partial x} + b\frac{\partial u}{\partial y} = f \;\; ; \;\; x \in \Omega$$
$$u = g \;\; ; \;\; x \in \Gamma$$

where a, b are the constant advection coefficients and the set $\Gamma \subset \partial\Omega$ is the part of the boundary where boundary conditions should be imposed.

Therefore, if $n = n(x)$ denote the outer unit normal to $\partial\Omega$ at $x \in \partial\Omega$. In such way we have that:

$$\Gamma = \{x \in \partial\Omega : (a, b) \cdot n(x) < 0\}.$$

Therefore, for the advection problem, the M boundary points and N internal points look as follows using $a = 1$ and $b = 1/2$ when they are drawn on the star:

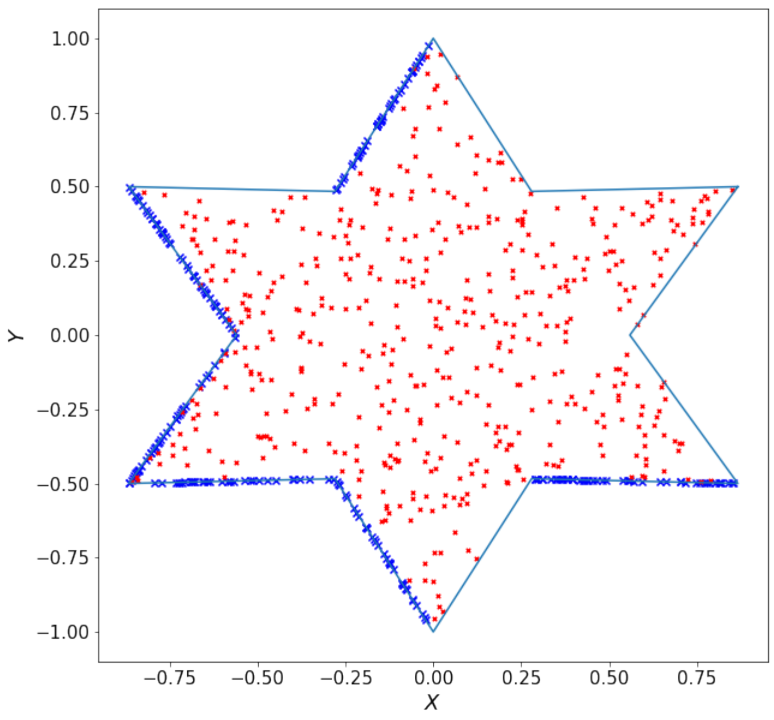


Fig. 3.6. Collocation points for solving the advection problem in 2D

Diffusion Problem

The diffusion problem is given by,

$$Lu = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f \quad ; \quad x \in \Omega$$
$$u = g \quad ; \quad x \in \Gamma$$

Then, the set $\Gamma \subset \partial\Omega$ is all the points in the boundary. In such way we have that:

$$\Gamma = \{x \in \partial\Omega\}.$$

So, for the diffusion problem, the M boundary points and N internal points look as follows when they are drawn on the star:

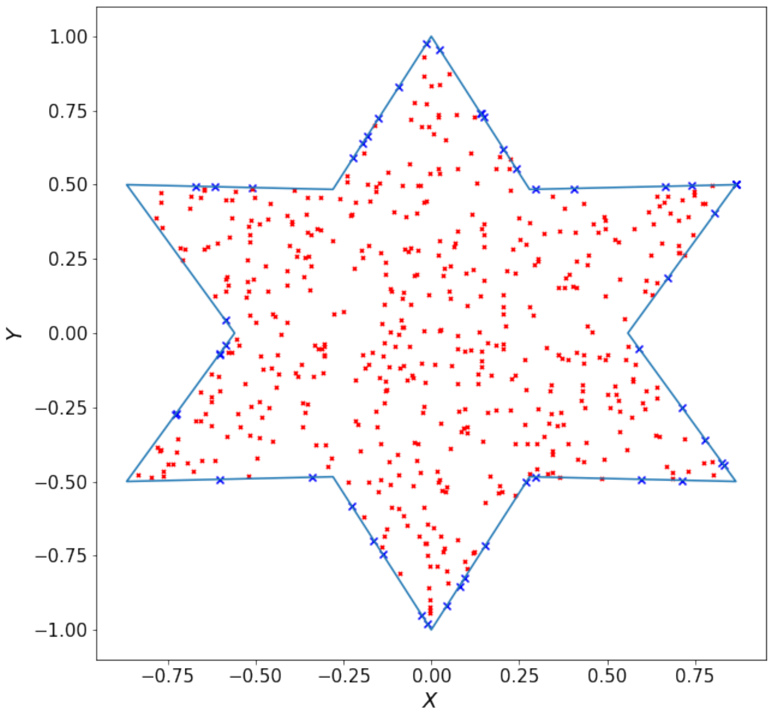


Fig. 3.7. Collocation points for solving the diffusion problem in 2D

Once the collection points are defined, it is possible to solve the PDE using the PINN method. In the next section, we show the results for both the advection and the diffusion problem.

3.2.3. Advection 2D

In this point is where we can see one of the more important advantages of working with ANNs since we can reuse, almost, the exact same code that we used for the calculation of the distance in one dimension. Next, we plot the smooth distance function calculated using the points in Fig 3.6 as a heat map over the domain in which the PDE is going to be solved:

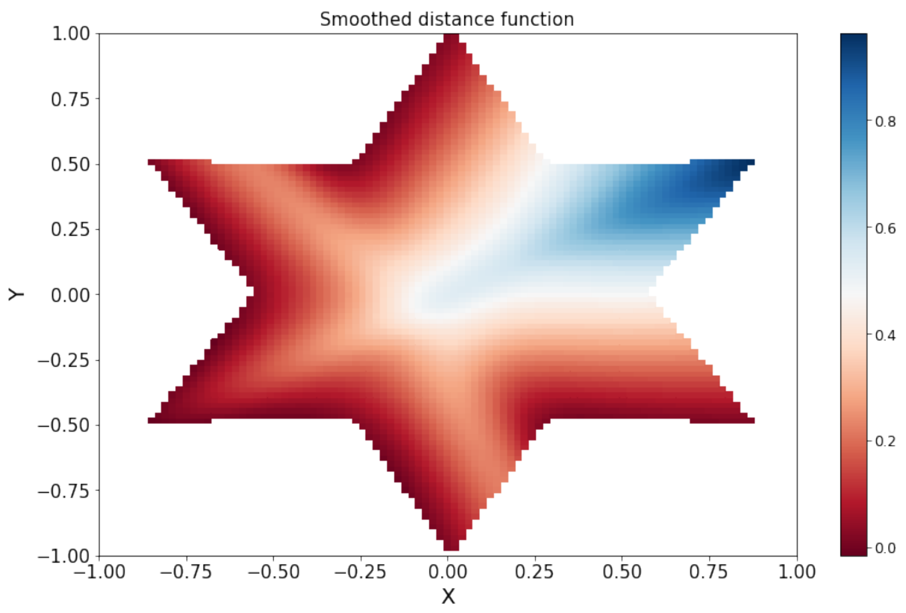


Fig. 3.8. Smooth distance function for solving the advection problem

For solving the advection equation in 2D, let's take the analytic solution as follows:

$$u = \frac{1}{2}\cos(\pi x)\sin(\pi y)$$

Therefore, with $a = 1$ and $b = 1/2$, we have:

$$f(x) = \frac{\pi}{2}\left(-\sin(\pi x)\sin(\pi y) + \frac{1}{2}\cos(\pi x)\cos(\pi y)\right)$$

Using this, and plotting the final result for 100000 training steps, the following result is obtained:

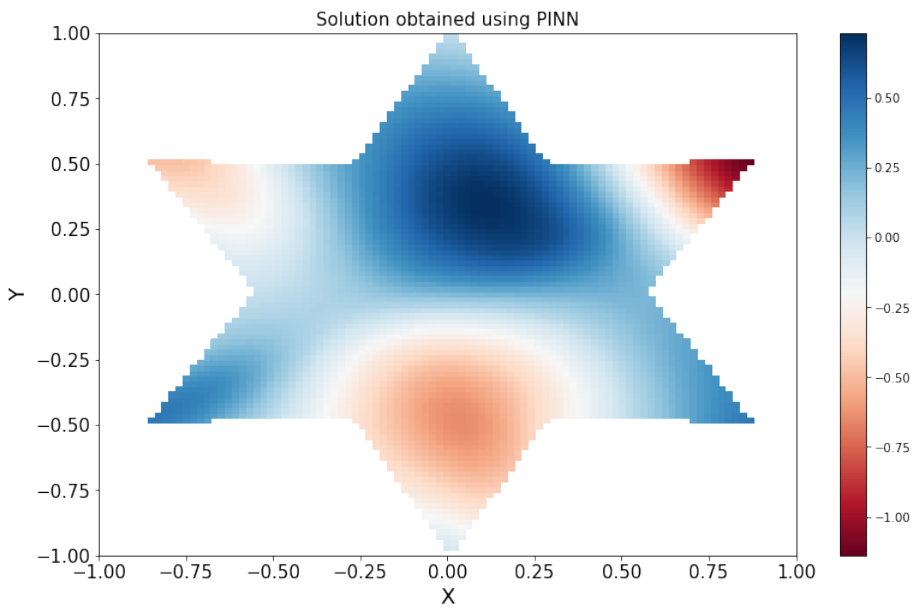


Fig. 3.9. Solution $\hat{u}$ for $n_{steps} = 10000$

In order to and evaluate how good the approximation is, next is shown the real solution:

