UNIVERSIDAD NACIONAL DE COLOMBIA

GRADUATION PROJECT

Physics Informed Neural Networks For Partial Differential Equation Solving

Author:
Alejandro José Yepes Peñaranda

Supervisor:
Frank Rodolfo Fonseca Fonseca

Facultad de Ciencias
Departamento de Física

June 25, 2023

One day a professor during class mentioned the following analogy

"Knowledge is like a infinite sand beach,
in which some times incredibly talented people bring to it big rocks,
but most of the time is filled grain by grain."

I would like this work to be my first little grain in that beach.

Abstract

Physics Informed Neural Networks For Partial Differential Equation Solving

by Alejandro José Yepes Peñaranda

Physics Informed Neural Networks are a new and interesting methodology of Machine Learning constructed to correctly predict the solution of differential problems that consist not only of partial differential equations but also that includes boundary or initial conditions. The goal of the present project is to develop a python module dedicated to constructing this type of architecture and it to be wide enough in order to easily adapt to the many possible differential equation systems as well as the multiple conditions to be imposed to the solution of the differential problem. Once this goal was met, multiple problems were tried out, including the simple harmonic oscillator system, Lotka-Volterra equations, reaction-diffusion systems among others, with an special interest in fluid simulation, in particular using Kovasznay flow formulation of the Navier-Stokes equations for simulating the interaction of a Newtonian fluid with an obstacle. Multiple great results were obtained, reassuring the great potential that this technology has in many fields that require a ton of computational resources, as computational fluid dynamics or even wave function estimation in complex systems used for material simulations.

Keywords: Physics Informed Neural Networs, Partial Differential Equations, Kovasznay Flow.

Acknowledgements

I would like to express my deepest gratitude and appreciation to my thesis director, professor Frank Fonseca, for all the help and invaluable guidance throughout all the process for developing this project. For pushing me forward and always attempting to get out of me the best I can give. For always giving not only me, but all his students, a word of encouragement to motivate us in difficult moments to be an student such as in the pandemic.

To my alma mater, the Universidad Nacional de Colombia, for giving me the opportunity to develop and grow not only in an academic sens, but also as a human. For giving me the chance to open my eyes and try to better understand my country through the stories that only there I could heard. To the people of Colombia, for making this space possible, although it seems like sometimes they do not comprehend the importance and the impact that the public university systems has in society.

To all my teachers, specially Mabelis Palacio, Eduardo Forero, Mauricio de la Hoz, Rafael Hurtado, Alvaro Valdes, Freddy Hernandez, José Arenas, Luis Demetrio, Thomas Muños for inspiring me and teaching me the love that I developed for science.

To my parents and brother, Margarita Peñaranda, York Yepes and Jorge Yepes, for always helping me grow and supporting me throughout this dream come true. To my grand parents, Cecilia Ruiz, José Yepes, Alicia Camargo and Luis Peñaranda (that is always with me spiritually). To my uncles and cousins, Margarita Valiente, Lina Yepes, Andrea Yepes, Luis Peñaranda, Joseph Peñaranda and Eddy Yepes, that let me share with them the years of my undergraduate studies. To my family, for always been there for me.

To my friends Cristian Mercado, Efrain Cardozo, Camilo Molinares, Andres Araujo, Fernando Suarez, Francisco Borrero, José Portela, Nicolas Piragua, Carlos Buitrago, Carlos Ramos, Daniel Rojas, Federico Aranguren, Cintia Pinilla, Laura Alfonso, Santiago Martinez and my cousin Luis Estrada, with whom I have had the honor of spending some of the best moments in my life. To my little nephew Gabriel, for giving me immeasurable happiness that recharges my soul every moment I can spent with him.

Professor F. Fonseca Fonseca would like to express its profound and sincere gratitude to the Universidad Nacional de Colombia for their unwavering support through the Hermes project 56144.

Table of Contents

Abstract **ii**

Acknowledgements **iii**

Table of Contents **iv**

List of Figures **vi**

List of Abbreviations **viii**

1 Introduction **1**

2 Theoretical Aspects **3**
2.1 Differential Equations 3
2.1.1 classification 3
2.1.2 Boundary Conditions 4
2.1.3 what is a differential problem? 4
2.2 Neural Networks, An Overview 5
2.2.1 Perceptron 5
2.2.2 Fully Connected Feed Forward Neural Networks 7
2.2.3 Universal Approximation Theorem 8
2.3 How Neural Networks Understand Physics? 11
2.3.1 Backpropagation 12
Stochastic Gradient Descent 12
2.3.2 And the physics? 13

3 Module Development **15**
3.1 Class initialization 16
3.2 Physics Enforcing 20
3.2.1 PDE Loss 20
3.2.2 BC Loss 22
3.3 Training Function 24

4 ODEs Solving using PINNs **29**
4.1 A Simple ODE example 29
4.2 Simple Harmonic Oscillator 32
4.3 Coupled ODEs 35

5 Solving PDEs using PINNs **39**
5.1 Diffusion Equation 39
5.2 One-Dimensional Burgers Equation 40
5.3 Kovasznay Flow 44
5.3.1 Flow around cylinder 46
Time Independent Solution 47
Time Dependent Simulation 48
5.3.2 Cavity Flow 51

6 Future Work **53**

7 Conclusions **55**

A Activation Functions **57**

B Other functions functions of the module. **58**
B.1 Forward Pass 58
B.2 Plot Loss History 58
B.3 Save Last Prediction 62

C Error function **63**

D Kovasznay flow discrepancy plots. **65**

E Not So Pretty PDE-PINN Examples: Exploring Challenging Results. **69**
E.1 Vibration of 2-Dimensional Membrane 69
E.1.1 Rectangular membrane 69
E.1.2 Circular membrane 71
E.2 Two-Dimensional Helmholtz Equation 72
E.3 Time dependent Cavity Flow 72

Bibliography **77**

List of Figures

2.1 Linear binary classification example 6
2.2 Non-linear binary classification example 7
2.3 Activation Function Examples 8
2.4 Feer Forward Neural Network 9
2.5 Module Point-Type Classification Example 14

4.1 Simple ODE DP ($\phi = 0$), Optimizer Comparison 30
4.2 Simple ODE DP ($\phi = \pi/2$), Optimizer Comparison 31
4.3 Simple ODE DP ($\phi = \pi$), Optimizer Comparison 31
4.4 Simple Harmonic Oscillator DP (first panel) 33
4.5 Simple Harmonic Oscillator DP (second panel) 34
4.6 Lotka-Volterra ODE system activation function comparison . . . 38

5.1 Diffusion Equation, first proposed DP 41
5.2 Diffusion Equation, second proposed DP 42
5.3 Loss History Plots For Diffusion Equation DPs 43
5.4 Burger Equation DP 45
5.5 Time-Independet Flow Interaction With An Obstacle 48
5.6 Time-Dependet Flow Interaction With An Obstacle 50
5.7 Time-Independent Flow Inside cavity Example 52

C.1 Error function behavior 64

D.1 Outlet Discrepancy Plot For Time-Dependent Flow Interaction With An Obstacle 65
D.2 Upper Wall Discrepancy Plot For Time-Dependent Flow Interaction With An Obstacle 66
D.3 Lower Wall Discrepancy Plot For Time-Dependent Flow Interaction With An Obstacle 66
D.4 Inlet Discrepancy Plot For Time-Dependent Flow Interaction With An Obstacle 67
D.5 Cylinder Wall Discrepancy Plot For Time-Dependent Flow Interaction With An Obstacle 67
D.6 Discrepancy Plots For Time-Independent Flow Interaction With An Obstacle 68

E.1 Rectangular Membrane Vibration DP 70

E.2 Radial Vibration Of Circular Membrane DP 73
E.3 Helmholtz Equation DP 74
E.4 Full-Domain Time Step Example Of The Time-Dependent Flow Inside Cavity DP 75
E.5 Zoomed Evolution Of The Time-Dependent Flow Inside Cavity DP 76

List of Abbreviations

PDE	**P**artial **D**ifferential **E**quation
FDM	**F**inite **D**ifference **M**ethod
FEM	**F**inite **E**lement **M**ethod
SM	**S**pectral **M**ethods
AI	**A**rtificial **I**ntelligence
ML	**M**achine **L**earning
NN	**N**eural **N**etwork
PINN	**P**hysics-**I**nformed **N**eural-**N**etwork
ODE	**O**rdinary **D**ifferential **E**quation
BC	**B**oundary **C**ondition
DP	**D**ifferential **P**roblem
FCN	**F**ully **C**onnected **N**etwork
QPINN	**Q**uantum-enhanced **P**hysics-**I**nformed **N**eural-**N**etwork

Introduction

Partial Differential Equations (PDEs) are an essential tool in many areas of science, particularly in physics. PDEs are used to describe the behavior of many physical systems, including the flow of fluids, propagation of waves, heat distribution, the behavior of electromagnetic fields among many other examples, as well as modelling the behavior of complex systems, such as atmospheric dynamics (e.g. Cullen, Norbury, and Purser, 1991) all the way through ecological (e.g. Brauer, Castillo-Chavez, and Castillo-Chavez, 2012) and economic (e.g. Basov, 2007) systems. The ability to solve PDEs is therefore critical for understanding and describing all the phenomena that surround us, which has led to many advances in science, technology and many other fields. Due to its importance, many methods have been developed for solving PDEs, which can be broadly categorized into analytical and computational methods (Karpatne et al., 2017).

Analytical methods involve solving the PDEs using mathematical techniques, such as separation of variables, method of characteristics, Laplace transforms, Fourier Transforms and Green's functions (Sepúlveda Soto, 2009). Each semester complete courses are imparted around the world in order to teach these, at least the least complex, methods to science and engineering students, although they are often limited to simple or idealized systems, and may not be applicable to complex or nonlinear systems. In the other hand, computational methods involve the approximation of the solution to the PDE using numerical techniques, such as finite difference (FDM) (Zhou, 1993; Smith, 1985), finite element (FEM) (Hughes, 2012; Chandrupatla and Belegundu, 2014), or spectral methods (SM) (Shen, Tang, and Wang, 2011; Hesthaven, Gottlieb, and Gottlieb, 2007). Their principle of functioning is based on dividing the domain intro a grid or mesh and use an iterative algorithm to solve the PDE. Although not providing an exact solution, they can handle complex and non linear systems and are widely used by the industry such as in Formula 1 aerodynamics (e.g. Larsson, Sato, and Ullbrand, 2005; Bienz et al., 2003) and Aircraft design(e.g. Wang, 2014; Gu, Ciampa, and Nagel, 2018), Structural engineering (e.g. Becker, 1992) and optimization of manufacturing processes (e.g. Edgar, Himmelblau, Lasdon, et al., 2001) among others.

With the massive popularization of artificial intelligence (AI) based tools in the recent years (e.g. OpenAI, 2023; Cobbe et al., 2021; Chen et al., 2020; Ingle and Phute, 2016), machine learning (ML) and in particular neural networks (NNs) had arise as a viable and innovative way for PDE solution approximation due to

their well known prediction capabilities. Unlike traditional numerical methods, NNs can learn the underlying structure of the system and accurately predict the solutions without discretizing the domain (Yadav, Yadav, Kumar, et al., 2015) or solving a system of equations. This makes them particularly effective for solving PDEs with complex geometries and nonlinear dynamics, where traditional numerical methods can be time and resource expensive. NNs can also be trained under incomplete, noisy, even the complete lack of data and can as well generalize to new scenarios. The systems that combines PDEs and NNs are called physics-informed neural networks (PINNs), and their development is the cornerstone around which its centered the present project.

The key difference between a PINN and a traditional NN when predicting the behavior of a PDE based in data points is that the PINN incorporates prior knowledge of the underlying "physics" into the NN architecture. With the prior knowledge, PINNs are able even capable of accurately capturing the behavior of a systems without the need of data (Schäfer, 2022), in contrast, traditional NNs are purely data-driven and lack the ability to explicitly enforce the physical laws governing the system being modeled. This means that traditional NNs may be less accurate when modeling physical systems, especially when experimental data is scarce or when the system being modeled is highly nonlinear.

The goal of the present project is to develop a general purpose[1] PINN python module that has an open and intuitive problem setting interface while maintaining complete freedom for boundary condition specification[2]. With this module the goal is to generate different models and apply them to the different physics problems and study their convergence as well as technology viability compared to other architectures and numerical methods.

[1]Most scripts in which the present project is based build the NN module around an specific PDE problem

[2]We acknowledge the existence of modules like PiNN (Shao et al., 2020), DeepXDE (Lu et al., 2021a), torchdiffeq (Chen, 2018) among others that already implements different approachs to tackle this goal.

Theoretical Aspects

2.1 Differential Equations

As already mentioned, differential equations, in general PDEs, are a fundamental tool for modelling complex phenomena, and are used extensively in a wide range of fields, including physics, engineering, materials science, finance and biology. The goal of the present section is to introduce key concepts as well as the notation to be used through the rest of the work.

At its most basic level, a differential equation consists of a **function**, or functions, ($\mathbf{f}$) and one or more of its derivatives, as well as any **independent variables** ($\mathbf{x}$) that the function depends on (Moya and Rojas, 2021). It is important to note that differential equations may, as well as "regular" equations, be part of a greater structure involving multiple of them defined as a **system**. The **solution** of a differential equation (or a system of differential equations) is then said to be the function (or set of functions) that satisfy the equation (or system). In the words of 2021, the solution of a differential equation is a function that, when substituted into the differential equation, makes it an identity.

2.1.1 classification

Differential equations can be classified into several types based on various properties, including their order, linearity, and specific properties such as ellipticity or hyperbolicity. One of the most common ways to classify differential equations is based on the number of independent variables they involve. Ordinary differential equations (ODEs) involve only one independent variable and describe the behavior of a function with respect to that variable. PDEs. on the other hand, involve multiple independent variables and describe the behavior of a function with respect to each of those variables (Nagle et al., 2005; Moya and Rojas, 2021). This distinction is typically important because the methods used to solve ODEs differ from those used to solve PDEs. ODEs are often solved using analytical techniques, while PDEs mostly require numerical methods due to their complexity. Additionally, the specific type of PDE can also impact the methods used to solve it, with elliptic, parabolic, and hyperbolic equations each requiring different techniques.

2.1.2 Boundary Conditions

Boundary conditions (BC) are a crucial part of many differential equations. They specify the behavior of the solution at the boundaries or edges of the domain in the the equation has been defined. In the case of ODEs, BCs often take the form of initial conditions, which specify the value of the solution and its derivatives at a certain starting point. In the case of PDEs they tend to be more complex, as they involve specifying the behavior of the solution along multiple boundaries in a multidimensional space. There are several types of BCs, including[1]:

- Dirichlet BCs: These conditions specify the value of the solution at the boundary,
- Neumann BCs: these conditions specify the derivative of the solution at the boundary,
- Robin BCs: these conditions specify a combination of the value and the derivative of the solution at the boundary,
- Mixed BCs: these conditions are a combinations of different types of boundary conditions at different parts of the boundary.[2]

2.1.3 what is a differential problem?

One then may define a **differential problem** (DP) as the combination of independent variables (defined in a domain), functions, the differential equation system that relates one another and the respective BCs that "fixes" the behavior of the solution at certain subdomain. For example, one of the DPs boarded while developing the module constructed in the present work was the simple harmonic oscillator. This DP includes the time (t) as independent variable, the spatial displacement (x) as function, the following differential equation, along with its corresponding initial condition as BC:

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = -\omega_0^2 x \qquad x(0) = A; \left.\frac{\mathrm{d}x}{\mathrm{d}t}\right|_{t=0} = 0.$$

In the module construction several architecture decisions, later to be discussed, were made, leading to the simplification of the notation as follows:

- No distinction between ODE and PDE is made at the moment of solving them with PINNs, they are all considered PDEs.

[1] Classification taken from Polyanin and Zaitsev, 1995 & Polyanin, 2016

[2] Robin boundary conditions are a single condition that includes both value and derivative information, while mixed boundary conditions are multiple conditions that specify different types of information at different parts of the boundary.

- All thee DP are considered as differential equations systems.
- BCs types do not matter, as well as their subdomain. For this reason when declaring them they can also be used for feeding the module with data from the "exact" solution[3].

It is also important to state that, for the sake of notation, every differential equation most be expressed as $D(x, f) = 0$, where D is a differential operator[4] that represents the equation. This operator can also include independent terms, such a forcing term in the case of the harmonic oscillator example. This change is made to later better express crucial concepts relating to the learning process.

2.2 Neural Networks, An Overview

NNs, or more precisely artificial neural networks, are a type of machine learning algorithm that have gained significant attention in recent years due to their ability to recognize patterns, make predictions and learn from data. They are used in a wide range of applications, including image and video recognition, natural language processing, speech and pattern recognition, robotics and autonomous systems among others. With their ability to analyze complex data and make decisions based on that analysis, neural networks have become an important tool in various fields of study, and their applications continue to expand as technology advances.

Mathematically speaking, NNs are a parallel distributed information processing structure in the form of a directed graph (Yadav, Yadav, Kumar, et al., 2015). Their beginning is taken to be McCulloch and Pitts, 1943 in which it is shown that they could, in theory, compute any arithmetic or logical function. The first successful neuro-computing unit is set to be the perceptron, developed in late 1950 by Frank Rosenblatt (Rosenblatt, 1958), Charles Wightman among others (Rosenblatt, 1961).

2.2.1 Perceptron

The perceptron is not other than the mathematical model of the biological neurons, and as so, they are the basic processing unit of the "nervous system", here modeled by a NN. The biological neurons are constituted out of dendrites, axon and a cell body, also known as soma, (Snell, 2010), their modeled equivalent are the input, output and the "body" which is in charge of the data processing. At

[3]It is typically used to feed a PINN using the results obtained through a "classical" computational algorithm such as finite differences.

[4]This notation was adopted for its simplicity, although a more accurate (and maybe correct) way of expressing it might be: $D(x, f) + F(x) = 0$.

its most simple use, the inputs and output of the neuron are treated as numbers, and the process that relates one another is:

$$z_j = b_j + \sum_{k=1}^{n} w_{jk} x_k, \tag{2.1}$$

where O_i is the output of the $i - th$ neuron, b_i is a bias term, w_{ij} is the weight (or strength) of the x_j input and n is the total number of inputs for the neuron. Used by its own, a perceptron unit can be used for binary classification problems, for example[5]: Let there be a population of flowers which can be red or blue, labeled as 1 and 0 respectively. An study measures two characteristics of them, like for example the height and width of the petals (let this parameters be the independent variables x_1 and x_2 respectively). One then may be tempted to use the measurements to classify the flowers as shown in Fig. 2.1 bellow.

Color	x_1	x_2
1	0.5	2.0
1	1.1	2.1
1	0.7	2.6
0	2.0	1.0
0	2.5	0.7
0	2.2	0.3

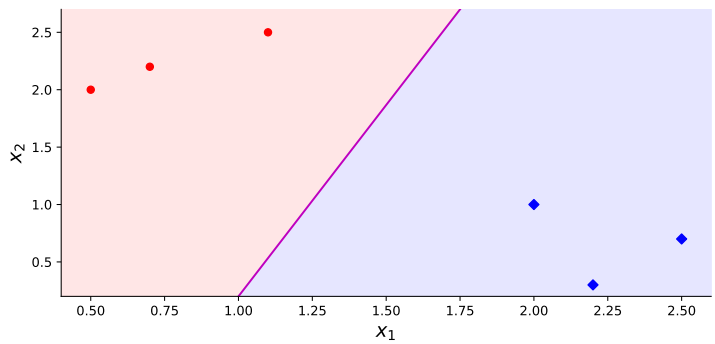

Figure 2.1 & Table 2.1: Binary classification problem using perceptron.

This is a toy model that makes use of perceptron for classification, if the perceptron output is lower than zero the flower is blue and red other wise. The purple line is called frontier and is the region at which the the output of perceptron is equal to zero, in mathematical terms $0 = b + w_1 v_x + w_2 x_2$. If extrapolated to greater dimensions, the frontier of decision is transformed into an hyperplane of that space, due to the linearity of the perceptron processing. this linearity is also a limiting factor for the use, as for example show in Fig. 2.2, where the dashed black line would be a more suitable frontier, rather than the ones estimated by a perceptron.