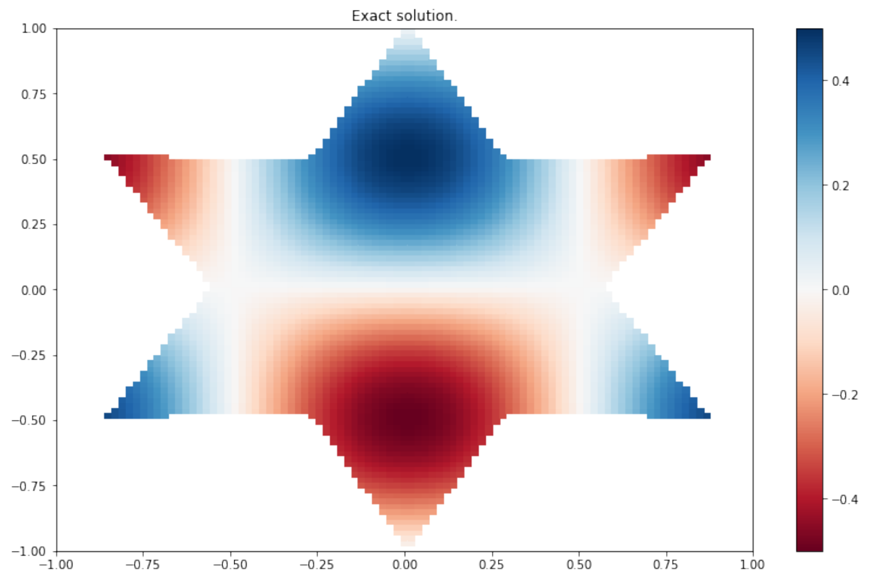


Fig. 3.10. Solution u

The first thing we have to note is that the result is quantitatively wrong. Although the solution resembles the real one, the model seems to be failing to represent the behavior of the solution in the top right star point, that is where the distance functions reach its maximum value. Later, in the analysis section, the difference of the computed and the real solution is plotted in order to better evaluate the differences between the two solutions.

3.2.4. Diffusion 2D

Exactly as it was done for the advection problem, now we calculate the smooth distance for the diffusion problem using the small ANN described in the former section. Next, we plot the smooth distance function calculated using the points in Fig 3.7 as a heat map over the domain in which the PDE is going to be solved:

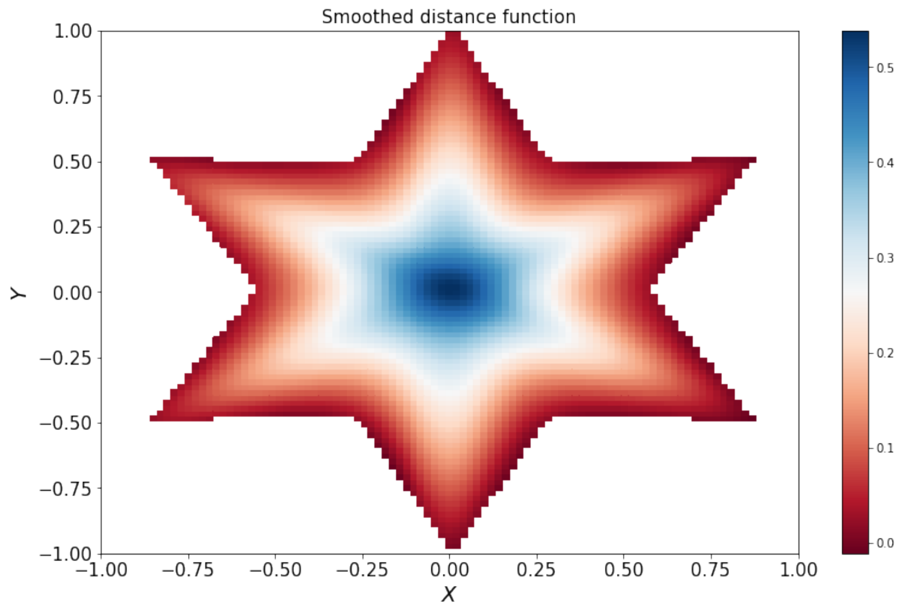


Fig. 3.11. Smooth distance function for solving the diffusion problem

For solving the diffusion equation in 2D, let's take the analytic solution as follows:

$$u = \exp(-(2x^2 + 4y^2)) + 1/2$$

Therefore, we have that the f function will be:

$$f(x) = 4\exp(-2(x^2 + 2y^2))(-1 + 4x^2) + 8\exp(-2(x^2 + 2y^2))(-1 + 8y^2)$$

Using this, and plotting the final result for 100000 training steps, the following result is obtained:

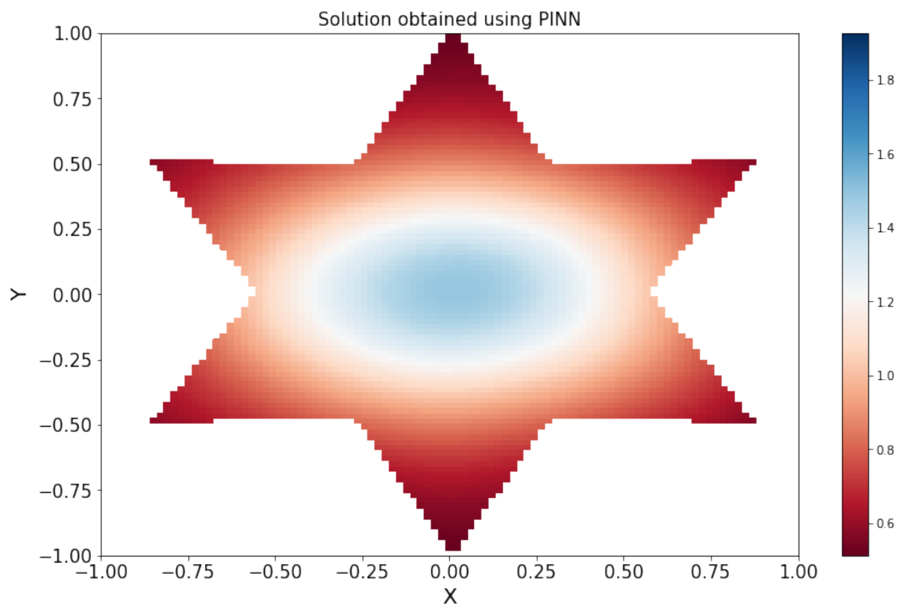


Fig. 3.12. Solution $\hat{u}$ for $n_{steps} = 10000$

Again, next is shown the real solution in order to evaluate qualitatively the solution obtained with the PINN:

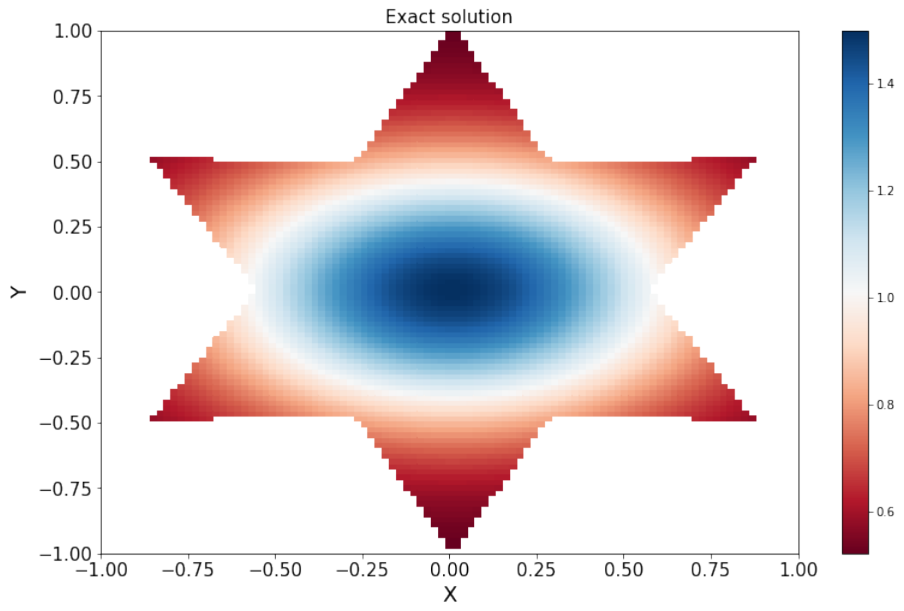


Fig. 3.13. Solution u

In this, the only that we can say is that qualitatively the result is good, but there are certain regions in where the computed solution diverges more. Later, in the analysis section, the difference of the computed and the real solution is plotted in order to evaluate the result obtained in a more qualitative way.

3.3. Analysis

To compare the results obtained with the analytical value, the maximum error between these two values is plotted against the training step for each problem solved.