To better capture the behavior of the data presented in Fig. 2.2 we might propose as inputs the squared value of the measured data. This approach might result use-full, although when a more complex data distribution is encountered, it might not be as simple the solution to predict the behavior. A better approach is

[5]this example is an adaptation of 2021

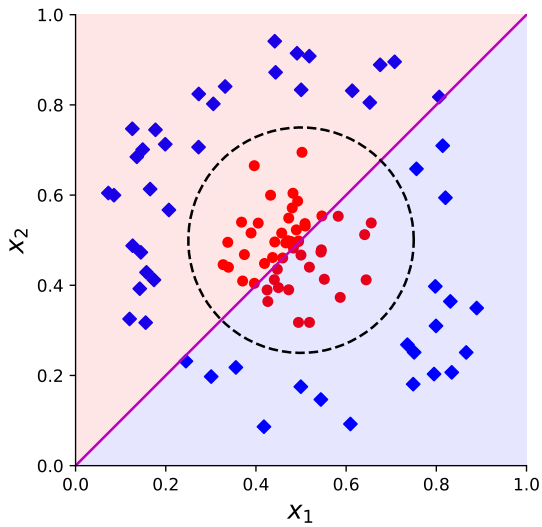

Figure 2.2: Non linear binary classification problem.

then to increase the number of perceptron cells used for the classification problem. Making a **network**, resembling the processing mechanism of the human brain.

2.2.2 Fully Connected Feed Forward Neural Networks

There are multiple ways a network can be constructed, each of this is said to be an **architecture**. Generally speaking, the architecture implemented for NNs consists of three types of neuron layers: input, hidden and output. The input layer is where the data is introduced into the NN, each neuron, also referred to as node, correspond to each feature of the input data, for example the two data measurements for each flower. The hidden layers are where the processing occurs, for example the above defined for the perceptron. The output layer is where the final result of the NN is generated, again for the flower classification example, this would only consists of one node corresponding to the color of the flower.(Sazli, 2006)

Particularly, the architecture implemented for the module construction is fully connected, which mean that the output of each node of the layer i is passed as input to every node of the $i + 1$. Between each layer a function is applied to all the outputs of the prior layer before connecting them to the next layer. This function is know as **activation function**, and is used in order to introduce non-linearities into the output of each neuron. There are multiple types of activation functions, some of them shown in Fig. 2.3.

It most also be stated that in recent times specialized activation functions are been developed for PINNs for example in Abbasi and Andersen, 2022. There is,

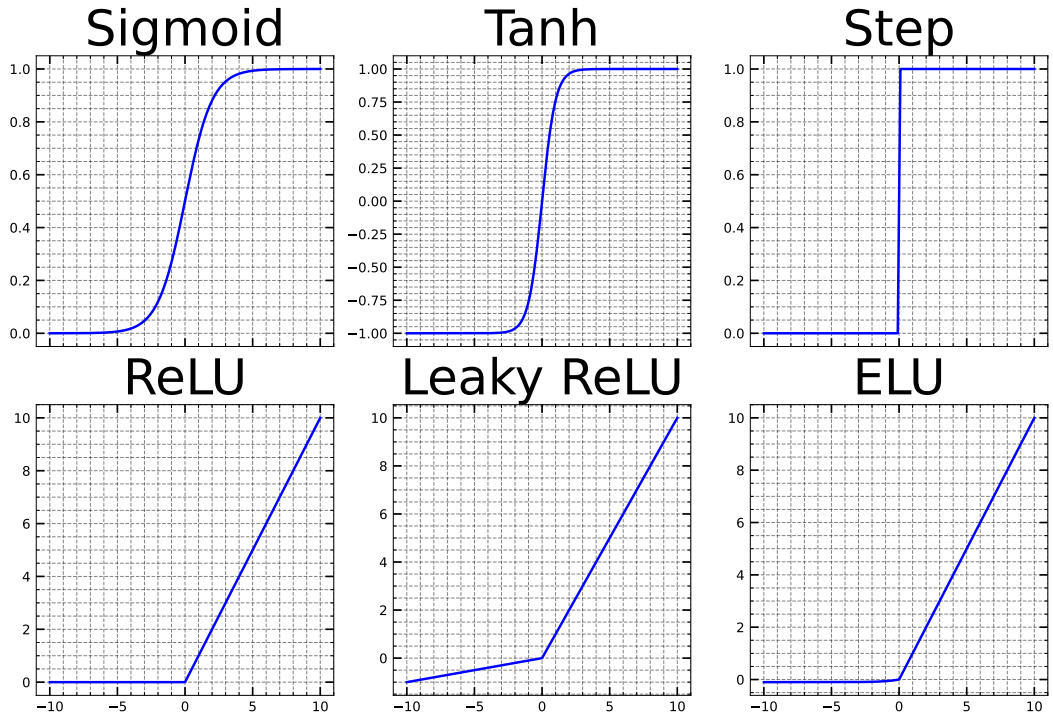

Figure 2.3: Common activation functions used in NNs. Their definition can be found in Tab. A.1

in fact, multiple implementations of architectures of NN for PINNs, in particular for the development of the module we use what is called as **Feed Forwrad**, which are the simplest form of NN. This type of architecture is characterised by having only unidirectional information flow, this means that, in comparison with other architectures, it is no possible for the NN to predict based on its own output. It will be a complete work by its own to study the impact of different NN architectures, such as Recurrent NNs, Radial Basis Function NN, Hopfield Network (with all its variations), Cellular NN, Finite Element NN, Wavelet NN among others, a better explanation of each one can be found in 2015. A representation of a fully connected, feed forward, neural network is show in Fig. 2.4, this NN are referred to by the number of layers they present, being this notation as follows $[i, l_1, \ldots, l_l, \ldots, l_n, o]$, where i and o are the number of input features and outputs, respectively, of the NN and l_l the number of nodes of the hidden layer l. If for example, there are n consecutive layers with the same size m, the notation is "relaxed" as $m \otimes n$, this notation is used in Fig. 2.4.

2.2.3 Universal Approximation Theorem

This section is deeply inspired by the work shown by Yadav, Yadav, Kumar, et al., 2015, with inspiration also taken from Kidger, 2022; Teshima et al., 2020. The universal approximation theorem for PINNs was first introduced by Hornik,

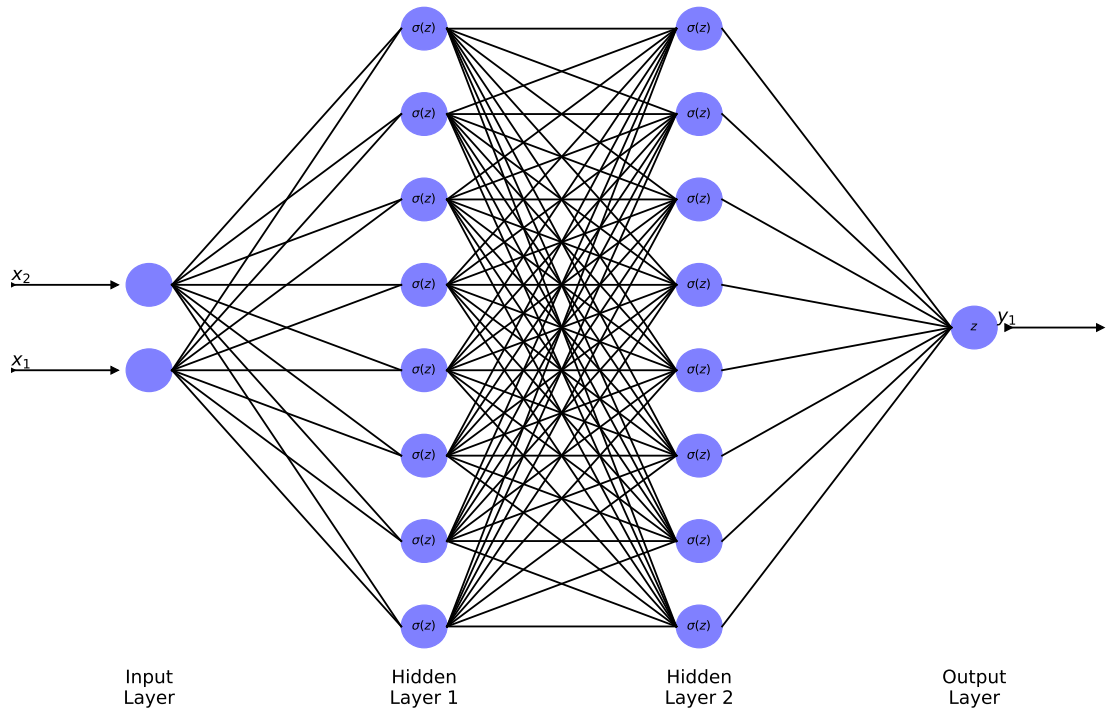

Figure 2.4: Example of $[2, 8 \otimes 2, 1]$ Feed Forward Neural Network.

Stinchcombe, and White, 1989; Hornik, Stinchcombe, and White, 1990.

At first, one may be tempted to used the NNs for solving DPs but, is it correct in a mathematical sense? Let $x^{[i]}$[6] be the input vector of the $i-th$ hidden layer, $W_j^{[i]}$ is then the weight vector and $b_j^{[i]}$ the bias associated with neuron j of the layer. Without taking into account the activation function, the output of the neuron would be as shown before in Eqn. 2.1. With the activation function σ it will transform as follows:

$$a_j^{[i]} = \sigma\left(W_j^{[i]} \cdot x^{[i]} + b_j^{[i]}\right). \tag{2.2}$$

For a $[n, 1, 1]$, the output $\hat{y}$ of the NN would be computed as:

$$\begin{aligned} z^{[1]} &= W_1^{[1]} \cdot x + b^{[1]} \\ a^{[1]} &= \sigma(z^{[1]}) \\ z^{[2]} &= W_1^{[2]} \cdot a^{[1]} + b^{[2]} \\ a^{[2]} = \sigma(z^{[2]})\hat{y} &= z^{[2]} \end{aligned}$$

[6] As the NN is fully connected, all the neurons in layer [i] will have the same input

A hidden layer of n inputs and m neurons, hence m outputs, would then be expressed as a matrix of the form:

$$W^{[i]} = \begin{pmatrix} W_1 \\ \vdots \\ W_m \end{pmatrix} \quad \textit{or more precisely} \quad W^{[i]} = \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{pmatrix}.$$

Meaning that the output of the NN can be expressed as:

$$\hat{y} = \prod_{[i]} \sigma^{[i]} \left(W^{[i]} \cdot x^{[i]} + b^{[i]} \right), \tag{2.3}$$

where $[i]$ iterates over all the hidden layer and the output layer $[o]$, considering $\sigma^{[o]}$ as an identity function. Depending on the DP, $\hat{y}$ can have different dimension, depending on the number m of functions to be predicted. Focusing only in on of them, the prediction of each function is:

$$f_m = \sum_{j=1}^{n_o} W_{mj}^{[o]} \left[\prod_{[i]}^{n_o} \sigma^{[i]} \left(W^{[i]} \cdot x^{[i]} + b^{[i]} \right) \right]_j + b_m^{[o]}. \tag{2.4}$$

It has been found in literature that feed forward NN are capable of approximating any Borel measurable function from one finite dimensional space to another with any desired degree of accuracy (Yadav, Yadav, Kumar, et al., 2015). The aforementioned based in the following definitions and theorems taken from Hornik, Stinchcombe, and White, 1989; Hornik, Stinchcombe, and White, 1990; Yadav, Yadav, Kumar, et al., 2015.

Definition 1:

"If $A^r : R^r \to R$ is the set of all functions of the form $A(x) = w \cdot x + b$, where w and x are vectors in R^r, $b \in R$ is a scalar and "$\cdot$" is the usual dot product for any $r \in N$." Yadav, Yadav, Kumar, et al., 2015.

Definition 2:

"Let $\sum^r G$ be a class of functions for any Borel measurable function $G(\cdot)$ is a mapping from R to R and $r \in N$.

$$f(x) = \sum_{j=1} O_j G(A_j(x)) \tag{2.5}$$

where, $x \in R^r, O_j \in R, A_j \in A^r$ and $q = 1, 2, \ldots$." Yadav, Yadav, Kumar, et al., 2015.

Theorem 1:

"*If G be any continuous non constant function from R to R, then $\Sigma\Pi^r(G)$ is uniformly dense on compacta in C^r, where C^r is a compat function from R^r to R.*" Yadav, Yadav, Kumar, et al., 2015. This means that feed forward neural networks, denoted by $\Sigma\Pi$ as in Eqn. 2.4, are capable of approximate, with arbitrary tolerance, any real valued continuous function over a compact set. An important consideration for the validity of the theorem is that the activation function most be continuous and non constant.

Theorem 2:

"*For every continuous non constant function G, every r and every probability measure μ on (R^r, B^r), $\Sigma\Pi^r G$ is ρ_u-dense in M^r.*" Yadav, Yadav, Kumar, et al., 2015. What this theorem states is that the standard feed forward NN with only a single hidden layer can approximate any continuous function uniformly on compact set and any measurable function arbitrary well in ρ_u. This means that Σ networks are also universal approximator.

Theorem 3:

"*For every function g in M^r there is a compact subset K of R^r and an $f \in \Sigma^r(\psi)$ such that for any $\epsilon > 0, \exists |f(x) - g(x)| < \epsilon$.*" Yadav, Yadav, Kumar, et al., 2015. This means that a single hidden layer feed forward NN can approximate any measurable function with an arbitrary precision (ϵ).

The above stated definitions and theorems are further explored in Kidger, 2022, giving specific theorems for cases such as time series.

2.3 How Neural Networks Understand Physics?

To answer this question we might take a step back and ask, how do neural networks learn? There are multiple classification of learning in AI, supervised, unsupervised, reinforced and competitive. In particular the learning method selected for the development of the present project was Supervised Learning, which its most important characteristic is to compare the NN predicted output to a target[7] and "inform" the NN how apart they are(Cunningham, Cord, and Delany, 2008; Desai, 2021). The distance from the NN output and the target its computed through a "Loss" function and may vary for each type of problem we try to tackle. For example, when trying to train a NN for binary classification,

[7]The NN here constructed doesn't require the a-priory knowledge of the DP exact solution, rather in this sense, the target and the deviation of the predicted output from it is computed using other methods later to be explained.

where the classes are encoded as $\{0, 1\}$, the most commonly used Loss function is $L(y, \hat{y}) = y - \sigma(\hat{y})$, with σ the step function, this function only outputs a one when the target and the prediction differs, the total loss is then computed as $\mathcal{L} = \frac{1}{N}\sum_i^N L(y_i, \hat{y}_i)$. But, what does a NN do with this information?

2.3.1 Backpropagation

Training a NN is an iterative process in which there exists to phases, a forward computation of the predicted output, and correspondingly the loss, and a backward process in which all the nodes "take consiousness" of how far are they from correctly predicting the output and how they can optimize their parameters in order to better perform. The most basic form of optimizer is the Stochastic Gradient Descent algorithm, this consist of updating the weight connection as well as the bias terms of each node taking into account the gradient of the loss (Rumelhart, Hinton, and Williams, 1986).

Stochastic Gradient Descent

Lets say the at training iteration k the NN has a set of parameters $W_k^{[i]}$, $b_k^{[i]}$ at layer $[i]$, with a set of inputs x its output it $\hat{y}_k = \prod_{[i]} \sigma^{[i]}\left(W_k^{[i]} \cdot x^{[i]} + b_k^{[i]}\right)$ which produces a loss $\mathcal{L}_k$. To optimize the result at the next iteration, $k+1$, the parameters are updated as follows:

$$W_{k+1}^{[i]} = W_k^{[i]} - \lambda \frac{\partial \mathcal{L}}{\partial W^{[i]}}, \qquad b_{k+1}^{[i]} = b_k^{[i]} - \lambda \frac{\partial \mathcal{L}}{\partial b^{[i]}}, \tag{2.6}$$

where λ is a manually set parameter called *learning-rate*(Rumelhart, Hinton, and Williams, 1986). As seen in the Eqn. 2.6, the hyperparemeter optimization is based on how much each of them has an impact in the loss, it is necessary to introduce the learning rate to control how much the parameter changes to ensure a smoother convergence to the desired output. Lets revisit the NN $[n, 1, 1]$ with σ the activation function and $\mathcal{L}_k$ the total loss at training iteration k. To update the parameters of each layer we most then follow[8] (Harold, Kushner, and Yin, 1997):

[8]although no subindices included, the following equations refers to the parameters at training iteration k

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W^{[2]}} &= \frac{\partial \mathcal{L}}{\partial a^{[2]}} \times \frac{\partial a^{[2]}}{\partial z^{[2]}} \times \frac{\partial z^{[2]}}{\partial W^{[2]}} \\
\frac{\partial \mathcal{L}}{\partial b^{[2]}} &= \frac{\partial \mathcal{L}}{\partial a^{[2]}} \times \frac{\partial a^{[2]}}{\partial z^{[2]}} \times \frac{\partial z^{[2]}}{\partial b^{[2]}} \\
\frac{\partial \mathcal{L}}{\partial W^{[1]}} &= \frac{\partial \mathcal{L}}{\partial a^{[2]}} \times \frac{\partial a^{[2]}}{\partial z^{[2]}} \times \frac{\partial z^{[2]}}{\partial a^{[1]}} \times \frac{\partial a^{[1]}}{\partial z^{[1]}} \times \frac{\partial z^{[1]}}{\partial W^{[1]}} \\
\frac{\partial \mathcal{L}}{\partial b^{[1]}} &= \frac{\partial \mathcal{L}}{\partial a^{[2]}} \times \frac{\partial a^{[2]}}{\partial z^{[2]}} \times \frac{\partial z^{[2]}}{\partial a^{[1]}} \times \frac{\partial a^{[1]}}{\partial z^{[1]}} \times \frac{\partial z^{[1]}}{\partial b^{[1]}}
\end{aligned}
$$

In general, for a NN with M hidden+output layers, the partial derivative with respect to the parameters at layer number p are computed as[9]:

$$
\frac{\partial \mathcal{L}}{\partial \theta^{[p]}} = \frac{\partial \mathcal{L}}{\partial z^{[M]}} \times \frac{\partial z^{[M]}}{\partial a^{[M-1]}} \times \frac{\partial a^{[M-1]}}{\partial \cdots} \cdots \frac{\partial \cdots}{\partial z^{[p+1]}} \times \frac{\partial z^{[p+1]}}{\partial a^{[p]}} \times \frac{\partial a^{[p]}}{\partial z^{[p]}} \times \frac{\partial z^{[p]}}{\partial \theta^{[p]}}, \tag{2.7}
$$

where $\theta^{[p]}$ is the general way of referring to the hyper-parameters $W^{[p]}$ and $b^{[p]}$ (Amari, 1993; Bottou, 2012; Rumelhart, Hinton, and Williams, 1986; Harold, Kushner, and Yin, 1997). The blue highlighted text is the algorithm known as backpropagation, and is the logical process used to compute the gradients.

It most be added that with the development of computational methods other optimizers had been developed that can offer a better performance such as Adam (Kingma and Ba, 2014) and Limited-memory Broyden–Fletcher–Goldfarb–Shanno (LBFGS) (Zhu et al., 1997; Saputro and Widyaningsih, 2017), this last rather than just the gradient, uses the Hessian matrix[10] to update the hyper-parameters.