Advection Problem 1D

First, let's see how the computed solution tends to look more like the real solution while the network is training:

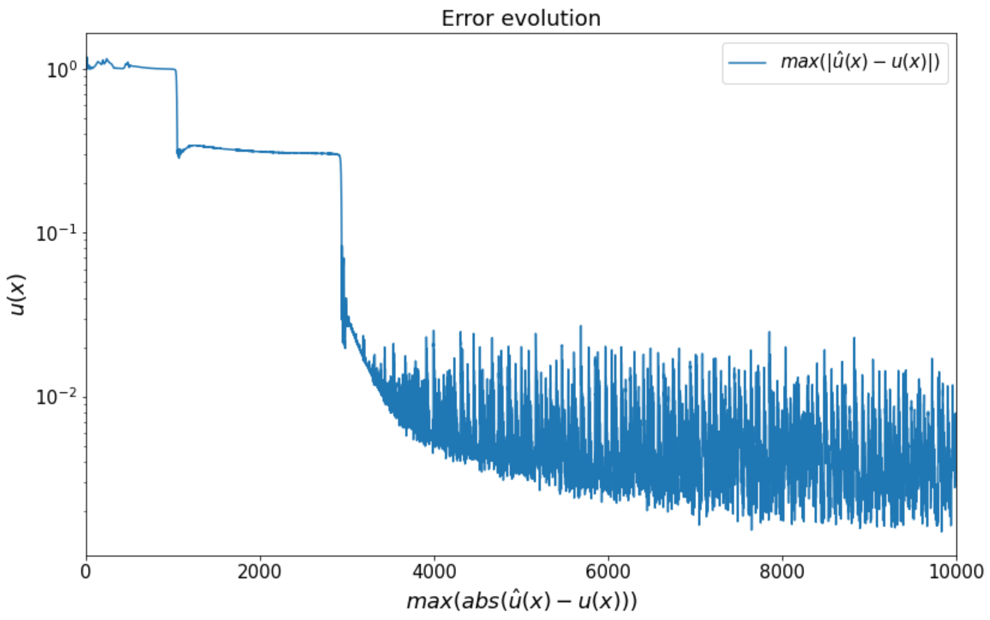


Fig. 3.14. Evolution of the error $max(|\hat{u}(x) - u(x)|)$ with the network training

The first thing that we observe in the former plot is that, at some points, is the value of the gradient of the loss function is so small that the method could consider it to be the minimum, and thus the optimization method would stop, and the solution would not improve anymore. This is fixed by introducing some randomness into the algorithm, in this case that logic is done Adam optimizer that we are using with parameters `learning_rate` $= 0.01, \beta_1 = 0.9, \beta_2 = 0.99$.

Next is plotted the difference between the result obtained $\hat{u}$ and the real solution of the problem u as a function of x:

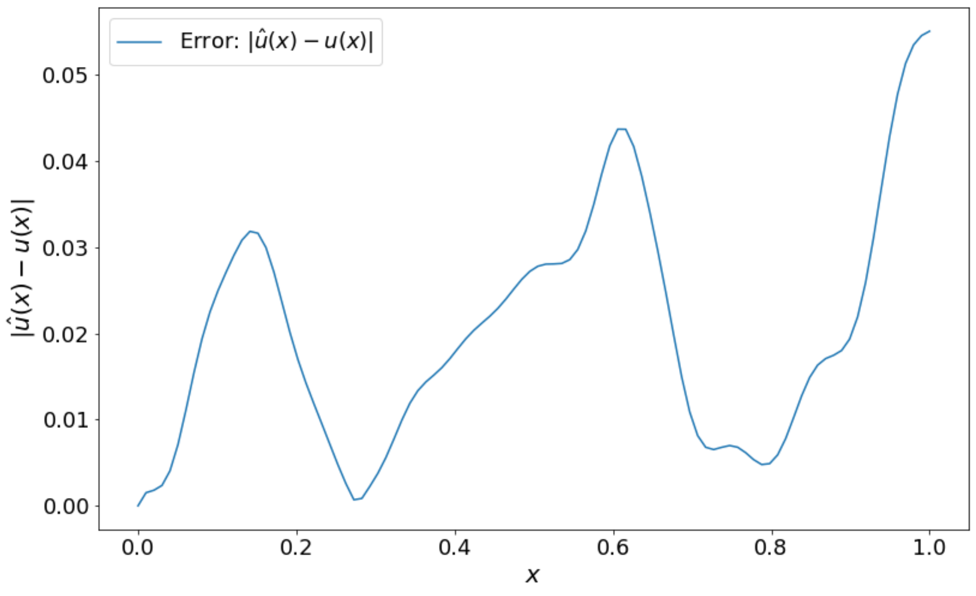


Fig. 3.15. Difference between the computed solution and the real solution ($|\hat{u}(x) - u(x)|$)

As is expected, the error is very small near to the boundary condition, and grow until it stabilize near to 0.07.

Diffusion Problem 1D

As for the advection problem, first we see how the difference between the computed and the real solution differ to each other by plotting its absolute difference while the network is training:

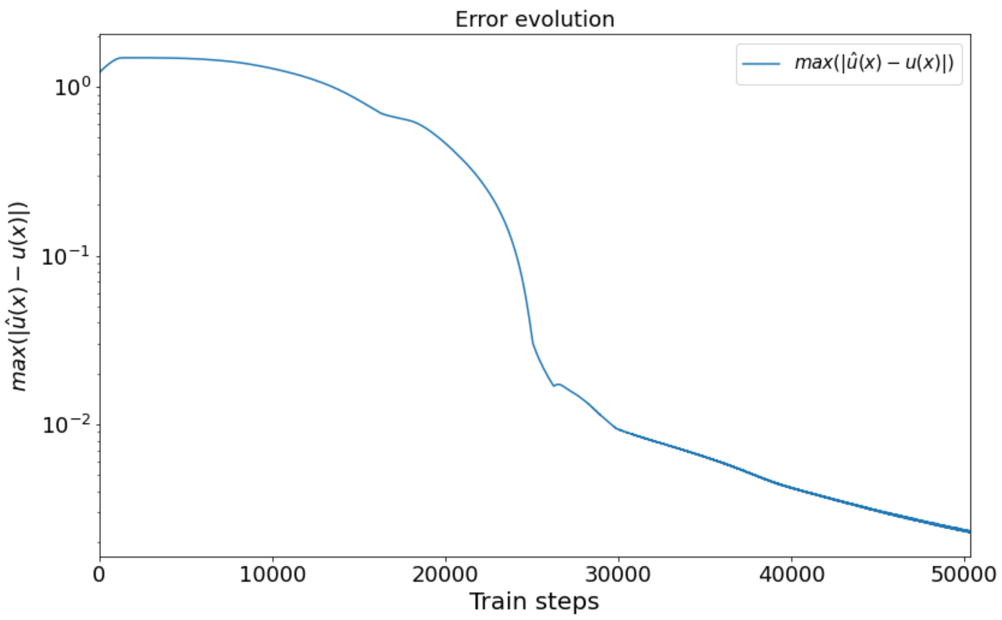


Fig. 3.16. Evolution of the error $max(|\hat{u}(x) - u(x)|)$ with the network training

It is important to remark here that the Diffusion problem required more train steps than the Advection problem. The number of steps used to stop the model is a hyperparameter that is found by observation of the maximum error

Again, the Adam optimizer help us to not get stuck in a local minimum, although, in this case the minimum is reached more smoothly. Also, it is possible to see that the minimum reached is a very stable one, since the maximum error between the real solution and the computed one does not change significantly.

Next is plotted the difference between the calculated PINN $\hat{u}$ and the real solution of the problem u as a function of x:

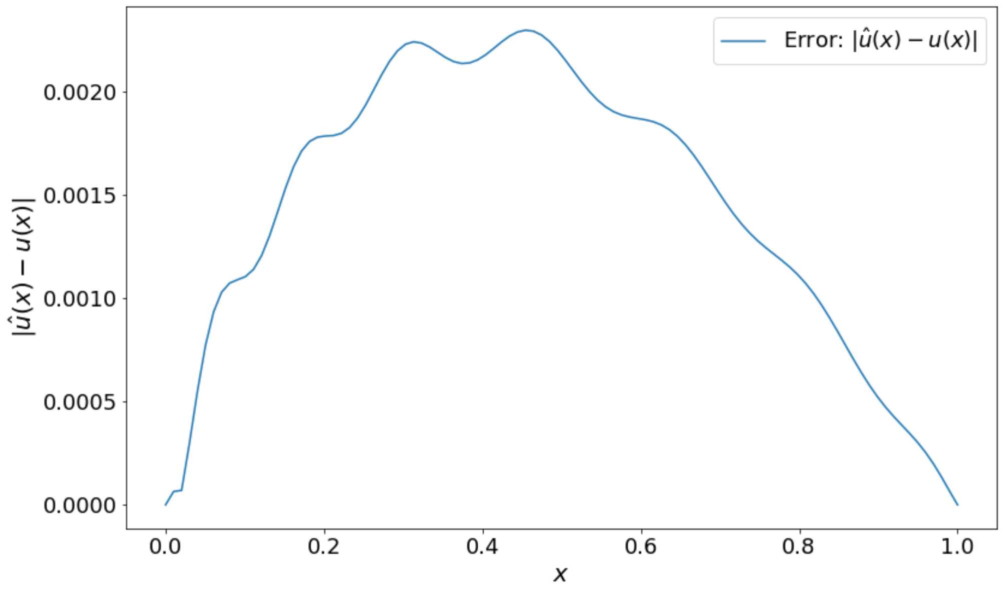


Fig. 3.17. Difference between the computed solution and the real solution ($|\hat{u}(x) - u(x)|$)

To compare the solutions in two dimensions, we will plot the heat map of the difference between the real solution and the computed one.

Advection Problem 2D

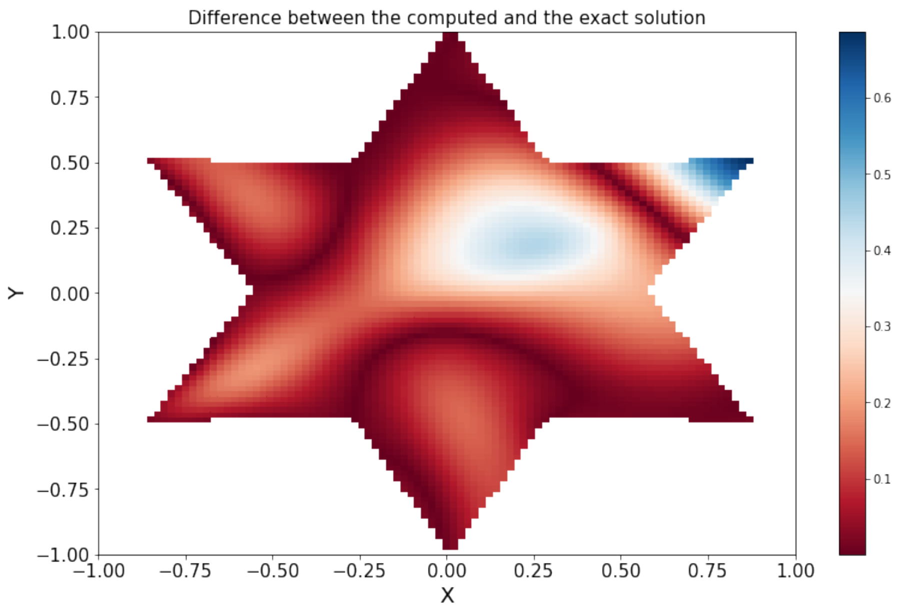


Fig. 3.18. Error ($|\hat{u}(x) - u(x)|$) plotted in the solution domain.

The principal point to remark here is that the maximum error reached by the error is nearly the 60% of the solution value. This is a huge problem, that take us to think even if the method could be used in the future for real world applications.

One of the possible improvements that we suggest, in order to check if the solution get closer to the real one, is to train the model for more time. Although we trained for 2 hours approximately, the resources were the one offer by free for Google Colab, maybe with more resources or more computation time the solution could improve.

Another possible approach could be to experiment with different network architectures. For example, using Convolutional Neural Networks instead of fully connected ones. For this, we imagine that points could be grouped in order to use it as observational points, the exact configuration is out of the scope of this work and is the duty of future attempts to improve the method. This has been an incredible tool when facing multidimensional problems, like image classification and artificial vision, so we guess it could help also in this kind of problems.

Diffusion Problem 2D

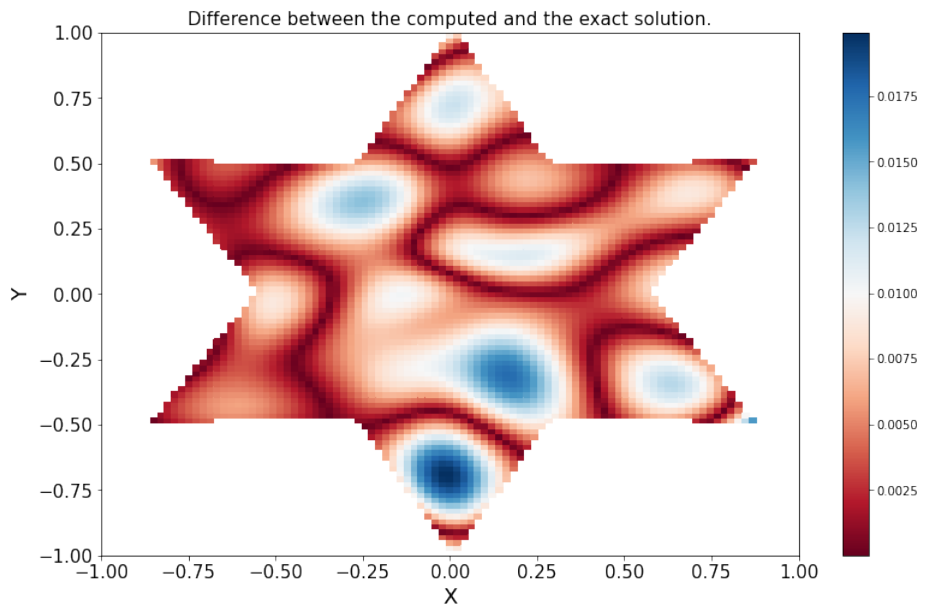


Fig. 3.19. Error ($|\hat{u}(x) - u(x)|$) plotted in the solution domain.

Different from advection problem, for the diffusion problem, as the distance function, the error is also more homogenous around the solution domain. Also, the error range has smaller values in this case.

3.4. Conclusions and contributions

- Our contribution in this work was reviewing the method propose in [1]. This was done by modifying the way the derivatives presented in the loss function were being calculated, introducing the use of AutoDiff. One thing that we could say about this approach is that the code is cleaner, and internally optimized in TensorFlow code. From this, we found the following conclusions for the method, and propose some work to be done in the future.

- The principal conclusion of this work is that, although, for the Advection in 2D problem, the solution was a poor approximation, PINNs look like something that is developing along with deep learning. This problem seems to be related with solving for points that are not close to the subset of the boundary where the boundary conditions are imposed.

  This technique is something that worth to keep working on, and the example of this is the solutions in 1D and the solution for the diffusion problem in 2D. Its benefits could be very big, once the problems for solving in points far from the boundary, for working in domains in which the creation of a mesh is complex to handle.

- Another advantage, that tell us that is important to keep working in solving the issues presented before, is that in this work is shown that the scaling of the dimensionality of the problem is trivial and also is handle with complex geometries. Although this does not imply that the scaling in the computational time is null, we could see from the code that the growth will be because we are solving for more points

  The only parameter that is important to have in mind when doing this scaling is the complexity of the ANN used to find y^L, the extra code used for solving the 2D problems is for defining and generating the geometry and the collocation points, not in the solving of the problem per se.

- As an additional remark, although big resources are available for everyone online, they are still very expensive in the scope of academic or pedagogical environments, and the resources that are commonly available locally or free online are still slow for training fully connected ANNs, even for the small ones.

  This is a clear disadvantage of the method for solving PDEs in regular geometries or even for small dimensions, but it is important to clarify that this complexity does not increase with the dimensionality of the problem or with the complexity of the domain geometry, so it is still an important resource when a problem with these characteristics is to be faced.

BIBLIOGRAPHY

[1] J. Berg and K. Nyström, "A unified deep artificial neural network approach to partial differential equations in complex geometries," *Neurocomputing*, vol. 317, pp. 28–41, 2018. DOI: 10.1016/j.neucom.2018.06.056.

[2] S. Cuomo *et al.*, "Scientific machine learning through physics–informed neural networks: Where we are and what's next," *Journal of Scientific Computing*, vol. 92, no. 3, 2022. DOI: 10.1007/s10915-022-01939-z.

[3] M. W. M. G. Dissanayake and N. Phan-Thien, "Neural-network-based approximations for solving partial differential equations," *Communications in Numerical Methods in Engineering*, vol. 10, no. 3, pp. 195–201, 1994. DOI: 10.1002/cnm.1640100303.

[4] Martín Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[5] G. Fasshauer, *Boundary value problems: Collocation*, 2007.

[6] S. SHARMA, *What the hell is perceptron?* 2022. [Online]. Available: https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53.

[7] S. Gillies *et al.*, *Shapely: Manipulation and analysis of geometric objects*, toblerity.org, 2007–. [Online]. Available: https://github.com/Toblerity/Shapely.

[8] I. Lagaris, A. Likas, and D. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 987–1000, 1998. DOI: 10.1109/72.712178.

[9] I. Lagaris, A. Likas, and D. Papageorgiou, "Neural-network methods for boundary value problems with irregular boundaries," *IEEE Transactions on Neural Networks*, vol. 11, no. 5, pp. 1041–1049, 2000. DOI: 10.1109/72.870037.

[10] E. Bisong, "Google colaboratory," in *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Berkeley, CA: Apress, 2019, pp. 59–64. DOI: 10.1007/978-1-4842-4470-8_7. [Online]. Available: https://doi.org/10.1007/978-1-4842-4470-8_7.

[11] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[12] F. Chollet *et al.* "Keras." (2015), [Online]. Available: https://github.com/fchollet/keras.

[13] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[14] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.5281/zenodo.4030140.

[15] S. Gillies *et al.*, *Shapely: Manipulation and analysis of geometric objects*, 2007–. [Online]. Available: https://github.com/shapely/shapely.

[16] I. Kaplansky, *An Introduction to Differential Algebra*. Paris: Hermann, 1957.

[17] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: A survey,” *arXiv preprint arXiv:1502.05767*, 2015.

[18] A. G. Baydin, B. A. Pearlmutter, and A. A. Radul, “Automatic differentiation in machine learning: A survey,” *CoRR*, vol. abs/1502.05767, 2015. arXiv: 1502.05767. [Online]. Available: http://arxiv.org/abs/1502.05767.