2.3.2 And the physics?

The aforementioned algorithms are of general use, but for the sake of training a neural network to solve a DP something else is needed. The "physics" is introduced to the PINN through the loss function, this means that the training process differs from a regular NN. The total loss is computed as:

$$
\mathcal{L} = \sum_{i}^{N_1} MSE(ResPDE_i(\hat{y}), 0) + \sum_{j}^{N_2} \alpha_j MSE(ResBC_j(\hat{y}_\partial), y_\partial), \tag{2.8}
$$

Where MSE is the Mean Squared Error function, that outputs the mean value of the squared difference between the two inputs and Res is the residual of the differential equation (PDE) and the boundary conditions (BC) and N_1, N_2 are

[9] in this example we wont assume that the loss function depends on the $a^{[M]}$ but just rather directly of $z^{[M]}$, the difference would be just to include the term: $\frac{\partial \mathcal{L}}{\partial a^{[M]}} \times \frac{\partial a^{[M]}}{\partial z^{[M]}}$ replacing $\frac{\partial \mathcal{L}}{\partial z^{[M]}}$

[10] ergo second order partial derivatives

the number of PDE and BC in the DP (Xiang et al., 2021; Thuerey et al., 2021; Desai, 2021). Taking into account that the PDE, when given the exact solution, its equal to zero, due to existence an uniqueness of the solution, when given the approximation estimated by the PINN, there will be a "residual" value the operator is equal to. All the PDE losses are computed using collocation points, which are all the points used for training, depending on the DP, each BC loss can be computed using a different set of points (this points are included are also collocation points), an example of the difference is presented in Fig. 2.5. Each BC loss is weighted by a parameter $\alpha \in [1, \infty)$ to compensate the size of the data points they are computed with.

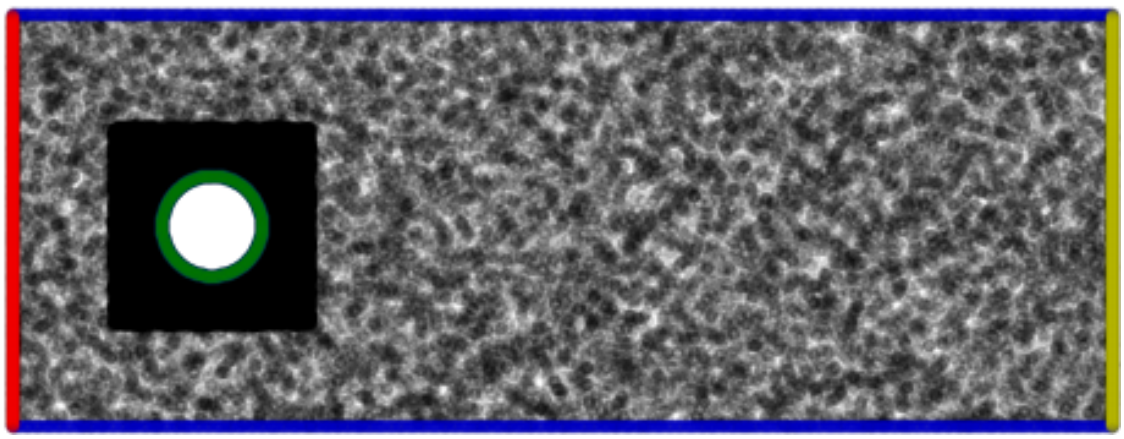

Figure 2.5: Color coded points to compute residuals. In red, blue, yellow and green points corresponding to boundary points x_∂. All the points, including the black are collocation points.

With the aforementioned, the PINN can determine how close to the exact solution is the estimated output and optimize it. It most also be stated that the collocation points are not necessarily the same points used for estimating later solutions, in the example of Fig 2.5 there exists a zone with higher point density due to the importance of the BC color coded with green, but for behavior estimation a more regular grid is preferred, as presented in Sec. 5.3.1, this is one great advantage of the PINN PDE solving methods respect to traditional ones.

Module Development

Once the theoretical aspects that support the theory are set, it is time now for discussing computational aspects and important architecture decisions that have been made while developing the module. The first important aspect is the selected programming language which was python. Firstly, Python is renown for its simplicity and readability, this makes it easier to create modules as well as expressing complex code blocks related to the declaration of PDE of BC. Secondly, Python present a vast array of scientific libraries and frameworks such as SciPy, NumPy, Matplotlib among other that further facilitates code implementation, as well as the existence of already developed machine learning libraries as TensorFlow, PyTorch, Scikit-learn, Keras among many others that makes easier the construction of the module, enabling efficient training and numerical computations. Python also present the largest active developer community that provides help and guidance, as well as maintaining updated the modules as well as their documentation[1]. The library selected as the backbone of the module is Pytorch due to its active research community, extensive customization options as well as its easy integration with other python modules, although the other considered options, specially TensorFlow and Keras (a TensorFlow API), are as good as the selected.

The required python packages for the module to function are:

- PyTorch
- Matplotlib
- Numpy
- tqdm
- time
- warnings
- os

The core of the module, at this moment in time, is the *Class* Fully Connected Network (FCN), which is in charge of NN construction as well as physics training.

[1]another important aspect, without trying to brag, is my bast experience programming in python obtained through multiple scientific projects, as well as courses.

3.1 Class initialization

The class FCN has as parent class the module "torch.nn" and has as initialization parameters the following:

- layers: list that contain the NN architecture (e.g. [1,20,20,1])
- ResPDE: function that computes the residuals of all the PDEs of the DP. Value most be set to a callable object.
- HardConstrain=None: optional argument in which a function can be applied to the output of the NN, for example to ensure that an argument is non-negative. If specified, value most be set to a callable object.
- activation=nn.Tanh(): activation function σ to be applied to the output of each node.
- lr=0.5: learning rate λ to be used in training.
- criterion=nn.MSELoss(reduction='mean'): Loss function L.
- optimizer='Adam': optimizer algorithm to use.
- scheduler=None: optional argument to vary the learning rate through the training process, most be set to a valid PyTorch scheduler.
- argsoptim={}: additional arguments for the optimizer.
- argschedu={}: additional arguments for the scheduler.
- numequ=1: number of PDE in the DP.
- numcdt=0: number of conditions to impose in the DP.
- focusmin=0.05: parameter to control training loss history graphic.
- focusmax=0.05: parameter to control training loss history graphic.
- name='PDESolver': name to store the PINN of a particular DP.

The firsts lines of code in the initialization function inherits the parent class properties as well as create, if not already, the working directories in which the NN will be stored, as well as the graphics it may produce. The corresponding code is shown bellow[2]:

[2]The code was written using visual studio code extension "better comments", which enables especial color coded line comments with keys such as *,? or TODO, it is recommended to use such tool while reading the code.

```
1  super().__init__() #? call __init__ from parent class
2  if not exists('Models'):
3      makedirs('Models')
4  if not exists('Figures'):
5      makedirs('Figures')
6  'model name'
7  self.name=name
8  if not exists(f'Models/{self.name}'):
9      makedirs(f'Models/{self.name}')
10     makedirs(f'Models/{self.name}/Final')
11     makedirs(f'Models/{self.name}/Best')
12 if not exists(f'Figures/{self.name}'):
13     makedirs(f'Figures/{self.name}')
14     makedirs(f'Figures/{self.name}/Images')
15     makedirs(f'Figures/{self.name}/Instances')
16 self.n = len([file for file in listdir(f'Models/{self.
   name}/Final')])+1
```

where in line 16 the value n is estimated, this number is important for DP that have been trained multiple times. The following code lines are in charge of initialization of important properties like activation function, loss function and the number of PDE and BC as well as catching secondary parameters, not strictly necessary for training, such as focusmin, focusmax or HardConstrain[3]. The code is presented bellow:

```
1  'activation function'
2  self.activation = activation#? activation function
   applied after each layer
3
4  'criterion'#? used for computing the loss out of the
   residuals estimated
5  self.criterion = criterion#? for forward(input) and
   predicted output
6
7  'hard constrain'#? used for imposing boundary conditions
8  if callable(HardConstrain):
9      self.HardConstrain = HardConstrain
10 else:
11     self.HardConstrain = None
12     if HardConstrain is not None:
13         print(f'Argument HardConstrain set to type: {
           type(HardConstrain)}, expected type function')
```

[3]the addition of this parameter is inspired by its existence in Lu et al., 2021a

```

'Number of equations in the system'
self.numequ = numequ
self.numcdt  = numcdt

'zoom to loss plots'

try:self.focusmin = float(focusmin)
except:self.focusmin = None
try:self.focusmax = float(focusmax)
except:self.focusmax = None

'Function to compute residuals of the PDE'
self.ResPDE = ResPDE#! OUTPUT MOST BE ITERABLE
```

The lats clarification is really important cause it speaks of the shape of the output expected from the function ResPDE (this function is explained in the following section). Lets say the DP that only needs a function to be estimated, it will be computed using a tensor x with shape $[m, n]$, m the number of data-points and n the number of input variables, the expected output of the ResPDE is $[m, 1]$ instead of just $[m,]$. The following block code treats the NN creation as well as its node initialization:

```
'Initialise neural network as a list using nn.Modulelist'
self.layers = layers
self.linears = nn.ModuleList([nn.Linear(self.layers[i], self.layers[i+1]) for i in range(len(self.layers)-1)]) #? construction of the network
self.iter = 0

'device to be used for training'
self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

'Xavier Normal Initialization'
# std = gain * sqrt(2/(input_dim+output_dim))
for i in range(len(self.layers)-1):

    # weights from a normal distribution with
    # Recommended gain value for tanh = 5/3?
    nn.init.xavier_normal_(self.linears[i].weight.data, gain=5/3)

```

```
    # set biases to zero
    nn.init.zeros_(self.linears[i].bias.data)
self.to(self.device)

'Initialize best loss'
self.best_valid_loss = float('inf')
```

The core of the above shown code was taken from Toscano, 2022[4]. The node initialization is Xavier normal (Glorot and Bengio, 2010), this ramdonly sets each weight of layer $[i]$ to a value out of the distribution $\mathcal{N}\left(0, \frac{5}{3}\sqrt{\frac{2}{n^{[i]}+n^{[i+1]}}}\right)$, where $n^{[i]}$ are the number of inputs of the $i-$th layer, and the biases to zero. This initialization will improve the convergence rate of the PINN. The variable "best_valid_loss" is also initialize later to be used. The remaining code block initialize the scheduler and the optimizer.

```
'optimizer selection'
try:
    self.optimizer = getattr(torch.optim, optimizer)(self.parameters(), lr=lr, **argsoptim)
except:
    self.optimizer = torch.optim.Adam(self.parameters(), lr=lr,**argsoptim)
    print(f'\nOptimizer "{optimizer}" was not found, Adam was used instead\n')

'learning rate scheduler selections'#? if wanted, the code can implement a learning rate scheduler
try:
    self.scheduler = getattr(torch.optim.lr_scheduler, scheduler)
except:
    self.scheduler = None
    if scheduler is not None:print(f'Learning Rate Scheduler "{scheduler}" was not found. Ingoring option.')
if self.scheduler is not None:self.scheduler = self.scheduler(self.optimizer,**argschedu)
```

[4]He has published an incredible series of videos in YouTube that explains very well the construction of a PINN.

3.2 Physics Enforcing

To train the PINN it is necessary to take into account the DP, in order to correctly approximate its solution. The total loss, as shown before, is the most important tool for doing so, in the module it is computed as follows:

```
def loss(self):
    """
    this function computes the total loss for the PDE solution estimation problem,
    it takes into account the loss generated by

    Returns:
        total_loss (_type_): lossPDE or weight*lossBD + lossPDE.
    """
    total_loss = sum(self.lossPDE(self.train_Xpde))
    if self.numcdt>0:
        total_loss+=sum(self.lossCDT())
    return total_loss
```

The function takes no input because they have been elsewhere attached to the class as parameters, such as 'self.train_Xpde'. As it may be observed, the "loss" function only calls two sub-functions which are the ones that are really in charge of the computation of the loss of the DP. Although it is common in the different available PINN generating modules to manage separately the loss terms involving the residual of the boundary conditions as well as the PDE system (e.g. Lu et al., 2021a), the interface for specifying the PDE as well as the boundary conditions is of complete own production, developed with the philosophy of giving complete control to the user for DP declaration.

3.2.1 PDE Loss

This function is indispensable for PINN training. As previously mentioned, it assumes that each PDE of the DP is written as an operator of the form $D(x, y) = 0$, thus when switching to the estimation computed by the PINN the output would be $D(x, \hat{y}) = ResPDE$, meaning that the value to compute the difference to is 0, this is assumed in lines 14 and 20 of the following code:

```
1  def lossPDE(self,Xpde):
2      """
3      This function computes the loss generated due to collocation points,
4      (aka points inside the regimen for computing the PDE). It used the
```

```
    residuals of the PDE so it is asummed for the exact solution f that:
    PDE(f,x)=0

    Args:
        Xpde (_type_): Collocation points

    Returns:
        lossPDE (_type_): loss due to the collocation points output estimation.
    """
    f_hat = torch.zeros(Xpde.shape[0],1).to(self.device)
    XpdeC = Xpde.clone()
    XpdeC.requires_grad = True
    loss_pde = []
    Ests = self.ResPDE(XpdeC,self.forward(XpdeC))
    for f_est in Ests:
        loss_pde.append(self.criterion(f_est,f_hat))
    return loss_pde
```

The most important component of the code is the ResPDE function, and as mentioned before, is declared by the use. This was made this way in order to give complete freedom for the module to be used in different scenarios. Lets use as an example the DP of coupled harmonic oscillators, the PDEs are:

$$m\frac{\partial^2 \Delta x_1}{\partial t^2} + (k+K)\Delta x_1 - K\Delta x_2 = 0,$$
$$m\frac{\partial^2 \Delta x_2}{\partial t^2} + (k+K)\Delta x_2 - K\Delta x_1 = 0,$$

Where Δx represents the displacement of each oscillator relative to the equilibrium point. The corresponding ResPDE function is:

```
def ResPDE(Tpde,Fcomp):
    y1, y2 = Fcomp.T
    y1 = y1.view(-1,1)
    y2 = y2.view(-1,1)

    y1_t = autograd.grad(y1,Tpde,torch.ones([Tpde.shape[0], 1]).to(model.device), retain_graph=True, create_graph=True)[0] #first derivative
    y1_tt = autograd.grad(y1_t,Tpde,torch.ones(Tpde.shape).to(model.device), create_graph=True)[0] # second derivative
```

```
    y2_t = autograd.grad(y2,Tpde,torch.ones([Tpde.shape[0], 1]).to(model.device), retain_graph=True, create_graph=True)[0] #first derivative
    y2_tt = autograd.grad(y2_t,Tpde,torch.ones(Tpde.shape).to(model.device), create_graph=True)[0] #second derivative

    f1 = m*y1_tt + (k+K)*y1 - K*y2 #? first equation residual
    f2 = m*y2_tt + (k+K)*y2 - K*y1 #? second equation residual

    return [f1,f2]
```

By standardization, the ResPDE function has two inputs which are a copy of the collocating points, given to the NN in the training function, later to be explained, and the prediction of the PINN made using this points. Out of this the user can make the necessary transformations to compute the residuals of the PDEs. The derivatives are computed using "torch.autograd" module, in this case they are straight forward, but if for example, $Tpde$ is constituted out of x, y, t, the output of this function would be:

```
y1, y2 = Fcomp.T
y1 = y1.view(-1,1)
y2 = y2.view(-1,1)

y1_x_y_t = autograd.grad(y1,Tpde,torch.ones([Tpde.shape[0], 1]).to(model.device), retain_graph=True, create_graph=True)[0] #first derivative

y1_t = y1_x_y_t[:,[2]]
```

3.2.2 BC Loss

The function to compute the loss out of the BCs is also simple to comprehend if habituated to python syntax, in which the weights α of Eqn. 2.8 are taken into account.

```
def lossCDT(self):
    """
    This function computes the loss generated due to boundary conditions

    Returns:
```

```
        lossBD (_type_): loss due to the boundary collocation points output estimation.
    """
    loss_bd = []
    for k in range(self.numcdt):
        Xcdt, FunCDT = self.CDT[k][:2]
        XcdtC = Xcdt.clone()
        XcdtC.requires_grad = True
        try:weight = torch.tensor(float(self.CDT[k][2]))
        except:weight = torch.tensor(1.)
        loss_bd.append(weight*FunCDT(XcdtC,self.forward(XcdtC)))
    return loss_bd
```

The core structure of the BC loss function resides in the object "CDT" which is defined by user in the training function. This object might at first be difficult to comprehend, but the architecture of this object was designed this way in order to give complete freedom to impose different conditions (CDT), not only relating boundary or limiting to Dirichlet conditions[5]. Retaking the example shown in Fig. 2.5, the CDT object for that case is:

$$CDT = \begin{matrix} [(X_\partial, & Func, & \alpha), \\ (X_\partial, & Func, & \alpha), \\ (X_\partial, & Func, & \alpha), \\ (X_\partial, & Func, & \alpha)] \end{matrix},$$

where X_∂ is the X tensor data points to predict using the PINN, $Func$ is a function to compute the loss of each BC and α is the weight optional parameter[6], each color refers to a different BC, meaning that the specified parameters are not equal although their name suggest so. $Func$ has a similar architecture of $ResPDE$, in which the inputs are X_∂ and the prediction of the PINN using this points, the only limitation here imposed is that it must return the value of the actual loss function, coded in the class as "criterion" property. This means that the user is completely free to impose any desired condition to the data points X_∂, including Dirichlet, Neumann and also, but not limited to, mixed boundary conditions. This lets the door open to use CDT as a way to include measurements in the training process[7], helping the convergence of the PINN model.

[5] as it was prior the inclusion of this object.

[6] In Xiang et al., 2021 this parameters are proposed also as NN parameters that adapt to enhance the learning

[7] if so, it most be taken with special care, cause this may cause over-fitting. In such cases it is recommended to limit the weight ($\alpha \in [1, 5]$) of the CDT.

3.3 Training Function

This is the most important function of the class, in contains the forward pass and the back propagation process. The input parameters it receive are the following:

- Xpde: collocating points,
- CDT=None: object that contains the conditions to impose,
- nEpochs=100: number of training epochs,
- history=True: save training history,
- Verbose=False: when True, enables the progress printing in the command line,
- nLoss=1000: how many iterations in between printing the current training loss, works only with Verbose=True,
- pScheduler=False: plot Scheduler history in the Loss history graphic,
- Test=[None,None]: [Xtest,Ytest] parameters to compare the training,
- nTest=10: how many iterations in between computing the test loss,
- PH=True: plot history, only in history=True (Plot History),
- SH=False: save detailed history, when multiple equations and condition it makes a separated plot for each class (Separated History),
- SB=True: save best prediction (Save Best),
- RE=True: rapid exist, enables a series of condition to finalize training before completing all the iterations (Rapid Exit).