[19] M. Bücker and P. Hovland, *Www.autodiff.org - community portal for automatic differentiation*, 2000. [Online]. Available: https://www.autodiff.org.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016.

ANNEXES 1: CODE FOR SOLVING THE ADVECTION IN 1D

```python
# -*- coding: utf-8 -*-
"""Advection1D.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/
                                                                    1rh0CmwfnVV024x9BTzo6RKQlZW46J2Zm


# Import libraries
"""

import tensorflow as tf
from tensorflow import keras
import tensorflow.keras.backend as kb
from tensorflow.keras import layers, regularizers
from tensorflow.keras.optimizers import Adam
import numpy as np
import keras
import timeit
import matplotlib.pylab as plt

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense
from tensorflow.keras import Input

from google.colab import output

from google.colab import drive
drive.mount('/content/drive')

"""# Constant and function definitions"""

#Seeds
np.random.seed(0)
tf.random.set_seed(0)

#Constants
Nd = 101 #Collocation points in the domain
Nb = 1   #Collocation points in the boundary

#Computation of non-smooth distance function d:
def d(xs, xbs):
  #xs: List of collocation points
  #xbs: List of boundary conditions
  xs, xbs = np.array(xs), np.array(xbs)
  ds = [min([np.linalg.norm(x - xb) for xb in xbs]) for x in xs]
  return ds

def custom_loss(y_actual,y_pred):
    custom_loss=(0.5) * 1 / (Nb + Nd) * (y_actual-y_pred)**2
    return custom_loss

start = timeit.default_timer()

"""# Distance Model"""

N = 10
def build_model_distance(NUM_PERCEPTRONS):
  model_distance=tf.keras.Sequential([
    tf.keras.layers.Dense(NUM_PERCEPTRONS, activation='sigmoid',
                          input_shape=[None, 1], dtype='float64'
                          ),
    tf.keras.layers.Dense(NUM_PERCEPTRONS,
                          activation='sigmoid'
                          ),
    tf.keras.layers.Dense(1,
                          dtype='float64'
                          )
  ])

  optimizer = Adam(learning_rate=0.01, beta_1=0.9, beta_2=0.99)#tf.keras.optimizers.RMSprop(0.001)

  model_distance.compile(loss=custom_loss,
                optimizer=optimizer,
                metrics=['mae', 'mse'])
  return model_distance

"""#Linear advection in 1D

The equation is given by:
\begin{align*}
Lu&=\frac{du}{dx}=f\\
u(0)&=g_0
\end{align*}

And taking the analytic solution as follows:
$$u = sin(2  x)cos(4  x)+1$$

Therefore,
\begin{align*}
f(x) &= 2  (\cos(2\pi x)\cos(4\pi x)-2\sin(2\pi x)\sin(4\pi x))\\
g_0 &= 1
```

```
\end{align*}

So, the ansatz, $\hat{u}=\hat{u}(x;w,b)$, take the form:

$$u(x) = G(x) + D(x)y^L (x; w, b) $$

and the advection problem could be rewrite it as:
\begin{equation*}
\frac{dG(x)}{dx} + D(x)\frac{dy^L}{dx} + y^L\frac{dD(x)}{dx} = f
\end{equation*}
"""

def u(x):
  return np.sin(2*np.pi*x)*np.cos(4*np.pi*x)+1
def f(x):
  return 2*np.pi*(np.cos(2*np.pi*x)*np.cos(4*np.pi*x)-2*np.sin(2*np.
                                    pi*x)*np.sin(4*np.pi*x))

n = 100

xs = np.linspace(0,1,n)
ys = np.array(d(xs,[0]))
ys = ys/np.max(ys)
plt.plot(xs, ys)

N = 10
model_distance = build_model_distance(N)

train_size = 20

#Random set to train of train_size random points
train_set = np.sort(np.random.choice(len(xs)-2, size=train_size,
                                  replace=False))
x_train = xs[train_set+1].reshape(-1,1)
y_train = ys[train_set+1].reshape(-1,1)

model_distance.fit(x_train, y_train, epochs=1500, verbose=False)

d = model_distance.predict(xs.reshape(-1,1)).reshape(-1)
plt.plot(xs,d)

"""## Using a derivative inside a loss function"""

#Define the model

inputs = Input(shape=(1,))
x = Dense(10, 'sigmoid', dtype='float64')(inputs)
x = Dense(10, 'sigmoid')(x)
x = Dense(10, 'sigmoid')(x)
y = Dense(1, dtype='float64')(x)
model = Model(inputs=inputs, outputs=y)
```

```python

tf.keras.utils.plot_model(model,show_shapes=True)

n = 100

def u(x):
  return np.sin(2*np.pi*x)*np.cos(4*np.pi*x)+1
def f(x):
  return 2*np.pi*(np.cos(2*np.pi*x)*np.cos(4*np.pi*x)-2*np.sin(2*np.pi*x)*np.sin(4*np.pi*x))
def D(x):
  return model_distance(x)
def G(x):
  return 1

opt = Adam(learning_rate=0.01, beta_1=0.9, beta_2=0.99)

num_samples = 100

#Random set to train of train_size random points
x_train = np.linspace(0.01,1-0.001,num_samples)
y_train = f(x_train)

# using the high level tf.data API for data handling
x_train = tf.reshape(x_train,(-1,1))
dataset = tf.data.Dataset.from_tensor_slices((x_train,y_train)).batch(100)

xs = np.linspace(0,1,num_samples) # puntos donde evaluo el modelo, distintos de los de colocaci n.

# we need to convert x to a variable if we want the tape to be
# able to compute the gradient according to x

x_variable = tf.Variable(x_train, dtype='float64')
ff = tf.constant(f(x_variable))

with tf.GradientTape() as g:
    g.watch(x_variable)
    Dx = D(x_variable)
dD_dx = g.gradient(Dx, x_variable)

D_pred = D(x_variable)

n_train_steps = 500000

v_loss  = np.empty((n_train_steps,0))
v_error = np.empty((n_train_steps,0))
y_pred_old = np.zeros_like(x_variable)
for step in range(n_train_steps):
    with tf.GradientTape() as model_tape:
        with tf.GradientTape() as y_tape:
            y_tape.watch(x_variable)
            y_pred = model(x_variable)
        dy_dx = y_tape.gradient(y_pred, x_variable)
        y_pred_old = y_pred.numpy()
        lu = dD_dx.numpy()*y_pred + D_pred.numpy()*dy_dx
        loss = tf.reduce_sum(tf.math.squared_difference(lu, ff))
        v_loss = np.append(v_loss,loss)
    grad = model_tape.gradient(loss, model.trainable_variables)
    opt.apply_gradients(zip(grad, model.trainable_variables))

    Yl = model.predict(xs.reshape(-1,1)).reshape(-1)
    Ds = [D(tf.constant([[x]])).numpy()[0][0][0] for x in xs]
    u_hat = G(xs)+Ds*Yl
    error = np.max(np.abs(u_hat-u(xs)))
    v_error = np.append(v_error, error)
    if step%10000==0:
        output.clear()
        print(f"Step {step}: loss={loss.numpy()}")
        print(f"Step {step}: Error={np.max(np.abs(u_hat-u(xs)))}")

        fig, ([ax0, ax1, ax2], [ax3, ax4, ax5]) = plt.subplots(nrows=2, ncols=3, sharex=False,
                                    figsize=(24, 10))
        ax0.plot(xs, u_hat, label="Prediction ($\hat{u}(x)$)")
        ax0.plot(xs, u(xs), label="True value ($u(x)$)")
        ax0.set_xlim(0, 1)
        ax0.legend()

        ax1.semilogy(v_loss, label="Loss function")
        ax1.set_xlim(0, v_loss.shape[0])
        ax1.legend()

        ax2.plot(x_variable.numpy(), lu, label="Lu(x)")
        ax2.plot(x_variable.numpy(), ff, label="f(x)")
        ax2.set_xlim(0, 1)
        ax2.legend()

        ax3.plot(x_train,y_pred, label="y_pred")
        ax3.plot(x_variable[1:-1],(u(x_variable[1:-1])-G(x_variable[1:-1]))/tf.reshape(D(x_variable[1:-1]), [-1, 1]), label="y_exact")
        ax3.set_xlim(0, 1)
        ax3.legend()

        ax4.plot(x_variable.numpy(), dy_dx.numpy(), label="y_pred_x")
        ax4.legend()

        ax5.semilogy(v_error, label=r"$max(|\hat{u}(x) - u(x)|)$")
        ax5.set_xlim(0, v_error.shape[0])
        ax5.legend()
        plt.savefig(f"/content/drive/MyDrive/plots_TFM/advection_1D_{step}.png")
        plt.show()

"""Using the Relu/Selu activation function, which is not differentiable, the neural network is not able to fit the solution of the differential equation.
Tanh works but not is as precise as work with sigmoid function.

### Saving the model
"""

model.summary()

model.save(f"/content/drive/MyDrive/models_TFM/advection_model_{num_samples}_samples")

"""### Loading the model"""

n = 100
num_samples = 200000
def u(x):
  return np.sin(2*np.pi*x)*np.cos(4*np.pi*x)+1
def f(x):
  return 2*np.pi*(np.cos(2*np.pi*x)*np.cos(4*np.pi*x)-2*np.sin(2*np.pi*x)*np.sin(4*np.pi*x))
def D(x):
  #return model_light.predict(x.numpy().reshape(-1,1)).reshape(-1)
  return x
def G(x):
  return 1

model_path = f"/content/drive/MyDrive/models_TFM/advection_model_{num_samples}_samples"
reconstructed_model = keras.models.load_model(model_path)

reconstructed_model.summary()

xs = np.linspace(0,1,n)
Yl = reconstructed_model.predict(xs.reshape(-1,1)).reshape(-1)
u_hat = G(xs)+D(xs)*Yl
plt.figure(figsize=(13,8))
plt.plot(xs, u_hat, label="Prediction ($\hat{u}(x)$)")
plt.plot(xs, u(xs), label="True value ($u(x)$)")
plt.legend()

```

```python
xs = np.linspace(0,1,n)

plt.figure(figsize=(13,8))
plt.plot(xs, abs(u_hat-u(xs)), label="Error: |$\hat{u}(x)-u(x)$|")
plt.legend()

stop = timeit.default_timer()
print('Time: ', stop - start)

"""Time using CPU:  2568.511007819001"""
```

ANNEXES 2: CODE FOR SOLVING THE DIFFUSION IN 1D

```python
# -*- coding: utf-8 -*-
"""Diffusion1D.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1VDrE-PvwMXyACpzrH-
                                    D0Un6rDNCeK4qQ

# Import libraries
"""

import tensorflow as tf
from tensorflow import keras
import tensorflow.keras.backend as kb
from tensorflow.keras import layers, regularizers
from tensorflow.keras.optimizers import Adam
import numpy as np
import keras
import timeit
import matplotlib.pylab as plt

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense
from tensorflow.keras import Input

from google.colab import output

from google.colab import drive
drive.mount('/content/drive')

"""# Constant and function definitions"""

#Seeds
np.random.seed(0)
tf.random.set_seed(0)

#Constants
Nd = 101 #Collocation points in the domain
Nb = 1   #Collocation points in the boundary

#Computation of non-smooth distance function d:
def d(xs, xbs):
  #xs: List of collocation points
  #xbs: List of boundary conditions
  xs, xbs = np.array(xs), np.array(xbs)
  ds = [min([np.linalg.norm(x - xb) for xb in xbs]) for x in xs]
  return ds

def custom_loss(y_actual,y_pred):
    custom_loss=(0.5) * 1 / (Nb + Nd) * (y_actual-y_pred)**2
    return custom_loss

start = timeit.default_timer()

"""# Distance Model"""

N = 10
def build_model_distance(NUM_PERCEPTRONS):
  model_distance=tf.keras.Sequential([
    tf.keras.layers.Dense(NUM_PERCEPTRONS, activation='sigmoid',
                          input_shape=[None, 1], dtype='float64'
                          ),
    tf.keras.layers.Dense(NUM_PERCEPTRONS,
                          activation='sigmoid'
                          ),
    tf.keras.layers.Dense(1,
                          dtype='float64'
                          )
  ])

  optimizer = Adam(learning_rate=0.01, beta_1=0.9, beta_2=0.99)#tf.keras.optimizers.RMSprop(0.001)

  model_distance.compile(loss=custom_loss,
                optimizer=optimizer,
                metrics=['mae', 'mse'])
  return model_distance

"""#Linear diffusion in 1D

The equation is given by:
\begin{align*}
Lu&=\frac{d^2u}{dx^2}=f\\
u(0)&=g_0
\end{align*}

And taking the analytic solution as follows:
$$u = sin\left(\frac{\pi x}{2}\right)cos(2  x)+1$$

Therefore,
\begin{align*}
f(x) &= -\frac{1}{4}  ^2\left(17cos(2  x)sin\left(\frac{ x }{2}\right)+8cos\left(\frac{ x }{2}\right)sin(2  x)\right)\\
g_0 &= 1
\end{align*}

So, the ansatz, $\hat{u}=\hat{u}(x;w,b)$, take the form:

$$u(x) = G(x) + D(x)y^L (x; w, b) $$

So, the problem becames:

$$\frac{d^2D(x)}{dx^2}y^L + 2\frac{dD(x)}{dx}\frac{dy}{dx} + D(x)\
                                        frac{d^2y}{dx^2} = f(x)$$
"""

def u(x):
  return np.sin(2*np.pi*x)*np.cos(4*np.pi*x)+1
def f(x):
  return 2*np.pi*(np.cos(2*np.pi*x)*np.cos(4*np.pi*x)-2*np.sin(2*np.
                                          pi*x)*np.sin(4*np.pi*x))

n = 100

xs = np.linspace(0,1,n)
ys = np.array(d(xs,[0, 1]))
ys = ys/np.max(ys)
plt.plot(xs, ys)

N = 10
model_distance = build_model_distance(N)

train_size = 20

#Random set to train of train_size random points
train_set = np.sort(np.random.choice(len(xs)-2, size=train_size,
                                    replace=False))
x_train = xs[train_set+1].reshape(-1,1)
y_train = ys[train_set+1].reshape(-1,1)

model_distance.fit(x_train, y_train, epochs=1500, verbose=False)

d = model_distance.predict(xs.reshape(-1,1)).reshape(-1)
plt.plot(xs,d)

"""## Using a derivative inside a loss function"""

#Define the model

inputs = Input(shape=(1,))
x = Dense(10, 'sigmoid', dtype='float64')(inputs)
x = Dense(10, 'sigmoid')(x)
y = Dense(1, dtype='float64')(x)
model = Model(inputs=inputs, outputs=y)
```

```python

tf.keras.utils.plot_model(model,show_shapes=True)

def u(x):
  return np.sin(np.pi*x/2)*np.cos(2*np.pi*x)+1
def f(x):
  return -(1/4)*np.pi**2*(17*np.cos(2*np.pi*x)*np.sin(np.pi*x/2)+8*np.cos(np.pi*x/2)*np.sin(2*np.pi*x))
def D(x):
  return model_distance(x)
def G(x):
  return 1

opt = Adam(learning_rate=0.0001, beta_1=0.9, beta_2=0.99)

num_samples = 100

#Random set to train of train_size random points
x_train = np.linspace(0.01,1-0.001,num_samples)
y_train = f(x_train)

# using the high level tf.data API for data handling
x_train = tf.reshape(x_train,(-1,1))
dataset = tf.data.Dataset.from_tensor_slices((x_train,y_train)).batch(100)

xs = np.linspace(0,1,num_samples) # puntos donde evaluo el modelo, distintos de los de colocaci n.

# we need to convert x to a variable if we want the tape to be
# able to compute the gradient according to x

x_variable = tf.Variable(x_train, dtype='float64')
ff = tf.constant(f(x_variable))

with tf.GradientTape() as D_tape:
    D_tape.watch(x_variable)
    with tf.GradientTape() as D2_tape:
        D2_tape.watch(x_variable)
        y = D(x_variable)
    dD_dx = D2_tape.gradient(y, x_variable)
d2D_dx2 = D_tape.gradient(dD_dx, x_variable)
D_pred = D(x_variable)

xs = np.linspace(0,1,num_samples) # Points where the model is evaluated, different to the collocation points.

# we need to convert x to a variable if we want the tape to be
# able to compute the gradient according to x
```

```python

n_train_steps = 500000

v_loss  = np.empty((n_train_steps,0))
v_error = np.empty((n_train_steps,0))
y_pred_old = np.zeros_like(x_variable)
for step in range(n_train_steps):
    with tf.GradientTape() as model_tape:
        with tf.GradientTape() as y_tape2:
          with tf.GradientTape() as y_tape1:
              y_tape1.watch(x_variable)
              y_pred = model(x_variable)
          y_tape2.watch(x_variable)
          dy_dx = y_tape1.gradient(y_pred, x_variable)
        d2y_dx2 = y_tape2.gradient(dy_dx, x_variable)
        y_pred_old = y_pred.numpy()
        lu = d2D_dx2.numpy()*y_pred +2*dD_dx.numpy()*dy_dx + D_pred.numpy()*d2y_dx2
        loss = tf.reduce_sum(tf.math.squared_difference(lu, ff))
        v_loss = np.append(v_loss,loss)
    grad = model_tape.gradient(loss, model.trainable_variables)
    opt.apply_gradients(zip(grad, model.trainable_variables))

    Yl = model.predict(xs.reshape(-1,1)).reshape(-1)
    Ds = [D(tf.constant([[x]])).numpy()[0][0][0] for x in xs]
    u_hat = G(xs)+Ds*Yl
    error = np.max(np.abs(u_hat-u(xs)))
    v_error = np.append(v_error, error)
    if step%10000==0:
        output.clear()
        print(f"Step {step}: loss={loss.numpy()}")
        print(f"Step {step}: Error={np.max(np.abs(u_hat-u(xs)))}")

        fig, ([ax0, ax1, ax2], [ax3, ax4, ax5]) = plt.subplots(nrows=2, ncols=3, sharex=False,
                                    figsize=(24, 10))
        ax0.plot(xs, u_hat, label="Prediction ($\hat{u}(x)$)")
        ax0.plot(xs, u(xs), label="True value ($u(x)$)")
        ax0.set_xlim(0, 1)
        ax0.legend()

        ax1.semilogy(v_loss, label="Loss function")
        ax1.set_xlim(0, v_loss.shape[0])
        ax1.legend()

        ax2.plot(x_variable.numpy(), lu, label="Lu(x)")
        ax2.plot(x_variable.numpy(), ff, label="f(x)")
        ax2.set_xlim(0, 1)
        ax2.legend()

        ax3.plot(x_train,y_pred, label="y_pred")
        ax3.plot(x_variable[1:-1],(u(x_variable[1:-1])-G(x_variable[1:-1]))/tf.reshape(D(x_variable[1:-1]), [-1,1]), label="y_exact")
        ax3.set_xlim(0, 1)
        ax3.legend()

        ax4.plot(x_variable.numpy(), dy_dx.numpy(), label="y_pred_x")
        ax4.plot(x_variable.numpy(), d2y_dx2.numpy(), label="y_pred_xx")
        ax4.legend()

        ax5.semilogy(v_error, label="M ximo del error con respecto a la soluci n exacta")
        ax5.set_xlim(0, v_error.shape[0])
        ax5.legend()
        plt.savefig(f"/content/drive/MyDrive/plots_TFM/diffusion_1D_{step}.png")
        plt.show()

"""Using the Relu/Selu activation function, which is not differentiable, the neural network is not able to fit the solution of the differential equation.
Tanh works but not is as precise as work with sigmoid function.

### Saving the model
"""

model.summary()

model.save(f"/content/drive/MyDrive/models_TFM/advection_model_{num_samples}_samples")

"""### Loading the model"""

n = 100

def u(x):
  return np.sin(2*np.pi*x)*np.cos(4*np.pi*x)+1
def f(x):
  return 2*np.pi*(np.cos(2*np.pi*x)*np.cos(4*np.pi*x)-2*np.sin(2*np.pi*x)*np.sin(4*np.pi*x))
def D(x):
  #return model_light.predict(x.numpy().reshape(-1,1)).reshape(-1)
  return x
def G(x):
  return 1
```

```

model_path = f"/content/drive/MyDrive/models_TFM/advection_model_{
                                            num_samples}_samples"
reconstructed_model = keras.models.load_model(model_path)

reconstructed_model.summary()

xs = np.linspace(0,1,n)
Yl = reconstructed_model.predict(xs.reshape(-1,1)).reshape(-1)
u_hat = G(xs)+D(xs)*Yl
plt.figure(figsize=(13,8))
plt.plot(xs, u_hat, label="Prediction ($\hat{u}(x)$)")
plt.plot(xs, u(xs), label="True value ($u(x)$)")
plt.legend()

xs = np.linspace(0,1,n)

plt.figure(figsize=(13,8))
plt.plot(xs, abs(u_hat-u(xs)), label="Error: |$\hat{u}(x)-u(x)$|")
plt.legend()

stop = timeit.default_timer()
print('Time: ', stop - start)

"""Time using CPU:  2568.511007819001"""
```