The function starts recomputing the parameter RE, and resets in to False if there are not enough training epochs (nEpochs<100). It also compute the parameter "maxepochs" later to be use in the rapid exit algorithm, in also checks the conditions, and stops the training if their quantity is not in accordance with the previously specified by "numcdt". also in the first lines it initializes important properties such as CDT. The block of code is the following:

```
if nEpochs<100:RE=False
if nEpochs<= 30000:maxepochs=0.05*nEpochs
else:maxepochs = 1500
if self.HardConstrain is None and CDT is None:
    print('No conditions implemented.')
if CDT is not None:
```

```
    if len(CDT) != self.numcdt:
        print(CDT)
        raise TypeError('Number of conditions not matching, killing training process')
self.CDT = CDT
self.train_Xpde = Xpde.clone()
self.loss_history = {'total_loss':[],'total_pde_loss':[],'total_bd_loss':[]}
```

The next part of the function is in charge of the detailed history SH as well as initializing, if needed, the test, as well as the iterator and the clock to measure the time taken to train.

```
SH = SH and (self.numequ>1 or self.numcdt>0)
if SH:
    for i in range(self.numequ):
        self.loss_history[f'pde_loss_{i}'] = []
    for i in range(self.numcdt):
        self.loss_history[f'bd_loss_{i}'] = []
if Test[0] is not None and Test[1] is not None:
    self.loss_history['test_loss'] = []
    self.loss_history['test_iter'] = []

if self.scheduler is not None:
    self.schhist=None
iterator = range(nEpochs)
if Verbose: iterator = tqdm(iterator, bar_format='{desc}: {percentage:6.2f}% |{bar}{r_bar}')
t0 = time()
```

The following lines of code, unless stated otherwise are nested in a for loop of the form "for n in iterator". The first and main code block of the training loop if the following:

```
try:
    self.optimizer.step(self.closure)
    Loss = self.loss()
except:
    Loss = self.closure()
    self.optimizer.step()
```

where "closure" is the following function:

```
def closure(self):
    self.optimizer.zero_grad()
    Loss = self.loss()
```

```
    Loss.backward()
    return Loss
```

The architecture *try/except* is selected because some optimizers requires the "closure" function as an argument and others don't accept it. The rest of the function is basically a switch pattern to store the history and so on. The first if statement triggers when saving the current best-loss (the smallest) and the core can be understand pretty straight forward.

```
if SB and Loss.detach().cpu().numpy()<self.best_valid_loss:
    self.best_valid_loss = Loss.detach().cpu().numpy()
    if Verbose and n > 2*maxepochs:
        print(f"\nBest validation loss: {self.best_valid_loss:.3e}")
        print(f"\nSaving best model for epoch: {n+1}\n")
    torch.save(self.state_dict(),f'Models/{self.name}/Best/StateDict_{self.n}.model')
```

The next lock of code is for giving a step to the learning rate scheduler, if this option is selected, as well as, if requested, store the scheduler history later to be plotted. It also prints (when Verbose=True) an update on the Loss convergence and stores it.

```
if self.scheduler is not None:
    if pScheduler:
        if self.schhist is not None:self.schhist.append(self.optimizer.param_groups[0]["lr"])
        else:self.schhist = [self.optimizer.param_groups[0]["lr"]]
    try:self.scheduler.step()
    except:self.scheduler.step(Loss)
if Verbose and n%nLoss==0:
    print(f'\nloss at iteration {n}: {Loss:.3e}')
self.loss_history['total_loss'].append(Loss.detach().cpu().numpy())
```

The next *if/else* block is designed to store separately the loss generated by each PDE as well as each condition.

```
if self.numcdt>0:
    loss_pde = self.lossPDE(self.train_Xpde)#.detach().cpu().numpy()
    loss_bd = self.lossCDT()
    self.loss_history['total_pde_loss'].append(0)
    for respde in loss_pde:
```

```
        self.loss_history['total_pde_loss'][-1] += respde.item()
    self.loss_history['total_bd_loss'].append(0)
    for bd in loss_bd:
        if bd is not None:
            self.loss_history['total_bd_loss'][-1] += bd.item()
    if SH:
        for i in range(self.numequ):
            self.loss_history[f'pde_loss_{i}'].append(loss_pde[i].item())
        for i in range(self.numcdt):
            if loss_bd[i] is not None:self.loss_history[f'bd_loss_{i}'].append(loss_bd[i].item())
else:
    if SH:
        loss_pde = self.lossPDE(self.train_Xpde)
        for i in range(self.numequ):
            self.loss_history[f'pde_loss_{i}'].append(loss_pde[i].item())
```

The second to last if statement has per goal fo compute Test loss when needed

```

if Test[0] is not None and Test[1] is not None and n%nTest==0:
    # try:
        self.loss_history['test_iter'].append(n)
        self.loss_history['test_loss'].append(self.criterion(self.forward(Test[0]),Test[1]).detach().cpu().numpy())
```

And this last one is the Rapid Exit algorithm.

```
if n>0.2*nEpochs and RE and ((Loss<1e-4 and np.std(self.loss_history['total_loss'][-int(maxepochs):])/np.mean(self.loss_history['total_loss'][-int(maxepochs):])<1e-3) or np.std(self.loss_history['total_loss'][-int(1.5*maxepochs):])/np.std(self.loss_history['total_loss'][-int(1.5*maxepochs):])<5e-5):
    print(f'Model already converged at epoch: {n}, breaking trainning.')
    print(f'Last Loss: {Loss:.3e}')
```

```
    print(f'Std of last {int(maxepochs)} Loss: {np.std(self.loss_history["total_loss"][-int(maxepochs):]):.3e}')
    print(f'Std of last {int(1.5*maxepochs)} Loss: {np.std(self.loss_history["total_loss"][-int(1.5*maxepochs):]):.3e}')
    break
```

To finalize the training function, and outside the for loop, the clock is stopped and stored as well as, if requested, the loss history is plotted:

```
self.time = time()-t0
if PH:
    return self.PlotHistory(pScheduler,SH=SH)
```

Other functions of the class are shown in Appendix B.

ODEs Solving using PINNs

Now that the module is built and fully functional it is time for testing its capabilities of predicting and adapting to many different DP including types of differential equations as well as boundary conditions. In this chapter the firsts examples of DP are shown in order to test the module and start to comprehend the different optimal configuration and forms of approaching the problems. Thus all the section that are here presented are part of the results of this project. At first the interest is to get a first approach to the module usage, limiting the DP to ODEs or, at most, systems of ODEs.

4.1 A Simple ODE example

The DP that was selected to show as a first example is a simple second order ODE or the form:

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = -\pi^2 \sin(\pi t + \phi),$$

in the time domain $t \in [-1, 1]$. Analytical solution of the ODE is of the form:

$$x(t) = \sin(\pi t + \phi),$$

where x is called "displacement" due to the resemblance with the solution of the simple harmonic oscillator and ϕ is a phase that determines the "beggining" of the solution curve. Rather than for its difficulty, this DP was selected in order to test some of the different available optimizer algorithms supplied by PyTorch. there were defined 2 "BCs", one regarding the actual boundaries of the DP and the other as a data-feeding mechanism for improving the model convergence.

For each tested optimizer 25 PINNs were trained, each of them using the same training collocation points, as well as boundary testing points. Three hundred (N_f) collocation points were selected using Latin-Hypercube-Sampling and including two extra points fixed to the boundaries of the DP, fifty (N_u) points out of which were selected for data fitting the PINNs. For computing the loss of the predicted curve an array of 1200 evenly spaced time points was selected at which the PINN makes a prediction and also the analytical solution is computed, out of this two curves the testing loss (L_T) is computed.

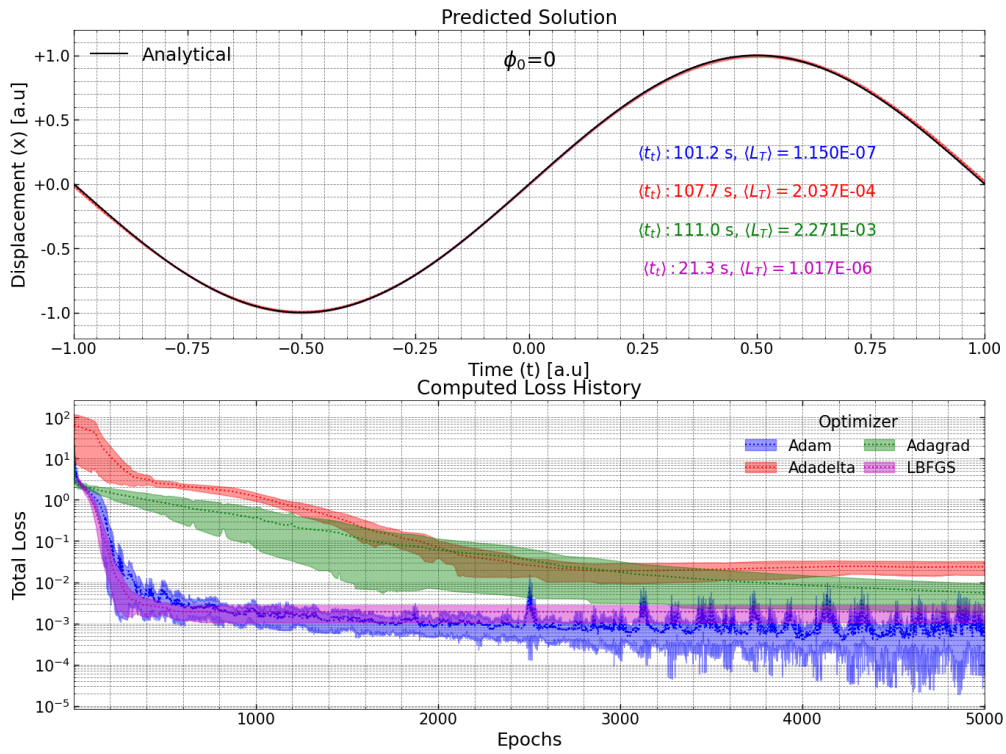

Figure 4.1: Training results, simple ODE DP with phase $\phi = 0$.

The training results are shown in Fig. 4.1, 4.2 & 4.3[1]. The all the PINNs were trained using the architecture $[1, 50 \otimes 2, 20, 50 \otimes 2, 1]$, in the predictive panel it is shown the average spent training time $\langle t_T \rangle$ as well as the mean teasting loss $\langle L_T \rangle$. Four different optimizer were tested between *Adam*, *Adagrad*, *Adadelta* and *LBFGS* for which mean and standard error are shown in the "Computed Loss History" panel of each figure.

Throughout the different "Computed Loss History" panels of the Figs. 4.1, 4.2 & 4.3 it is consistently found that there is a clear performance different between the optimizers *Adagrad* and *Adadelta* with the modern optimizers *Adam* and *LBFGS*, although is terms of time spent for the training it is clearly the *LBFGS* the fastest option. It is important to note that there is a consistent behavior in the *LBFGS* history loss curve which "stops learning" around at most 600 learning iterations, this is due to the set of arguments used, further explained in the example of Sec. 5.2, which is responsible of the very short training time required, although at the of a more computer-expensive calculations. It is crucial for the viability of PINNs the selection of a suitable optimizer for the training process, one that combines low training time in order to be suitable for replacing classical

[1]Although they can not be seen clearly, all the models predictions (color coded dotted lines), including their related error (color filled areas using the same color code of the line), are shown in the predicted solution panels.

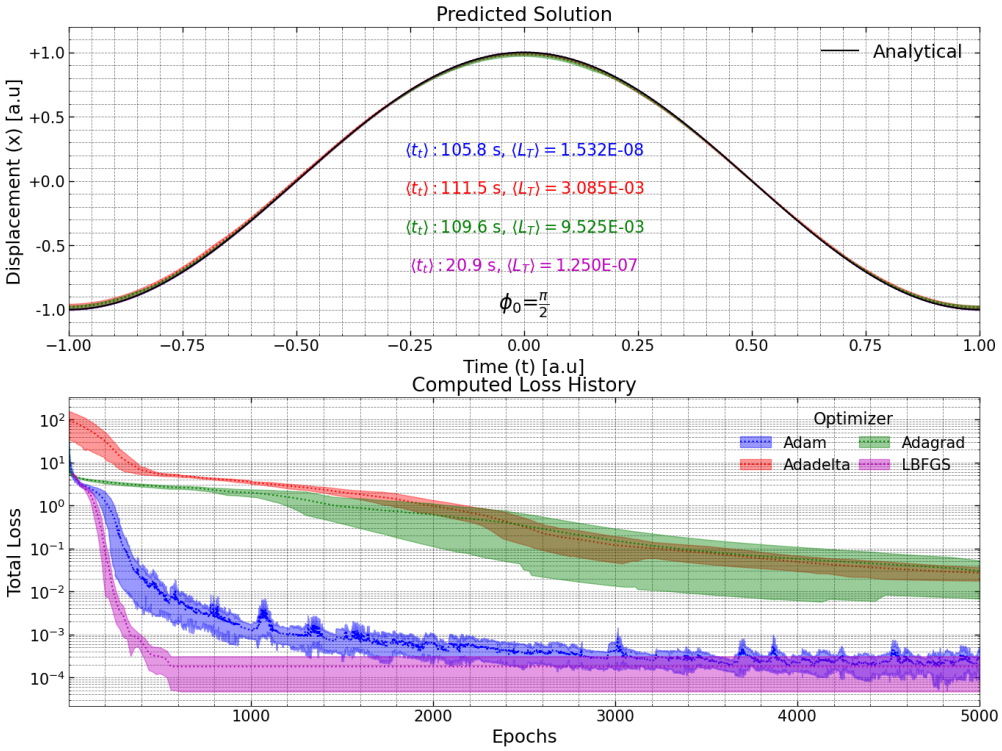

Figure 4.2: Training results, simple ODE DP with phase $\phi = \pi/2$.

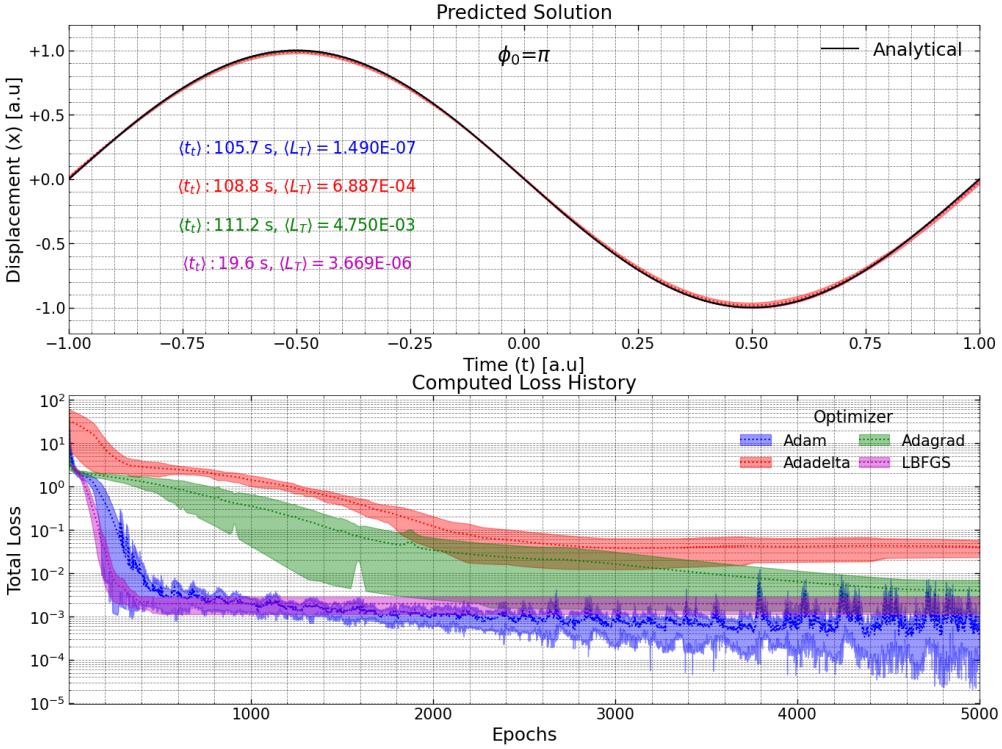

Figure 4.3: Training results, simple ODE DP with phase $\phi = \pi$.

computational algorithms but as well one that present a rapid convergence. It was also found that the threshold for determining the convergence of the modes if a best loss[2] of around $10^{-3} \sim 10^{-4}$ as shown in the "Predicted Solution" panels of each figure.

The ResPDE function, as well as the CDT used are:

```
'Loss Function'
def ResPDE(Tpde,Xpred):
    x_t = autograd.grad(Xpred,Tpde,torch.ones([Tpde.shape[0],1]).to(model.device), retain_graph=True, create_graph=True)[0]
    x_tt = autograd.grad(x_t,Tpde,torch.ones([Tpde.shape[0],1]).to(model.device), retain_graph=True, create_graph=True)[0]

    res = x_tt + (np.pi**2)*Xpred
    return [res]

'Exact Solution'
def Solution(t):
    return torch.sin(np.pi*t+phi0)

'Initial Condition'
def FCDT(Tcdt,Xcdt):
    return model.criterion(Xcdt,Solution(Tcdt))
```

4.2 Simple Harmonic Oscillator

As widely known, the simple harmonic oscillator is a fundamental concept in physics that plays a crucial role in understanding a wide range of phenomena. It describes a system that oscillates back and forth around an stable equilibrium position, such as a mass-spring system, a pendulum (up to some approximation) or even a circuit. The harmonic oscillator is of great importance because it serves as a foundation for more complex systems, contributing to the development of theories in classical mechanics, quantum mechanics, and many other branches of physics.

The ODE that describes the system is relatively simple, being a second order linear equation, which has straightforward mathematical solution. Lets say that our system is the mass-spring, in which the mass is displaced a distance $x = x_0$

[2] the one of the loss history, not the test loss

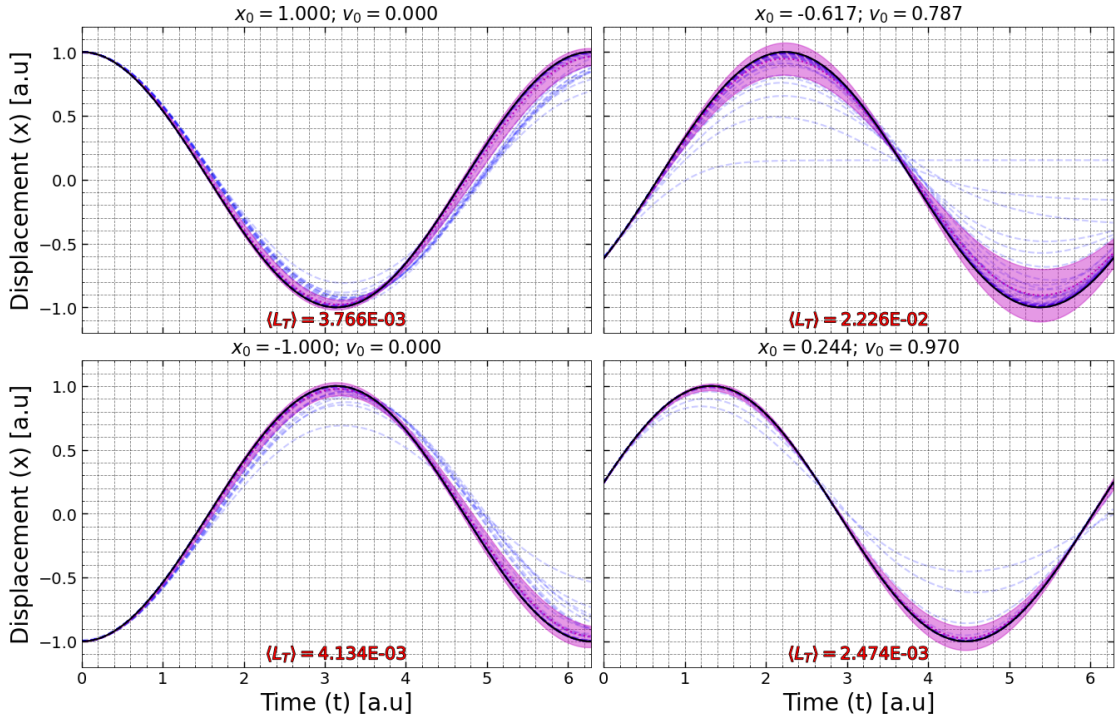

Figure 4.4: Each panel represent a DP with different initial conditions specified in the title.

and has a velocity $v = v_0$ at $t = 0$, with spring constant k and the mass of value m. This sets the DP as:

$$m\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + kx = 0 \quad x(0) = x_0,\ v(0) = V_0.$$

It is widely known that this system has as analytical solution the function:

$$x(t) = x_0\cos(\omega_0 t) + \frac{v_0}{\omega_0}\sin(\omega_0 t),$$

where $\omega_0 = \sqrt{k/m}$ is the angular frequency of the system. To solve the system the proposed architecture for the PINN is [1,40 $\otimes$ 3,1], with $\lambda = 0.075$, 2000 training steps and Adam as optimizer. The prediction for multiple initial values are presented in Figs. 4.4 & 4.5. In each panel of the figures the black continuous line represent the analytical solution for the DP, the average prediction (as well as a 1σ confidence interval) are presented in purple, computed out of 50 fully trained PINN, and as really light blue curves each of the PINN prediction.