ANNEXES 3: CODE FOR SOLVING THE ADVECTION IN 2D

```python
# -*- coding: utf-8 -*-
"""Advection2D.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/
                                                    1tHc7eyfwwBFJHGXvPHJE631CIMcppC9b


# Distance function in 2D (Linear advection 2D)
"""

import turtle
import random

import tensorflow as tf
import tensorflow as tf
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense
from tensorflow.keras import Input
from keras.models import load_model

import numpy as np
from shapely.geometry import Point, Polygon
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount('/content/drive')

"""# Constant and function definitions"""

#Seeds
np.random.seed(0)
tf.random.set_seed(0)

#Constants
Nd = 101 #Collocation points in the domain
Nb = 1   #Collocation points in the boundary

#Computation of non-smooth distance function d:
def d(xs, xbs):
  #xs: List of collocation points
  #xbs: List of boundary conditions
  xs, xbs = np.array(xs), np.array(xbs)
  ds = [min([np.linalg.norm(x - xb) for xb in xbs]) for x in xs]
  return ds

def custom_loss(y_actual,y_pred):
    custom_loss=(0.5) * 1 / (Nb + Nd) * (y_actual-y_pred)**2
    return custom_loss

def get_optimizer():
  return tf.keras.optimizers.Adam(lr_schedule)

"""## Define a star and use polygon to check points inside"""

star=[]
r = np.sqrt(0.5**2 + 0.25**2)
R = 1

num_vert = 12
start = np.pi / 6
for n in range(0,12):
  if n % 2:
    rad = r
  else:
    rad = R
  x = rad * np.cos(start + 2 * np.pi * n / num_vert)
  y = rad * np.sin(start + 2 * np.pi * n / num_vert)
  star.append([x,y])

star_poly = Polygon(star)

x,y = star_poly.exterior.xy
fig, ax = plt.subplots(figsize=(8,8))

ax.plot(x,y)

"""## Generate M random points in the boundary"""

def get_random_boundary_point(polygon):
  x,y = star_poly.exterior.xy
  points = [*zip(x,y)]
  random_index = random.randint(0, len(points)-1)
  a, b = np.array(points[random_index]), np.array(points[(
                                             random_index + 1) % len(points
                                             )])
  new_random_point = a + random.random() * (b - a)
  return list(new_random_point)

M = 50
random_boundary_points = [get_random_boundary_point(star_poly) for _
                                           in range(M)]

```

```python
x,y = star_poly.exterior.xy
fig, ax = plt.subplots(figsize=(8,8))

ax.plot(x,y)
ax.scatter(*list(zip(*random_boundary_points)), color="blue", marker
                                          ='x')
plt.show()

"""## Generate N random points in the boundary (Satisfying inflow
                                          condition)"""

def cross(p, q):
  p_x, p_y = p
  q_x, q_y = q
  return p_x*q_y - p_y*q_x

def get_random_boundary_point_inflow(polygon, cond: list):
  x, y = star_poly.exterior.xy
  points = [*zip(x,y)]
  while True:
    random_index = random.randint(0, len(points)-1)
    a, b = np.array(points[random_index]), np.array(points[(
                                              random_index + 1) % len(
                                              points)])
    dx, dy = b - a
    s_x = [dx, dy]
    n_x = [-dy, dx]
    if cross(s_x, n_x) > 0:
      n_x = [dy, -dx]
    if np.dot(n_x, cond) < 0:
      new_random_point = a + random.random() * (b - a)
      return list(new_random_point)

M = 50
random_boundary_points = [get_random_boundary_point_inflow(star_poly
                                          , [1, 1]) for _ in range(M)]

x,y = star_poly.exterior.xy
fig, ax = plt.subplots(figsize=(8,8))

ax.plot(x,y)
ax.scatter(*list(zip(*random_boundary_points)), color="blue", marker
                                          ='x')
plt.show()

"""## Generate N random points inside"""

def get_random_inside_polygon(polygon):
  minx, miny, maxx, maxy = polygon.bounds
  while True:
    p = Point(random.uniform(minx, maxx), random.uniform(miny, maxy))
    if polygon.contains(p):
      x, y = p.x, p.y
      return [x, y]

N = 500
random_inside_points = [get_random_inside_polygon(star_poly) for _ in range(N)]

x,y = star_poly.exterior.xy
fig, ax = plt.subplots(figsize=(8,8))

ax.plot(x,y)
ax.scatter(*list(zip(*random_boundary_points)), color="blue", marker='x')
ax.scatter(*list(zip(*random_inside_points)), color="red", marker='x')
plt.show()

"""# Model for distance function (Linear advection 2D) """

#Define the model

inputs = Input(shape=(2,))
x = Dense(20, 'sigmoid', dtype='float64')(inputs)
x = Dense(20, 'sigmoid')(x)
y = Dense(1, dtype='float64')(x)
model_distance = Model(inputs=inputs, outputs=y)

optimizer = Adam(learning_rate=0.01, beta_1=0.9, beta_2=0.99)
model_distance.compile(loss=custom_loss,
                optimizer=optimizer,
                metrics=['mae', 'mse'])

tf.keras.utils.plot_model(model_distance,show_shapes=True)

"""## Training"""

N = 500
M = 250

random_inside_points = [get_random_inside_polygon(star_poly) for _ in range(N)]
random_boundary_points = [get_random_boundary_point_inflow(star_poly, [1, 1/2]) for _ in range(M)]

#Random set to train of train_size random points in the domain (excluding extremal points)
x_train = np.array(random_inside_points).reshape(-1, 2)
ds = d(x_train, random_boundary_points)
```

```python
y_train = np.array(ds).reshape(-1, 1)

x,y = star_poly.exterior.xy
fig, ax = plt.subplots(figsize=(8,8))

ax.plot(x,y)
ax.scatter(*list(zip(*random_boundary_points)), color="blue", marker
                                        ='x')
ax.scatter(*list(zip(*random_inside_points)), color="red", marker='x
                                        ', s=10)
plt.show()

model_distance.fit(x_train, y_train, epochs=1000, verbose=False)

"""## Plotting distance model"""

@np.vectorize
def distance_inside(x, y, polygon: Polygon):
  distance = model_distance.predict([[x, y]])[0][0] if polygon.
                                        contains(Point([x, y])) else
                                        np.nan
  return distance

# generate 2 2d grids for the x & y bounds
y, x = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))
z = distance_inside(x, y, star_poly)

z_min, z_max = np.nanmin(z), np.nanmax(z)

fig, ax = plt.subplots(figsize=(13,8))

c = ax.pcolormesh(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
ax.set_title('Smoothed distance function')
# set the limits of the plot to the limits of the data
ax.axis([x.min(), x.max(), y.min(), y.max()])
fig.colorbar(c, ax=ax)

plt.show()

"""## Save model distance"""

model_distance.save('/content/drive/MyDrive/models_TFM/
                                        model_distance.h5')

"""## Load model distance"""

model_distance = load_model(
    '/content/drive/MyDrive/models_TFM/model_distance.h5',
    custom_objects={'custom_loss':custom_loss}
    )