The average loss of the trained models is $\langle L_T \rangle =$ and took to train an average of (40 ± 6) s. The ResPDE function, as well as the functions used to enforce the initial conditions are:

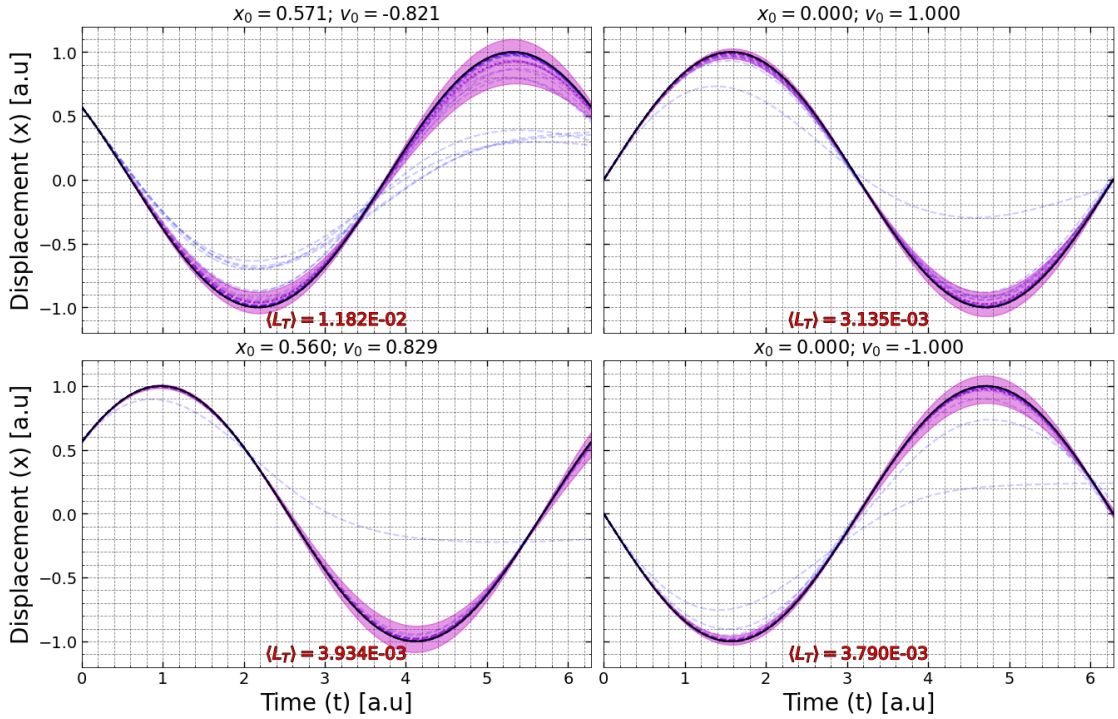

Figure 4.5: Each panel represent a DP with different initial conditions specified in the title.

```
'Loss Function'
def ResPDE(Tpde,Xpred):
    x_t = autograd.grad(Xpred,Tpde,torch.ones([Tpde.shape[0],1]).to(model.device), retain_graph=True, create_graph=True)[0]
    x_tt = autograd.grad(x_t,Tpde,torch.ones([Tpde.shape[0],1]).to(model.device), retain_graph=True, create_graph=True)[0]

    res = x_tt + w0*Xpred
    return [res]

'Initial Condition'
def InitialPos(Tcdt,Xcdt):
    x0 = X0*torch.ones(Xcdt.shape[0],1).to(model.device)
    return model.criterion(Xcdt,x0)
def InitialVel(Tcdt,Xcdt):
    v = autograd.grad(Xcdt,Tcdt,torch.ones([Tcdt.shape[0],1]).to(model.device),retain_graph=True, create_graph=True)[0]
    v0 = V0*torch.ones(v.shape[0],1).to(model.device)
```

```
    return model.criterion(v,v0)
```

4.3 Coupled ODEs

Coupled ODEs are a set of differential equations that are dependent on each other. Their importance lies in their ability to model complex systems and phenomena found in various disciplines, some examples in the field of fluid dynamics, modeling diseases or even population dynamics.

An example of such systems are the Lotka-Volterra equations, which are a pair of first order nonlinear differential equations of the form:

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} =& \alpha x - \beta xy,\\ \frac{\mathrm{d}y}{\mathrm{d}t} =& \delta xy - \gamma y.\end{aligned}$$

This model is used for describing the predator-prey interaction in nature. When rewriting the equations as follows:

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} =& x(\alpha - \beta y),\\ \frac{\mathrm{d}y}{\mathrm{d}t} =& y(\delta x - \gamma),\end{aligned}$$

we can then "combine" them obtaining the following equations:

$$\begin{aligned}\frac{\mathrm{d}y}{\mathrm{d}x} =& -\frac{y}{x}\frac{\delta x - \gamma}{\beta y - \alpha},\\ 0 =& \frac{\beta y - \alpha}{y}\mathrm{d}y + \frac{\delta x - \gamma}{x}\mathrm{d}x,\end{aligned}$$

which has as solution:

$$V = \delta x - \gamma \ln(x) + \beta y - \alpha \ln(y). \tag{4.1}$$

This equation describes a close loop solution to the equation around the non-trivial equilibrium point $(\frac{\gamma}{\delta}, \frac{\alpha}{\beta})$[3]. For training, the selected architecture is $[1, 25 \otimes 3, 20, 25 \otimes 3, 2]$ in order to give the PINN the sufficient non-linearity. There were trained 10 DP each of them differing from its starting point from $x_0 = y_0 = 0.9$ up to $x_0 = y_0 = 1.8$ in jumps[4] of 0.1 and parameters $\alpha = 2/3$, $\beta = 4/3$, $\delta =$

[3]the trivial equilibrium point is (0,0)

[4]The V values of each curve are computed from this initial values

1, $\gamma = 1$.[5] Each DP is trained two times, the first of then using $\sigma = Tanh$ and the other using $\sigma = Sin$ in order to compare the results obtained by changing the activation function.
Lets enumerate the curves with integers i from 0 to 9[6] this means that $x_{0_i} = y_{0_i} = 0.9 + 0.1 \cdot i$ and the corresponding $V_i(x_{0_i}, y_{0_i})$ computed using Eq. 4.1. The curve is then trained in a time range $t \in [t_{min} = 0, t_{max_i} = 20 + 10 \cdot i^{1.05}]$[7], with a number of collocation points $N_{f_i} = \lfloor 7500 \frac{t_{max_i}}{20} \rfloor$ and a number of training steps $steps_i = 10000 + \lfloor 10000 \frac{t_{max_i} - 20}{20 + 10 \cdot 9^{1.05}} \rfloor$. The collocation points are chosen in this and the rest of the DP with Latin-Hypercube-Sampling[8] (Stein, 1987; Loh, 1996).

The ResPDE function, as well as the initial condition enforcing function are shown bellow:

```
def ResPDE(Tpde,Fpde):
    xpde, ypde = Fpde.T
    xpde = xpde.view(-1,1)
    ypde = ypde.view(-1,1)
    x_t = autograd.grad(xpde,Tpde,torch.ones([xpde.shape
        [0],1]).to(model.device),retain_graph=True,
        create_graph=True)[0]
    y_t = autograd.grad(ypde,Tpde,torch.ones([xpde.shape
        [0],1]).to(model.device),retain_graph=True,
        create_graph=True)[0]
    f1 = x_t - alpha*xpde + beta*xpde*ypde
    f2 = y_t - delta*xpde*ypde + gamma*ypde
    f3 = delta*xpde - gamma*torch.log(xpde) + beta*ypde
        - alpha*torch.log(ypde) - V
    return [f1,f2,f3]

'Initial condition'
def InitialPos(Tcdt,Fcdt):
    xcdt, ycdt = Fcdt.T
    X0 = x0*torch.ones(Tcdt.shape[0],1).to(model.device)
    Y0 = y0*torch.ones(Tcdt.shape[0],1).to(model.device)
    res = model.criterion(xcdt,X0)+model.criterion(ycdt,
        Y0)
    return res
```

[5]Example taken from Wikipedia contributors, 2023, although firstly inspired to present it by the simulation shown in this tweet of G. Peyré

[6]notation inherit from python

[7]The period of the system can be computed using the formula estimateds by Waldvogel, 1983

[8]in particular its implementation in pyDOE library

It is important to note that in ResPDE the Eq. 4.1 is included under the pseudo $f3$. As in many other computational methods, it is important to take into account external conditions, not only just the PDEs to ensure a grate convergence of the PINN[9]. The results of each training are displayed in Fig. 4.6.

It is important to note that, the times used for training exceed the required time to close the computed loops, this means that when training the DP it completely stops, as seen in Fig. 4.6, this might be due to the weak time dependence of the system. Comparing the results obtained using $\sigma = Tanh$ and $\sigma = Sin$ it is seen that, except for the most inner curve, the rest of the DP present lower loss, form 4.8% all the way to 76.0%, it also helps with the curve stopping inconvenience. This result was expected following the findings of Sitzmann et al., 2020; Faroughi et al., 2022; Wong et al., 2022[10].

[9]this will also be important for future DP

[10]I would like to adhire myself to the iniciative of Sitzmann et al., 2020 and encourage the implementation of sinusoidal function as a base activation function in pytorch.

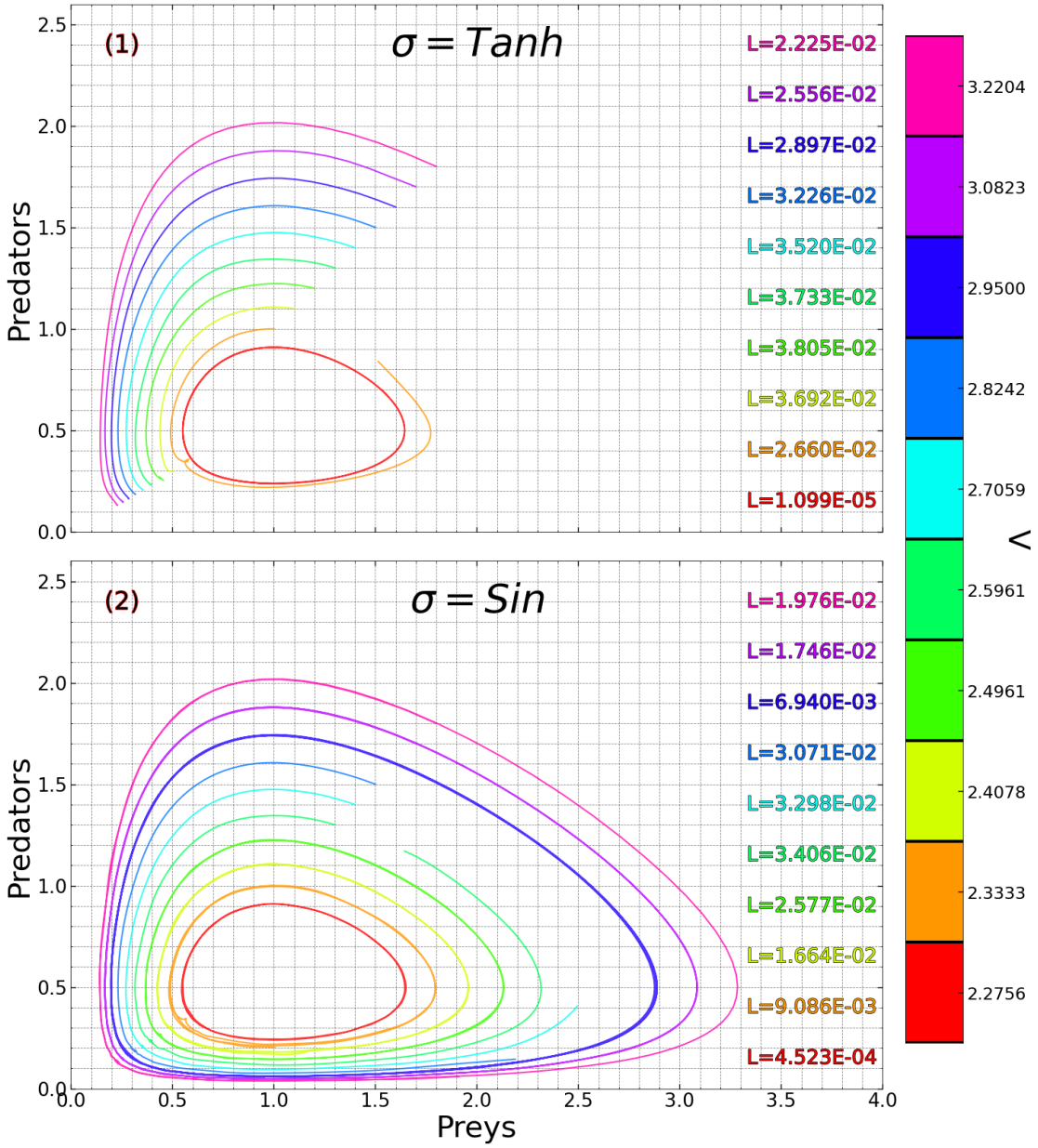

Figure 4.6: Results of PINN $[1, 25 \otimes 3, 20, 25 \otimes 3, 2]$ using as activation function (1)$\sigma = Tanh$, and (2)$\sigma = Sin$. Equi-potential curves where its color represents the constant value V used to compute them.

Solving PDEs using PINNs

Although their capability for solving ODE only DP, the field where PINNs come in handy is when solving DP involving PDE, where the systems are so complex there is not any analytical solution and the "traditional" computational methods are not efficient enough.

5.1 Diffusion Equation

As a first DP the diffusion equation was selected due to its simplicity as well as its already known solution, two cases were studied inspired by Lu et al., 2021a; Toscano, 2022. The first of them defined as:

$$\begin{gathered}\frac{\partial y}{\partial t} - \frac{\partial^2 y}{\partial x^2} + e^{-t}\left(\sin(\pi x) - \pi^2 \sin(\pi x)\right) = 0 \\ x \in [-1, 1],\ t \in [0, 1] \\ y(x, 0) = \sin(\pi x) \\ y(-1, t) = y(1, t) = 0 \\ y(x, t) = e^{-t}\sin(\pi x)\end{gathered}$$

An the second:

$$\begin{gathered}\frac{\partial y}{\partial t} - \frac{\partial^2 y}{\partial x^2} - e^{-t}\left(\frac{3}{2}\sin(2x) + \frac{8}{3}\sin(3x) + \frac{15}{4}\sin(4x) + \frac{63}{8}\sin(8x)\right) = 0 \\ x \in [-\pi, \pi],\ t \in [0, 1] \\ y(x, 0) = \frac{3}{2}\sin(2x) + \frac{8}{3}\sin(3x) + \frac{15}{4}\sin(4x) + \frac{63}{8}\sin(8x) \\ y(-\pi, t) = y(\pi, t) = 0 \\ y(x, t) = e^{-t}\left(\sin(x) + \frac{1}{2}\sin(2x) + \frac{1}{3}\sin(3x) + \frac{1}{4}\sin(4x) + \frac{1}{8}\sin(8\pi)\right)\end{gathered}$$

For both of them the selected PINN architecture is $[2, 32 \otimes 2, 1]$ with activation function $\sigma = Tanh$, initial learning rate $\lambda = 10^{-3}$ with learning rate scheduler $StepLR$ that, at training iteration 12500[1], multiplies the initial learning rate by 0.5. The results for both DP are shown respectively in Fig. 5.1 & 5.2, where the

[1] 5000 in the second case

error is computed as $err = 2\frac{y_{real}-y_{pred}}{1+|y_{real}|}$[2] which is introduced as the rule to measure error through all the following examples.

It can be seen in the error panel of both figures (Fig. 5.1 & 5.2) that the PINN predicted solution starts to differ from the real solution the further apart from $t = 0$, this can be due to the lack of constrains at this epochs, meaning that the inclusion of real-value (obtained with the analytical solution or other first approach computational methods) data can be of help to the model convergence. In Fig. 5.3 a comparison between the the convergence of the model is displayed, it can be seen that, for the first DP with learning rate variying at 12500 the loss history curve presents a slightly quicker convergence although a lot more fluctuations at "early" [5000,12500] and late [13000,$\cdots$] epochs in comparison to the second DP.

5.2 One-Dimensional Burgers Equation

This DP is a fundamental PDE that appears in fluid mechanics, nonlinear acoustics, gas dynamics and even traffic flow problems (Caldwell, Wanless, and Cook, 1981). The form of the PDE is:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2},$$

with diffusion coefficient (also known as kinematic viscosity) $\nu = 0.01/\pi$.[3] Let's define the domain with $x \in [-1, 1]$ and $t \in [0, 1]$ with boundary and initial conditions:

$$u(x, 0) = \sin(\pi x) \qquad u(-1, t) = u(1, t) = 0$$

The solution for this DP was computed using traditional computational methods, and the file was taken from Bhustali, 2021. The PINN architecture selected was $[2, 20 \otimes 7, 2]$ with *LBFGS* optimizer using the following arguments:

```
argsoptim = {'max_iter':20,'max_eval':None,'
   tolerance_grad':1e-11,
            'tolerance_change':1e-11,'history_size':100,
               'line_search_fn':'strong_wolfe'}
```

[2]see Appendix C for contextualization of the behavior of the selected error function across different relations of y_{real} and y_{pred}.

[3]This example was extracted from Toscano, 2022; Bhustali, 2021; Raissi, Perdikaris, and Karniadakis, 2017a

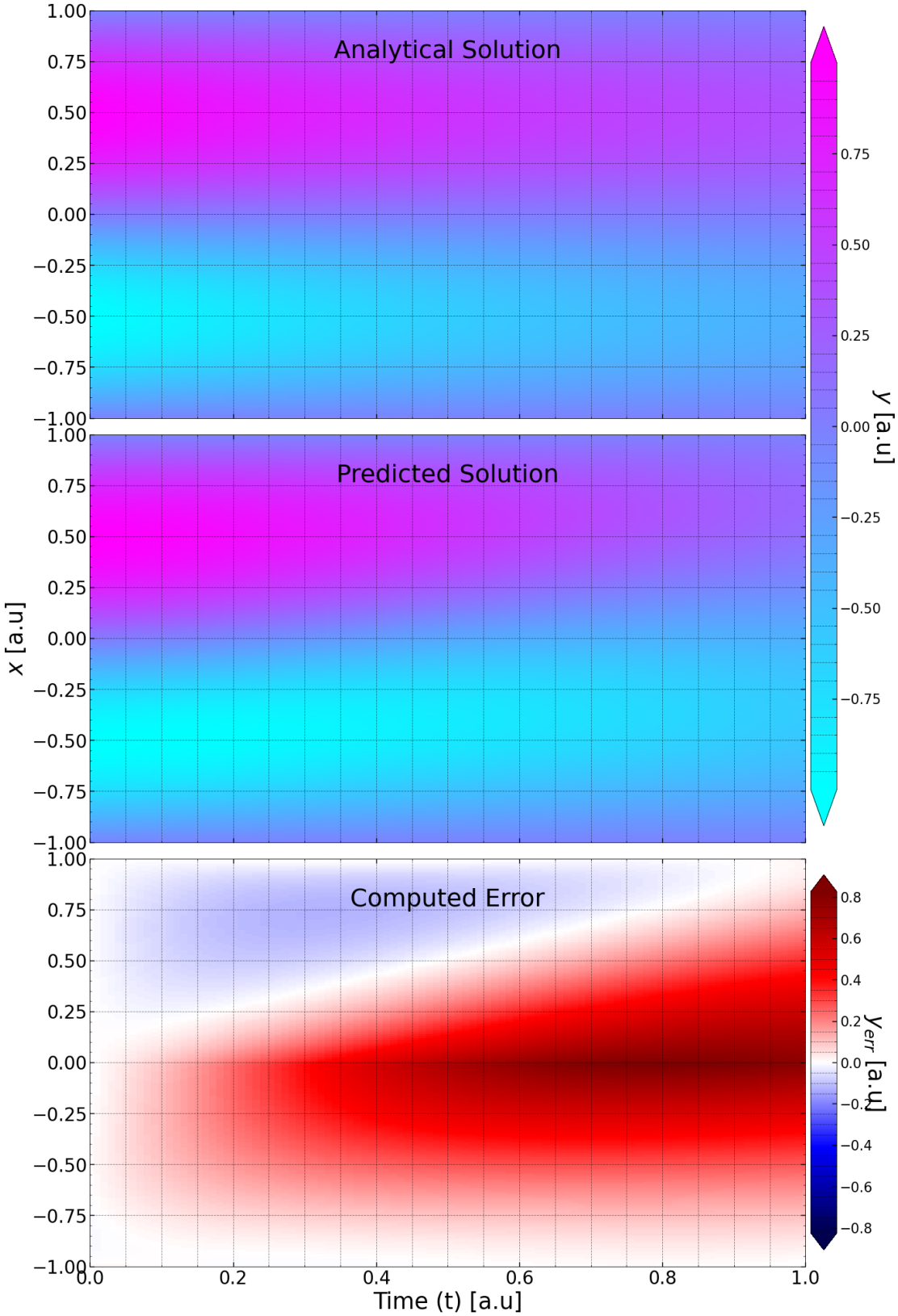

Figure 5.1: Diffusion equation solution for first proposed DP.

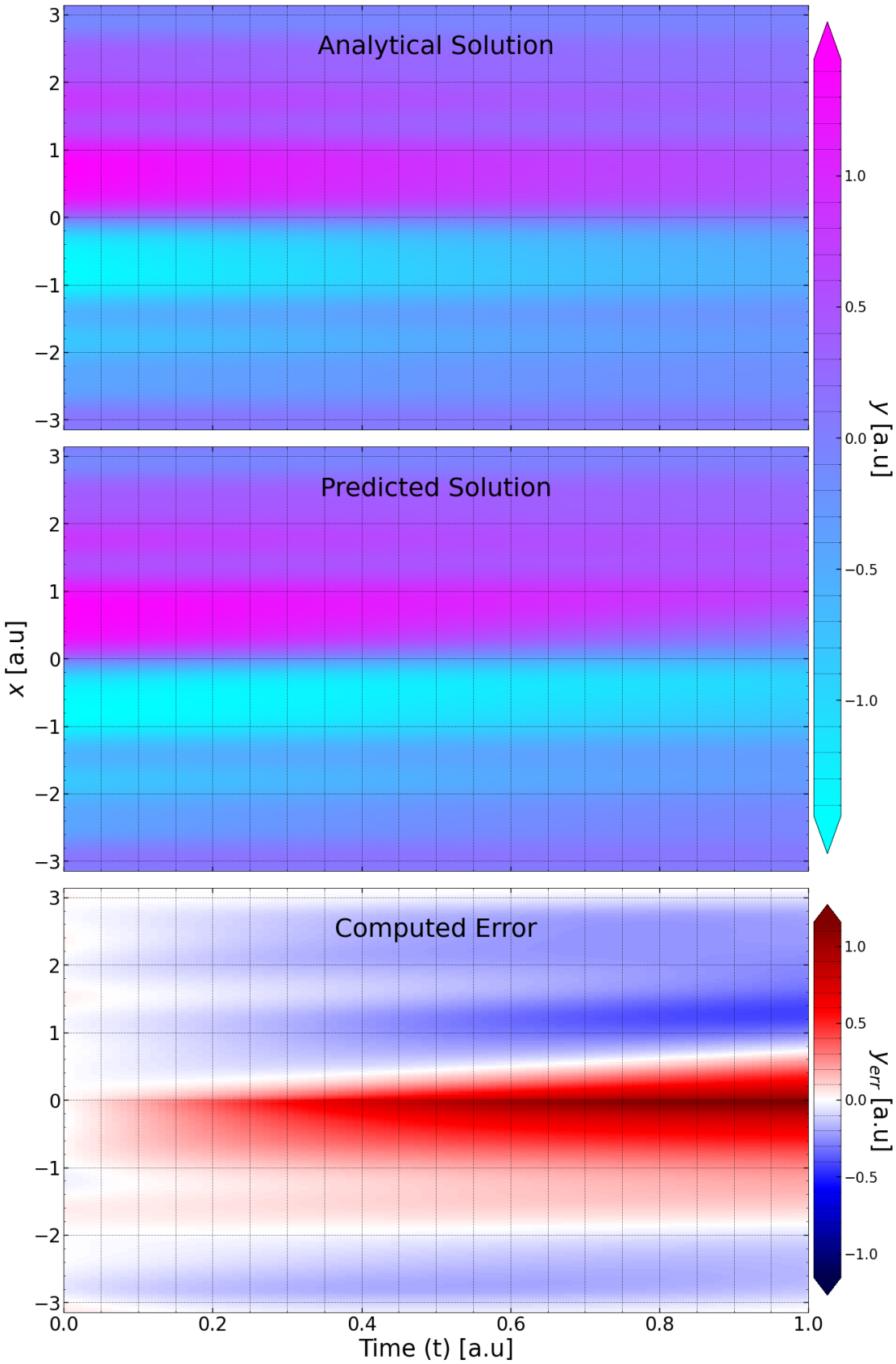

Figure 5.2: Diffusion equation solution for second proposed DP.

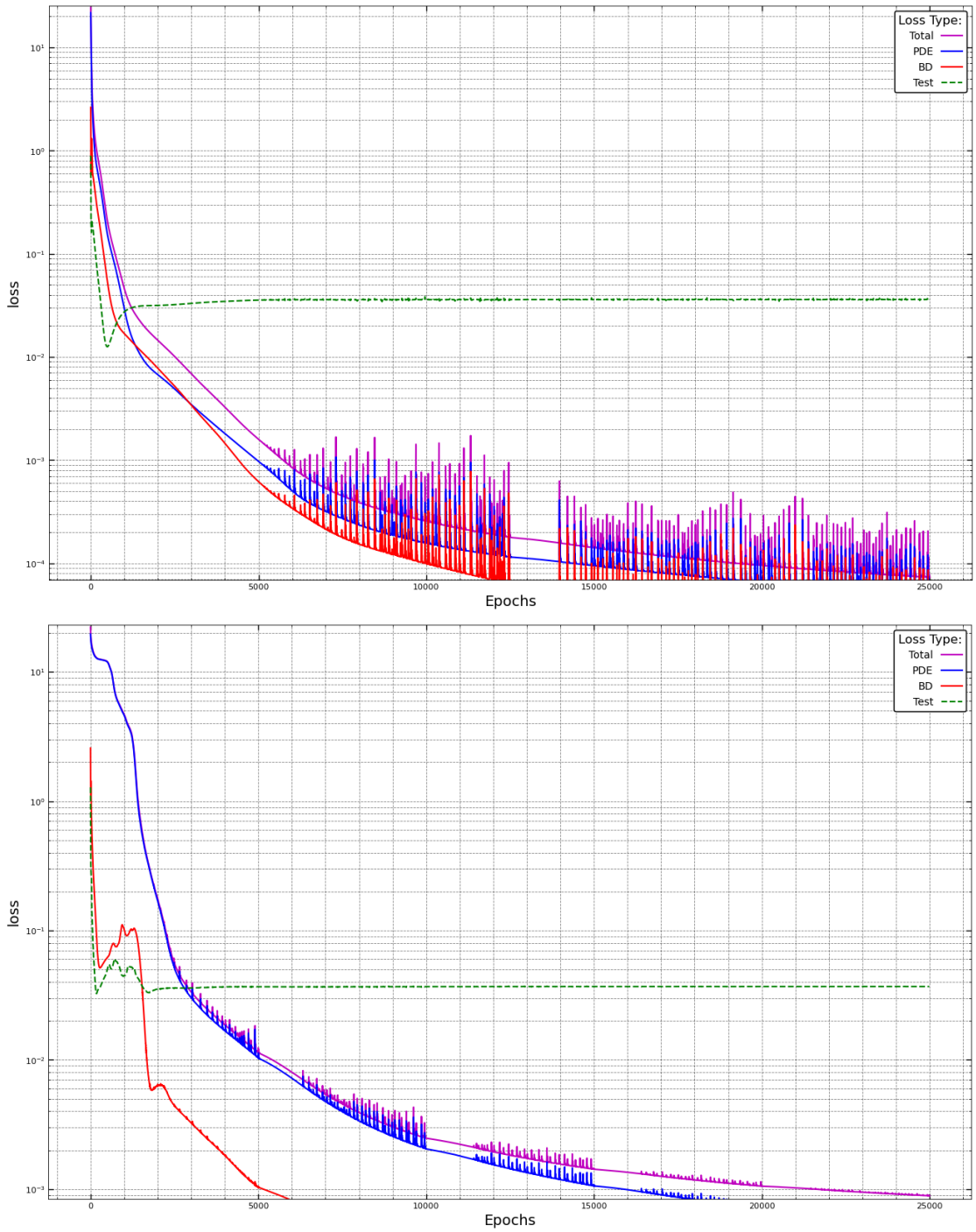

Figure 5.3: Comparison of convergence of both diffusion DP, the order of the panels is in accordance to its introduction in the text.

Another interesting approach, if not noted by now, is the selection of 2 outputs for the NN instead of just one, this was done because it helped with the convergence of the model[4]. The three conditions presented before were also combined as one, and then only selected a limited number of data points $Nu = 100$, the number of collocation points is $Nf = 10000$. The implemented learning rate $\lambda = 0.1$. The results of the training are shown in Fig. 5.4. As it can be appreciated in the discrepancy panel of the mentioned figure, there exist a great agreement between both methods used for estimating the solution, with the only great discrepancy found located in the "boundary" zone of $x = 0$ for $t > 0.2$. The time spent training the PINN for one thousand steps is under 10 minutes, finding convergence $\mathcal{L} \sim 10^4$ at 200 training steps[5].

Other PDE DP examples of solutions using PINNs can be found in Appendix E.

5.3 Kovasznay Flow

A great field of application for PINNs is fluid dynamics. Typically the systems that it study are complex, with no analytical solution. The afore mentioned help the development of the computational fluid dynamics to supply, mostly the industry, with accurate simulation of the behaviour of objects under fluids or fluids itself, such as airplane and car aerodynamics, ventilation, heat transfer and thermal management (such as electronic cooling in PCs), even blood flow simulation. Fluid dynamics is governed by the Navier-Stokes equations, which depending the fluid characteristics can be written in different forms. Limiting the DP to a two dimensional incompressible flow, the governing equations are:

$$\begin{aligned} \boldsymbol{\nabla} \cdot \mathbf{v} =& 0, \\ \frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \boldsymbol{\nabla})\mathbf{v} =& -\frac{1}{\rho}\boldsymbol{\nabla} p + \frac{\mu}{\rho}\nabla^2 \mathbf{v} + \mathbf{g}, \end{aligned}$$

where $\mathbf{v} = (u, v)$ is the velocity vector, p is the pressure, μ is the viscocity of the fluid, ρ its density and $\mathbf{g}$ is the gravitational acceleration, following the formulation used by Rao, Sun, and Liu, 2020 & Zhang et al., 2021 the last equation transform as:

$$\begin{aligned} \frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \boldsymbol{\nabla})\mathbf{v} =& \frac{1}{\rho}\boldsymbol{\nabla} \cdot \sigma + \mathbf{g}, \\ \sigma =& -p\mathbf{I} + \mu(\boldsymbol{\nabla}\mathbf{v} + \boldsymbol{\nabla}\mathbf{v}^T), \end{aligned}$$

defining the Cauchy stress tensor σ, which may be also expressed as:

[4] the explanation for this is still out of my comprehension

[5] moment after which the optimizer stops training

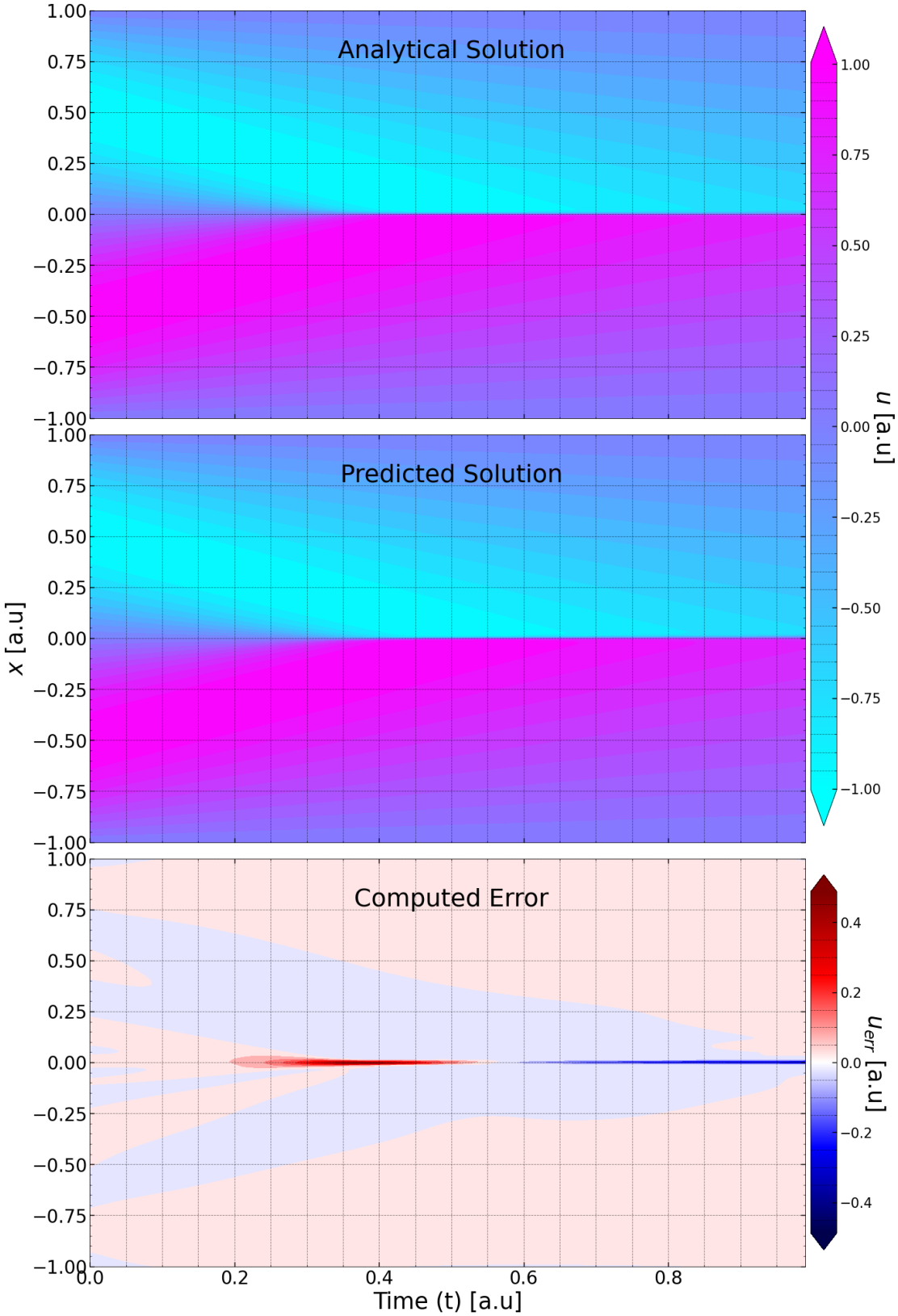

Figure 5.4: Burger equation results estimated by PINN compared to the ones computed using traditional computational methods.

$$\begin{bmatrix}\sigma_{11} & \sigma_{12}\\ \sigma_{21} & \sigma_{22}\end{bmatrix} = \begin{bmatrix}-p & 0\\ 0 & -p\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} + \mu\begin{bmatrix}0 & \tau_{12}\\ \tau_{21} & 0\end{bmatrix},$$
$$\frac{\partial \sigma_{ij}}{\partial x_j} = \frac{\partial}{\partial x_j}\left[-p\delta_{ij} + \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)\right].$$

Taking into account the properties of the Cauchy stress tensor, the following conditions are obtained:

$$\sigma = \sigma^{dev} + \sigma^{hyd}, \qquad -p = \sigma^{hyd} = \frac{1}{2}tr(\sigma)/$$

For decreasing the number functions the PINN needs to predict we introducing as well the stream function ψ as in Kovasznay, 1948; Okada39, 2020[6] that satisfies:

$$u = \frac{\partial \psi}{\partial y}, \qquad v = -\frac{\partial \psi}{\partial x}.$$

In summary, the PDE of the DP are:

$$u = \frac{\partial \psi}{\partial y}, \qquad v = -\frac{\partial \psi}{\partial x},$$
$$\rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) = \frac{\partial \sigma_{11}}{\partial x} + \frac{\partial \sigma_{12}}{\partial y}, \quad \rho\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) = \frac{\partial \sigma_{12}}{\partial x} + \frac{\partial \sigma_{22}}{\partial y},$$
$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \qquad \sigma_{12} = \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right),$$
$$\sigma_{11} = -p + 2\mu\frac{\partial u}{\partial x}, \qquad \sigma_{22} = -p + 2\mu\frac{\partial v}{\partial y},$$
$$p = -\frac{\sigma_{11} + \sigma_{22}}{2},$$

where the equations in blue are the aforementioned definitions and therefore will not be used in training, just for transformation purposes. This means that the output functions of the PINN are $[\psi, p, \sigma_{11}, \sigma_{22}, \sigma_{12}]$.

5.3.1 Flow around cylinder

The first DP to consider is the flow generated around an obstacle, in particular a circular cylinder. This DP is proposed, among many other references, in Rao, Sun, and Liu, 2020, Zhang et al., 2021 and even as an example in Lu et al., 2021a documentation. The domain is set to the rectangle $(x, y) \in [0, 1.1] \times [0, 0.41]$ with the cylinder centered in $(0.2, 0.2)$ with radius $r = 0.05$. The boundary conditions of the problem can be summarised in:

[6] It was not posible to fine the name of the author

$$
\begin{aligned}
u(x,y) =& v(x,y) = 0 && \forall (x,y) \in WALL, \\
p(1.1,y) =& 0 && \forall (1.1,y) \in OUTLET, \\
v(0,y) =& 0 && \forall (0,y) \in INLET,
\end{aligned}
$$

where the $INLET$ points are defined as the red points of Fig. 2.5, the $OUTLET$ points as the yellow and the $WALL$ as the blue and green of the same figure. The decision of increasing the number of collocation points around the obstacle is to properly capture the behaviour of the fluid around it, this include the enforcing of the no-splip (Wang et al., 2021) boundary condition there defined. The DP will be studied both considering the evolution of the system and not, including or excluding, depending the case, the time dependent terms os the Navier-Stokes equation. Additional boundary conditions will be defined in each case. In both cases a HardConstrain was used in order to enforce that the pressure p is non negative.

Time Independent Solution

Ignoring the time dependent terms of the above mentioned PDEs (the derivatives respect to time of both velocity components), we obtain the time-independent Kovasznay formulation. Including the boundary condition:

$$
u(0,y) = \frac{4y(0.41-y)}{0.41^2} \quad \forall (0,y) \in INLET,
$$

the DP is now fully defined. The architecture selected to tackle the problem solving is a $[2, 50 \otimes 8, 5]$ FCN. The predicted solution computed using finite element methods, as well as the PINN prediction are shown in Fig. 5.5[7] as well as the found discrepancy between them.

Taking as truth the results found using finite element simulation methods it can be observed that in both velocity components plots that the PINN produces an over estimation of the values, this can be perfectly seen around the cylinder obstacle. The major discrepancy found involves the pressure of the fluid, where the finite element method predicts greater values specially close to the inlet of the fluid. It is difficult to state a reason of this discrepancy, due to the close relation of the speed magnitude and the pressure level, although one of them may be that pressure is only constraint in the outlet of the flow.

Among many interesting facts found in Fig. 5.5, there most be noted the successful imposition of the defined boundary conditions as shown in each of the discrepancy panels of the figure. In particular, the one regarding the enforcing of the non-splip condition around the cylindrical obstacle was a complete

[7] although looking like an ellipse, the obstacle is a circle, this effect is generated by the difference in scale between the plotted axis, which was used to for a better looking image.