```

```python
"""# Model for the smooth extension of the boundary data ($g(x,y)$)
                                    (Linear advection 2D)"""

def u(x, y):
  return 0.5 * np.cos(np.pi * x) * np.sin(np.pi * y)

def g(x, y):
  return u(x, y)

#Define the model

inputs = Input(shape=(2,))
x = Dense(10, 'sigmoid', dtype='float64')(inputs)
x = Dense(10, 'sigmoid')(x)
y = Dense(1, dtype='float64')(x)
model_boundary = Model(inputs=inputs, outputs=y)

optimizer = Adam(learning_rate=0.01, beta_1=0.9, beta_2=0.99)
model_boundary.compile(loss=custom_loss,
                optimizer=optimizer,
                metrics=['mae', 'mse'])

tf.keras.utils.plot_model(model_boundary,show_shapes=True)

x_train = np.array(random_boundary_points).reshape(-1, 2)
y_train = np.array([g(x, y) for x, y in x_train])

model_boundary.fit(x_train, y_train, epochs=1000, verbose=False)

"""## Save model boundary"""

model_boundary.save('/content/drive/MyDrive/models_TFM/
                                    model_boundary.h5')

"""## Load model boundary"""

model_boundary = load_model(
    '/content/drive/MyDrive/models_TFM/model_boundary.h5',
    custom_objects={'custom_loss':custom_loss}
    )

"""# Linear advection in 2D

The equation is given by:
\begin{align*}
Lu&=a\frac{\partial u}{\partial x} + b\frac{\partial u}{\partial y}=
                                    f \;\;;\; x \in \Omega\\
u&=g \;\;;\; x \in \Gamma\\
\end{align*}

```

```
where a, b are the constant advection coefficients. The set $\Gamma \subset \partial \Omega$ is the part of the boundary where boundary conditions should
be imposed. And let $n = n(x)$ denote the outer unit normal to $\partial\Omega$ at $x \in\partial \Omega$. In such way we have that:

$$\Gamma = \{x\in\partial\Omega: (a, b) \cdot n(x) < 0\}.$$

And taking the analytic solution as follows:
$$u = \frac{1}{2}\cos(\pi x)\sin(\pi y)$$

Therefore, with $a = 1$ and $b=1/2$, we have:
\begin{align*}
f(x) &= \frac{\pi}{2}\left(-\sin(\pi x)\sin(\pi y) + \frac{1}{2}\cos(\pi x)\cos(\pi y)\right)
\end{align*}

So, the ansatz, $\hat{u}=\hat{u}(x;w,b)$, take the form:

$$u(x) = G(x) + D(x)y^L (x; w, b) $$
"""

import tensorflow as tf
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense
from tensorflow.keras import Input
import numpy as np
import matplotlib.pylab as plt

#Define the model

inputs = Input(shape=(2,))
x = Dense(10, 'sigmoid', dtype='float64')(inputs)
x = Dense(10, 'sigmoid')(x)
x = Dense(10, 'sigmoid')(x)
x = Dense(10, 'sigmoid')(x)
x = Dense(10, 'sigmoid')(x)
y = Dense(1, dtype='float64')(x)
model = Model(inputs=inputs, outputs=y)

tf.keras.utils.plot_model(model,show_shapes=True)

a, b = 1, 1/2

def u(x, y):
  return 0.5 * np.cos(np.pi*x) * np.sin(np.pi*y)

```

```python
def f(x, y, a=1, b=1/2):
  df_dx = -np.pi*np.sin(np.pi*y)*np.sin(np.pi*x)
  df_dy = np.pi*np.cos(np.pi*y)*np.cos(np.pi*x)
  return a*df_dx + b*df_dy

def D(x, y):
  points = tf.transpose([x, y])
  return model_distance(points)

def G(x, y):
  points = tf.transpose([x, y])
  return model_boundary(points)

num_samples = 5000

#Random set to train of train_size random points
inside_points = [get_random_inside_polygon(star_poly) for _ in range(num_samples)]

x_train = tf.constant([point[0] for point in inside_points], dtype='float64')
y_train = tf.constant([point[1] for point in inside_points], dtype='float64')

z_train = f(x_train, y_train)
# using the high level tf.data API for data handling
dataset = tf.data.Dataset.from_tensor_slices((x_train,y_train)).batch(1)

opt = Adam(learning_rate=0.01, beta_1=0.9, beta_2=0.99)

x_variable = tf.Variable(x_train)
y_variable = tf.Variable(y_train)

with tf.GradientTape() as gt_x, tf.GradientTape() as gt_y:
    gt_x.watch(x_variable)
    gt_y.watch(y_variable)
    G_xy = G(x_variable, y_variable)
dG_dx = gt_x.gradient(G_xy, x_variable)
dG_dy = gt_y.gradient(G_xy, y_variable)

G_pred = G(x_variable, y_variable)

with tf.GradientTape() as dt_x, tf.GradientTape() as dt_y:
    dt_x.watch(x_variable)
    dt_y.watch(y_variable)
    D_xy = D(x_variable, y_variable)
dD_dx = dt_x.gradient(D_xy, x_variable)
dD_dy = dt_y.gradient(D_xy, y_variable)

D_pred = D(x_variable, y_variable)
```

```python

ff = tf.constant(f(x_variable, y_variable))

n_train_steps = 100000

v_loss  = np.empty((n_train_steps,0))
v_error = np.empty((n_train_steps,0))
z_pred_old = np.zeros_like(x_variable)


for step in range(n_train_steps):
    with tf.GradientTape() as model_tape:
        with tf.GradientTape() as z_tape_x, tf.GradientTape() as
                                                    z_tape_y:
          z_tape_x.watch(x_variable)
          z_tape_y.watch(y_variable)
          z_pred = model(tf.transpose([x_variable, y_variable]))
        dz_dx = z_tape_x.gradient(z_pred, x_variable)
        dz_dy = z_tape_y.gradient(z_pred, y_variable)
        z_pred_old = z_pred.numpy()
        lu = a*(dG_dx.numpy() + dD_dx.numpy()*z_pred.numpy().reshape
                                                    (-1) + D_pred.numpy()*
                                                    dz_dx) + b*(dG_dy.numpy
                                                    () + dD_dy.numpy()*
                                                    z_pred.numpy().reshape(-
                                                    1) + D_pred.numpy()*
                                                    dz_dy)
        loss = tf.reduce_sum(tf.math.squared_difference(lu, ff))
        v_loss = np.append(v_loss,loss)

    grads = model_tape.gradient(loss, model.trainable_variables)
    opt.apply_gradients(
        (grad, var)
        for (grad, var) in zip(grads, model.trainable_variables)
        if grad is not None
)


    Yl = model(tf.transpose([x_variable, y_variable])).numpy().
                                                reshape(-1)
    Ds = D(x_variable, y_variable).numpy().reshape(-1)
    Gs = G(x_variable, y_variable).numpy().reshape(-1)
    u_hat = Gs+Ds*Yl
    error = np.max(np.abs(u_hat-u(x_variable, y_variable)))
    v_error = np.append(v_error, error)

np.savetxt('/content/drive/MyDrive/plots_TFM/error_advection_2D.txt'
                                            , v_error)

"""## Plotting advection model"""

```

```
@np.vectorize
def model_inside(x, y, polygon: Polygon):
  distance = model.predict([[x, y]])[0][0] if polygon.contains(Point
                                               ([x, y])) else np.nan
  return distance

@np.vectorize
def model_distances(x, y, polygon: Polygon):
  distance_value = model_distance.predict([[x, y]])[0][0] if polygon
                                               .contains(Point([x, y])) else
                                               np.nan
  return distance_value

@np.vectorize
def model_boundaries(x, y, polygon: Polygon):
  boundary_value = model_boundary.predict([[x, y]])[0][0] if polygon
                                               .contains(Point([x, y])) else
                                               np.nan
  return boundary_value

# generate 2 2d grids for the x & y bounds
y, x = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))
z = model_inside(x, y, star_poly) * model_distances(x, y, star_poly)
                                              + model_boundaries(x, y,
                                             star_poly)

z_min, z_max = np.nanmin(z), np.nanmax(z)

fig, ax = plt.subplots(figsize=(13,8))

c = ax.pcolormesh(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
ax.set_title('Smoothed function')
# set the limits of the plot to the limits of the data
ax.axis([x.min(), x.max(), y.min(), y.max()])
fig.colorbar(c, ax=ax)

plt.show()

"""## Save advection model 2D """

model.save('/content/drive/MyDrive/models_TFM/model_advection.h5')

"""## Load advection model 2D """

model = load_model(
    '/content/drive/MyDrive/models_TFM/model_advection.h5',
    custom_objects={'custom_loss':custom_loss}
    )
```