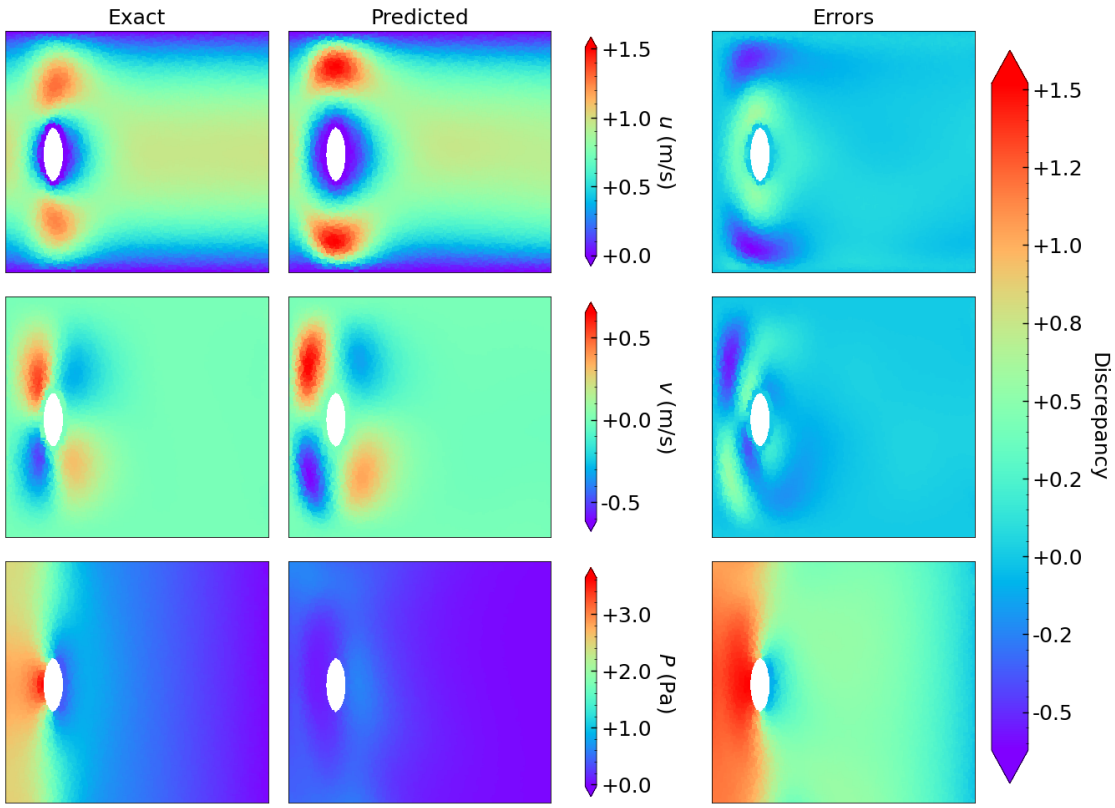

Figure 5.5: Time independent Kovasznay flow PINN prediction and comparison with exact solution which was taken from Rao, Sun, and Liu, 2020

success. It most be noted that the process in order to enforce this condition was to "set" each component of the fluid velocity, which in the present example works perfectly due to the absence of movement in the obstacle, although a better approach might be the one used by the tradition computational methods which computes the tangential con perpendicular components of the fluid velocity and "sets" them respectively to the velocity of the object and the zero, this could be accomplished by computing this components using the collocation-point (x, y) position. In Tab. 5.1 the computed discrepancy percentage found[8] comparing the PINN-predicted results to the analytical solution as well as the imposed boundary conditions.

Time Dependent Simulation

Now taking into account the time dependent term and changing the inlet condition for:

[8]the values where computed using the mean of the absolute value of the discrepancy values found using the error function

Table 5.1: Average found discrepancy for the time-independent fluid interaction with an obstacle DP.

CDT \ Func	u [%]	v [%]	P [%]
Total	8.24	5.77	53.94
INLET	0.94	0.39	–
UPPER WALL	0.16	0.32	–
LOWER WALL	0.17	0.32	–
CYLINDER	0.42	0.35	–
OUTLET	–	–	1.03

$$u(0,y,t) = \frac{4y(0.41-y)}{0.41^2}\left[sin\left(2\pi t + \frac{3\pi}{2}\right) + 1\right] \quad \forall(0,y,t) \in INLET \times [0,0.5]$$

where as suggested in the last equation, the time is simulation time is defined in the interval $[0, 0.5]$. Appart from the previous mentioned, all the DCT remains the same, and in terms of PINN architecture the selected is $[3, 50 \otimes 7, 5]$. The simulation results are presented[9] in Fig. 5.6.

As expected, the flow speed, as well as its pressure, evolves increasing as the simulation time elapses. It is also seen that the reached magnitudes of the simulation are greater than the predicted in the time independent example, this causes a greater "vacum" space after the obstacle.

It can be Observed in the evolution presented in Fig. 5.6 that the found flow is stationary, as predicted by the time independent example of Fig. 5.5, also as expected a symmetry is found between both sides of the obstacle. It was also seen that the boundary conditions were conserved even with the time evolution, with an special impact of the no-slip conditions for interacting with the obstacle. This result is crucial, because is demonstrate the viability of the PINN technology in an industry such as aircraft design for estimating the results they could find when testing a wing profile design in a wind tunnel. For non static cases it is important to apply the changes previously mentioned in the non-slip condition. In Tab. 5.2 the average discrepancy of the predicted results[10] for the boundary conditions is presented, reinforcing the previously mentioned capabilities of the technology.

Compared to the results presented in Tab. 5.1 there was an increase in the discrepancy between the imposed boundary condition values and the predicted

[9]Partially presented, the simulation was computed in time steps of 0.01, but only some frames are shown

[10]The total loss was not computed due to the lack of a reference analytical solution.

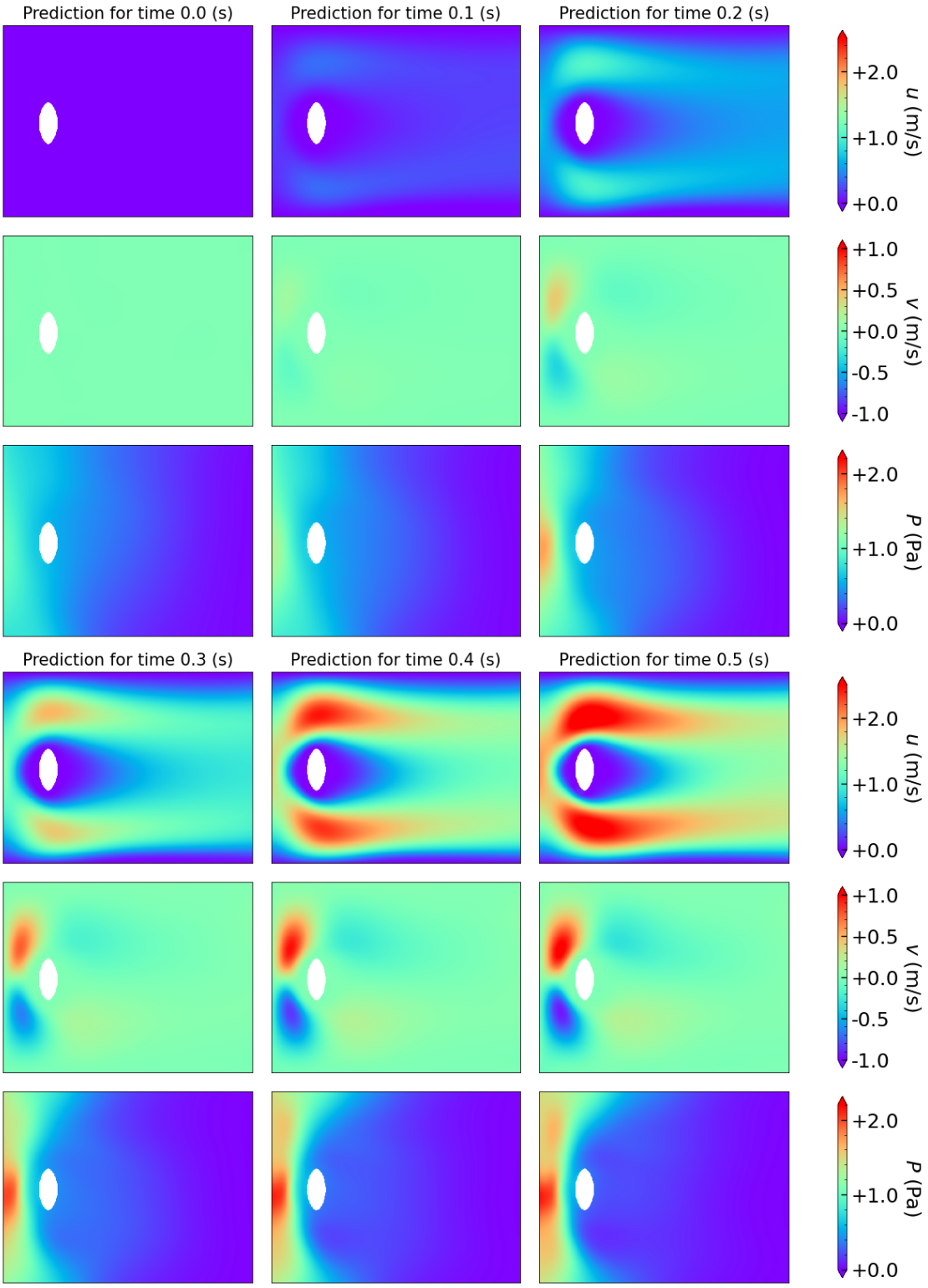

Figure 5.6: Kovasznay flow time evolution simulation obtained by PINN prediction. The color scale is maintained through the different panels.

Table 5.2: Average found discrepancy for the time-dependent fluid interaction with an obstacle DP.

CDT \ Func	u [%]	v [%]	P [%]
INLET	2.56	1.24	–
UPPER WALL	0.91	1.74	–
LOWER WALL	0.51	1.64	–
CYLINDER	1.35	0.75	–
OUTLET	–	–	2.14

with the PINN, this can be explained by the training of the present DP, it requires considerably more computational resources that were not available, this led to a reduction in the architecture complexity[11] as well as the inclusion of a third independent variable (time), caused a decrease in the density-per-epoch of the X_CDT points, also for reducing the computational requirements for training the model. In Appendix D the discrepancy graphics for each boundary condition for both fluid-obstacle interaction DPs are shown.

5.3.2 Cavity Flow

Applying now the Kovasznay flow formulation of the Navier-Stokes equations to the DP of a flow inside a square cavity $(x, y) \in [0, 1] \times [0, 1]$ in which the upper most side there is a constant flow at speed $u_{lid} = 1\, m/s$[12]. The time independent DP will be studied, considering as CDT only the previously mentioned lid condition as well as the "no slip" conditions in the other walls, which means that the velocity of the fluid relative to the wall is zero. The selected architecture is $[2, 45 \otimes 8, 5]$. The selected optimizer is $LBFGS$ with the parameters as in Sec. 5.2. For the time independent DP the predicted solution is shown in Fig. 5.7.

As expected following the results presented by Wong et al., 2022, the prediction shows a central vortex in the fluid, as well as steady corners at the bottom of the square. The time dependent cavity flow DP is part of the Appendix E.3.

[11] the connections of the time-independent PINN are 45^8 while the time dependent 50^7 ($45^8 > 50^7$)

[12] this example is extracted from Wong et al., 2022; Chiu, 2018

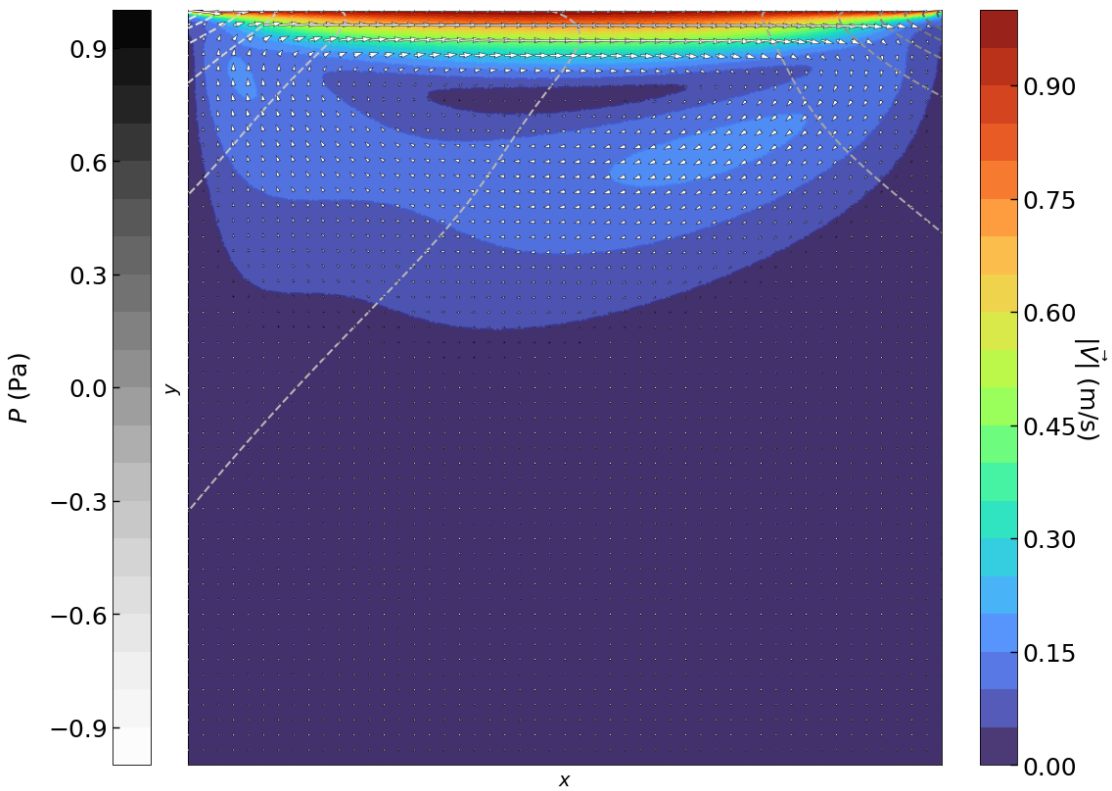

Figure 5.7: Predicted PINN solutions where the contour plot represents the magnitude of the total velocity $\left|\vec{V}\right| = \sqrt{u^2 + v^2}$, the white arrows represent the direction and magnitude of the flow and the grey scaled non-continuous lines represent the equipotential pressure contour lines.

Future Work

As shown by the different application examples presented in this document, PINNs are a powerful and revolutionary tool in scientific and engineering research that has come to the scene in recent years due to the powerful combination they take advantage of, mixing NNs training with physics based constrains (in the form of DPs) to further enhance their predictive capabilities when dealing with physics models. There are many areas in which the work done for this project can be improved, and I would like to enumerate them in order to help and promote future students to tackle them, continuing with the here made progress.

PINNs can be used not only for predicting the functional behavior of a DP that has a set of parameters as for example the angular frequency in the harmonic oscillator problem[1], but also to, based on a group of "experiment-based" data points, predict the parameters of the DP[2]. This is a complete area of PINNs that was neither studied in the present project nor included in the developed module. Although not requiring an enormous quantity of software adaptation, because the required architecture coincides with the one already developed, incurring in this area of PINNs comes with the necessity of a better data handling mechanism, including the possibility of error handling[3] of the data at the moment of training, forcing the opening maybe of a new function to include in the "loss" function called "lossDAT" that handles the above mentioned case, instead of the utilization of "lossDCT" to include them as used in this iteration of the module. This "lossDAT" function will also be of great use in the direct problems specially if error handling capabilities are included. An example of this type of PINNs is presented in Raissi, Perdikaris, and Karniadakis, 2017b

Another interesting feature to include, regarding also the "loss" function, is the possibility of adaptive weighting α parameters for training controlled either by the user or the PINN itself as presented in Xiang et al., 2021. This will open the possibility of dedicated training time for the different conditions the DP requires while also helping the model convergence as presented in the previously mentioned article. Other interesting possibility is to include adaptive activation

[1] this mode of operation of PINNs is called *direct problem*.

[2] called *inverse problems.*

[3] This will then mean that it is necessary to introduce data classification in theoretical and experimental.

functions as shown in Jagtap, Kawaguchi, and Karniadakis, 2020.

Other possible area of development for the module is the inclusion of, if possible user defined, diverse NNs architectures as the ones listed in Sec 3.6 of Yadav, Yadav, and Kumar, 2015. In particular this feature is inspired by the implementation of a convolutional NN architecture in order to solve the Burgers equation presented in Ren et al., 2022b[4]. It is possible also to include specific DP based architecture as predetermined sub-modules that includes pre-defined physical conserved quantities as for example the energy conserving PINNs for Hamiltonian defined DPs presented by Desai, 2021.

It is also a great option to take as a next step of this work to apply the developed current state developed module, although much better if the above recommendations are already included, to many other different complex physical systems, in particular, but not limited to, in the field of fluid dynamics[5], at which it has shown great potential in many literature implementations (e.g. Rao, Sun, and Liu, 2020; Zhang et al., 2021; Lu et al., 2021b; Raissi, Perdikaris, and Karniadakis, 2019 among many others). Other interesting fields are quantum mechanics with the Schrödinger equation (e.g. Shah et al., 2022; Wang and Yan, 2021) which can be applied to many fields, but with a particular great use in complex-material simulations, a field that is now lead by the computationally-expensive Density Functional Theory simulations (e.g. Lemay, 2023 & Nadeau and Antonius, 2023) using software like ABINIT (Gonze et al., 2020). PINNs present also great potential in outside physics implementations such as in economy, biology or even medicine (e.g. Bai, Chaolu, and Bilige, 2022; Yang, Zhang, and Karniadakis, 2020).

A whole new field of DP solving using NN can be opened implementing operator learning, architecture known as DeepONet(Lu et al., 2021b; Goswami et al., 2022). This type of architecture is also held by a universal approximation theorem and is not much different of the one implemented for PINNs, only being a two branch network. This architecture can lead to a whole new module creation. A great example of first step is shown in Toscano, 2022.

Another interesting, newly born, has arise in the recent years thanks to the development and improvements done in the field of quantum machine learning. As shown by Markidis, 2022 & Vadyala and Betgeri, 2023 quantum-enhanced PINNs (QPINNs) inherit the benefits of quantum-driven machine learning conserving the advantages of PINN model-training. It must also be stated that, although QPINNs, due to the quantum technology, are faster, it comes at the price of reliability due to their current state of quantum computing technologies.

[4]link to the github site of the mentioned code Ren et al., 2022a.

[5]or any other field that is at the moment massively computing driven.

Conclusions

As shown in Chapters 4 & 5, the constructed python module can be easily adapted for solving multiple DP with no restrictions about boundary conditions neither number of equations nor variables, meaning that it is sufficiently general to be considered a suitable tool for solving a wide range of DP.

Across the different benchmarked DP, it has been found that a great convergence is reached when the PINN present a total loss in the order of 10^{-4}. Although this does not necessarily imply that the prediction will be close enough to the analytical result, as shown in Figs. 4.1, 4.2, 4.3, 5.1, 5.2 and specially 5.3, where it is clearly shown that the Test loss is stagnated at $\sim 5 \cdot 10^{-2}$ even when the total loss keeps decreasing an reaching "well converged values".

As per the application of PINNs for ODEs solution it has been found that they offer a great alternative to traditional computational algorithms, but as mentioned in Chapter 6, it is necessary to further implement conditions and properties to the PINN to avoid results as the shown in Fig. 4.6.

It was also confirmed in Figs. 4.1, 4.2 & 4.3 that more "modern" optimizers such as Adam or LBFGS are a much better selection when training a PINN because they give a faster convergence to the results. In particular LBFGS is found to be a better option compared to Adam due to its extremely low required training time, although at the price of more aggresive resource consumption.

Through the tested DP it was found, specially in the examples shown in Appendix E, that as in many other applications of machine learning there will be difficulties with the utilization of PINNs in many different DP that require a bast knowledge in the topic for improving the problem adding possible new conditions regarding for example symmetries of the model or even hidden conserved quantities as well as a lot of experience using PINNs for correctly deciding its architecture, which includes number of layers, neuron per layer, activation function, even loss function[1]. The importance of the chosen architecture is remarked in the example of Fig. 4.6, where the usage of periodic activation functions helps

[1] in particular this feature of the module was not changed, but there exists other interesting loss functions as Huber or even other interesting techniques such as the one presented by Bullwinkel et al., 2022 to be applied in PINNs.

with the convergence of the PINN to the real solution or for example by the diffusion equations examples where the same PINN architecture is not capable of adjusting in equal magnitude to the two tested DP.

As per the main DP tested, the Kovasznay flow, specially the related to the fluid interaction with an obstacle, compelling evidence was found in Figs. 5.5, 5.6 & 5.7 as well as the Tabs. 5.1 & 5.2 that the usage of PINNs in such systems is a viable alternative to traditional methods for generating such simulations, although an improved version of PINNs shall be used, in order to reduce for example the computational requirements and training time, as well as to help the model convergence.

The repository of the present work can be found in the present link[2].

[2]at the moment of publication of this project the repository is still been worked at.

Appendix A: Activation Functions

The mathematical definition of the activation functions shown in Fig. 2.3 are presented in Tab A.1. (Yadav, Yadav, Kumar, et al., 2015; Jadon, 2018)

Table A.1: Mathematical definition of commonly used activation functions in NNs.

Function	Mathematical Definition
Sigmoid	$\frac{1}{e^{-x}+1}$
Tanh	$tanh(x)$
Step	$\begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$
ReLU	$\begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$
Leaky ReLU	$\begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases}$
ELU	$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$
Linear	$b + a \cdot x$
Gaussian	$\exp\left(-\frac{\lVert x-a \rVert}{2\sigma^2}\right)$
MultiQuadratics	$\sqrt{\lVert x-a \rVert^2 + b^2}$

Appendix B: Other functions functions of the module.

B.1 Forward Pass

Code of the forward pass step for computing the output of the NN, taken directly from Toscano, 2022.

```
def forward(self,x):
    'this function computes the output of the neural network'
    if torch.is_tensor(x) != True:
        x = torch.from_numpy(x)
    a = x.float()
    for i in range(len(self.layers)-2):
        z = self.linears[i](a)
        a = self.activation(z)
    a = self.linears[-1](a)
    if self.HardConstrain is not None:
        a = self.HardConstrain(x,a)#? optional step to impose boundary conditions
    return a
```

B.2 Plot Loss History

```
def PlotHistory(self,pScheduler=False,SH=False):
    fig, ax = plt.subplots(1,figsize=(16,10))
    fig.subplots_adjust(hspace=0.01,wspace=0)
    if pScheduler:
        try:
            lrp = ax.plot(self.schhist,color='k',ls='-.-',alpha=0.7)
            legend1 = plt.legend(lrp,['Learning rate'],loc='lower left')
            ax.add_artist(legend1)
```

```
            legend1.get_frame().set_alpha(None)
            legend1.get_frame().set_facecolor((1, 0, 0, 0.1))
        except:pass

    total = ax.plot(self.loss_history['total_loss'],color='m',ls='-',label='Total')
    legends = total
    if len(self.loss_history['total_pde_loss'])>0:
        pde = ax.plot(self.loss_history['total_pde_loss'],color='b',ls='-',label='PDE')
        bd = ax.plot(self.loss_history['total_bd_loss'],color='r',ls='-',label='BD')
        legends+=pde;legends+=bd
    try:test = ax.plot(self.loss_history['test_iter'],self.loss_history['test_loss'],color='g',ls='--',label='Test');legends+=test
    except:pass
    labs = [curve.get_label() for curve in legends]
    legend2 = plt.legend(legends,labs,title='Loss Type:',markerfirst=False,edgecolor='k',alignment='right',loc='upper right',title_fontsize=12)

    legend2.get_frame().set_alpha(None)
    legend2.get_frame().set_facecolor((1, 1, 1, 1))
    ax.add_artist(legend2)
    ax.set_yscale('log');ax.set_ylabel('loss');ax.set_xlabel('Epochs')
    ymin,ymax=ax.get_ylim()
    try:nymin = min(self.loss_history[f'total_loss'])*(1-self.focusmin)
    except: nymin = ymin
    try:nymax = max(self.loss_history[f'total_loss'])*(1+self.focusmax)
    except: nymax = ymax
    if nymin>ymin:ymin=nymin
    if ymax>nymax:ymax=nymax
    ax.set_ylim(ymin,ymax)
    fig.savefig(f'Figures/{self.name}/Images/Total_loss_{self.n}.png',bbox_inches='tight')
    if SH:
        cmap1 = get_cmap(3*self.numequ+1);cmap2 = get_cmap(3*self.numcdt+1)
        if self.numcdt>1 and self.numequ>1:
```

```
            fig2, ax2 = plt.subplots(2,figsize=(16,15),sharex=True)
        else:
            fig2, ax2 = plt.subplots(1,figsize=(16,10))
            ax2 = [ax2]
        fig2.subplots_adjust(hspace=0.01,wspace=0)
        name=''
        a = 0
        if self.numequ>1:
            name = '_pde'
            a = 1
            leg = ax2[0].plot(self.loss_history[f'total{name}_loss'],color='k',ls='-',label='Total')
            for i in range(self.numequ):
                leg += ax2[0].plot(self.loss_history[f'pde_loss_{i}'],color=cmap1(3*i),ls='-',label=f'Eq {i+1}')
            labs = [curve.get_label() for curve in leg]
            legend2 = ax2[0].legend(leg,labs,title='PDE Loss:',markerfirst=False,edgecolor='k',alignment='right',loc='upper right',title_fontsize=12)
            legend2.get_frame().set_alpha(None)
            legend2.get_frame().set_facecolor((1, 1, 1, 1))
            ax2[0].set_yscale('log');ax2[0].set_ylabel('loss')
            ymin,ymax=ax2[0].get_ylim()
            try:nymin = min(self.loss_history[f'total{name}_loss'])*(1-self.focusmin)
            except: nymin = ymin
            try:nymax = max(self.loss_history[f'total{name}_loss'])*(1+self.focusmax)
            except: nymax = ymax
            if nymin>ymin:ymin=nymin
            if ymax>nymax:ymax=nymax
            ax2[0].set_ylim(ymin,ymax)
        if self.numcdt>1:
            leg = ax2[a].plot(self.loss_history['total_bd_loss'],color='k',ls='-',label='Total')
            for i in range(self.numcdt):
```

```
                    if len(self.loss_history[f'bd_loss_{i}'])>0:leg += ax2[a].plot(self.loss_history[f'bd_loss_{i}'],color=cmap2(3*i),ls='-',label=f'Func␣{i+1}')
                labs = [curve.get_label() for curve in leg]
                legend2 = ax2[a].legend(leg,labs,title='BD␣Loss:',markerfirst=False,edgecolor='k',alignment='right',loc='upper␣right',title_fontsize=12)
                legend2.get_frame().set_alpha(None)
                legend2.get_frame().set_facecolor((1, 1, 1, 1))
                ax2[a].set_yscale('log');ax2[a].set_ylabel('loss')

                ymin,ymax=ax2[a].get_ylim()
                try:nymin = min(self.loss_history[f'total_bd_loss'])*(1-self.focusmin)
                except: nymin = ymin
                try:nymax = max(self.loss_history[f'total_bd_loss'])*(1+self.focusmax)
                except: nymax = ymax
                if nymin>ymin:ymin=nymin
                if ymax>nymax:ymax=nymax
                ax2[a].set_ylim(ymin,ymax)
            ax2[-1].set_xlabel('Epochs')
            fig2.savefig(f'Figures/{self.name}/Images/Detailed_loss_{self.n}.png',bbox_inches='tight')
            return fig, ax, fig2, ax2
        return fig, ax
```

```
def get_cmap(n, name='hsv'):
    '''Returns a function that maps each index in 0, 1, ..., n-1 to a distinct
    RGB color; the keyword argument name must be a standard mpl colormap name.'''
    return plt.cm.get_cmap(name, n)
```

[1]

[1]Code taken from Ali, 2014

B.3 Save Last Prediction

```
def save(self):
    if not exists('Models'):
        makedirs('Models')
    print(self.name)
    torch.save(self,f'Models/{self.name}/Final/Trained_{
       self.n}.model')
```

Appendix C: Error function

A non trivial decision to be made is the selection of the function to compute the found discrepancy between the analytical results and the predicted ones. This can be measured using the trivial function:

$$err = \frac{y_{real} - y_{pred}}{y_{real}},$$

but this function has the disadvantage of not been properly defined for $y_{real} = 0$, value that in the shown examples is recurrent, those motivates the necessity of a new definition of the discrepancy. There are multiple ways to tackle this problem as presented in Leo, Pentti, and Yrjo O., 1985 & Vartia, 1976 although the mayority of the options here presented still conserves the non-determination for null values of the real solution or even are not defined for negative values. Due to this reasons the selected discrepancy funciton was:

$$err = 2\frac{y_{real} - y_{pred}}{|y_{real}| + 1},$$

which behaves as shown in Fig. C.1.

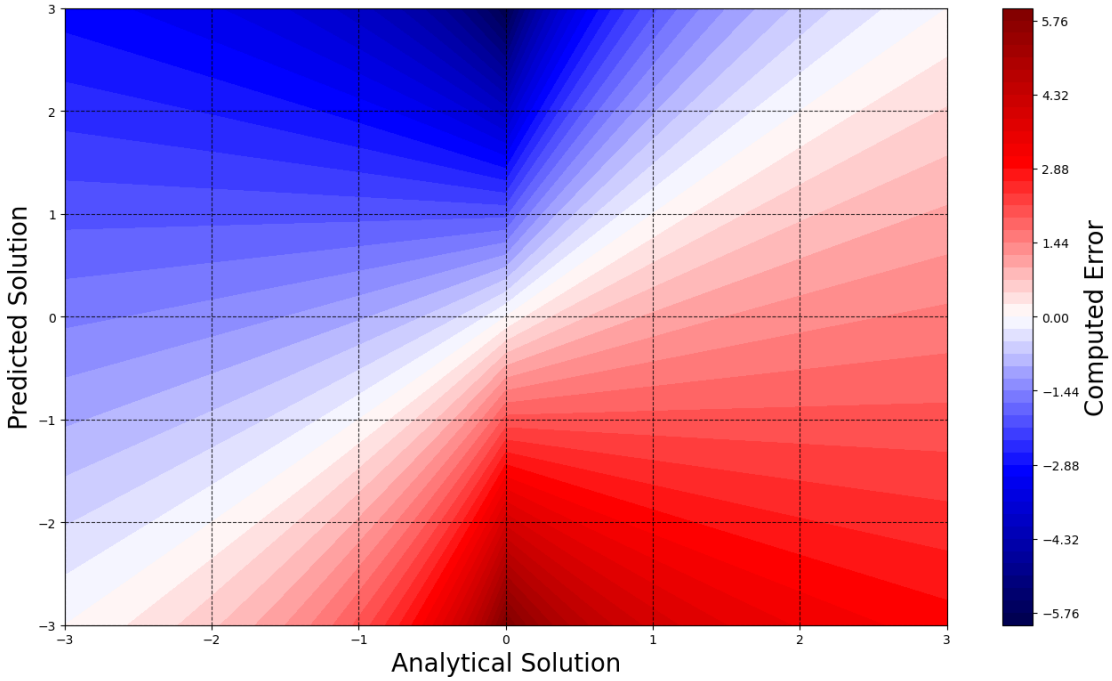

Figure C.1: Error function behavior

Appendix D: Kovasznay flow discrepancy plots.

For the sake of space saving ,the first shown discrepancy plots, consisting of Figs. D.1, D.2, D.3, D.4 & D.5, are the ones of the time-dependent DP, the time independent discrepancy plots are shown in Fig. D.6

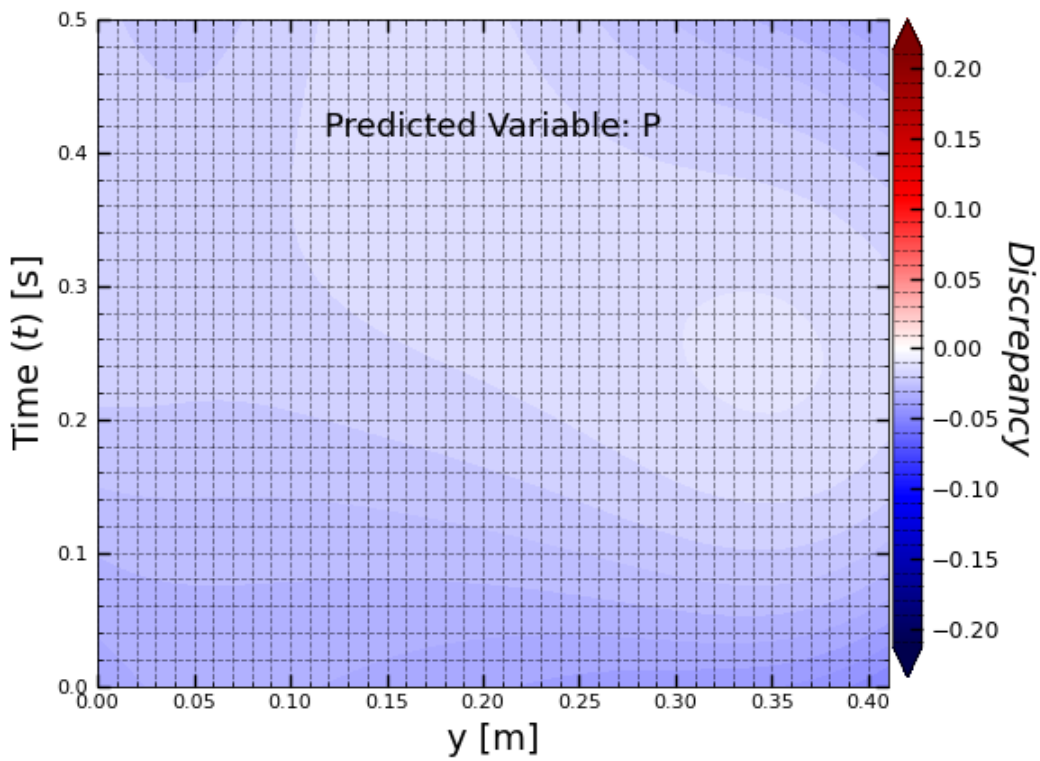

Figure D.1: Outlet discrepancy plot for the time dependent DP.

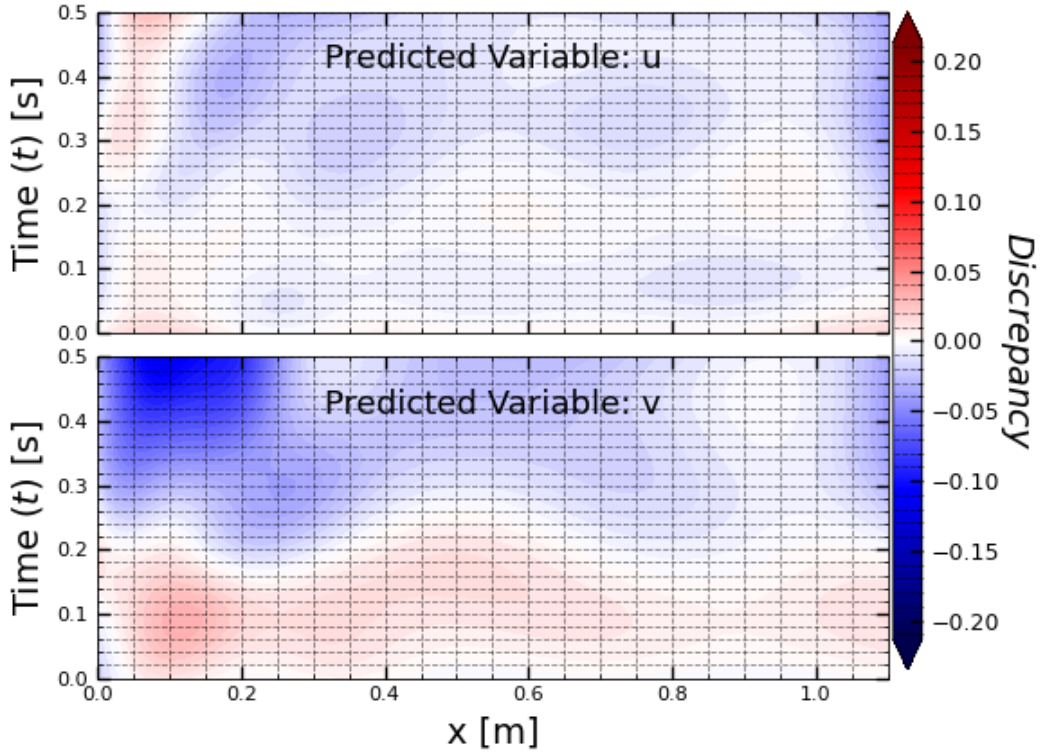

Figure D.2: Non-slip condition at upper wall discrepancy plot for the time dependent DP.

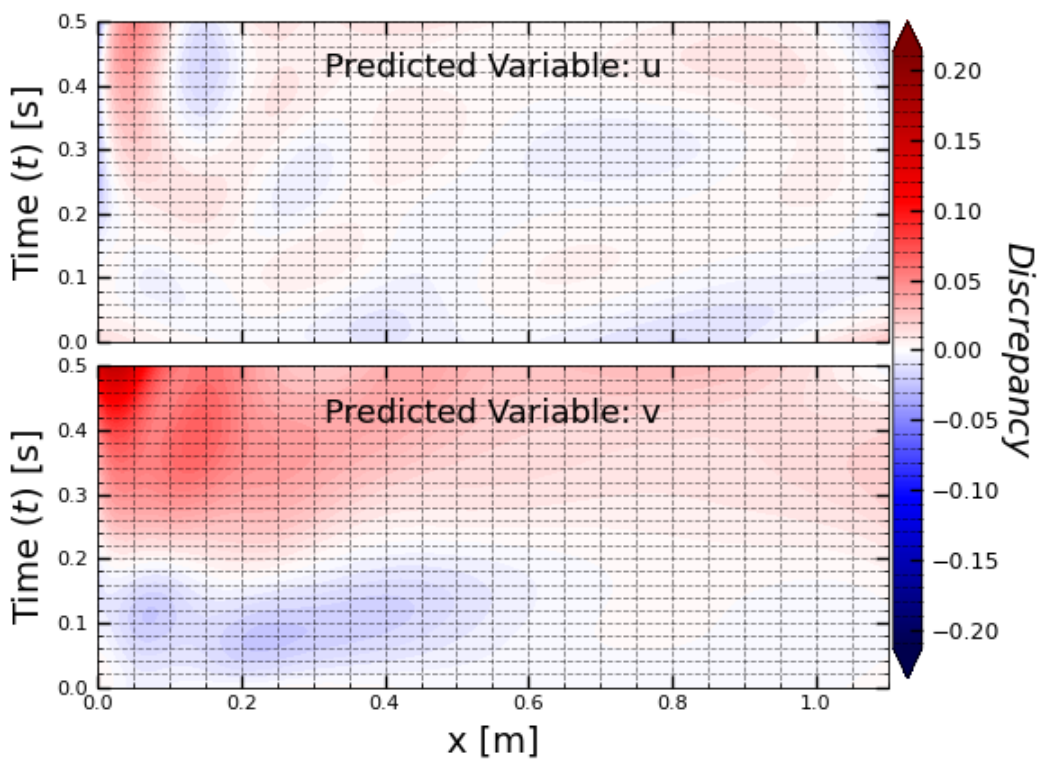

Figure D.3: Non-slip condition at lower wall discrepancy plot for the time dependent DP.

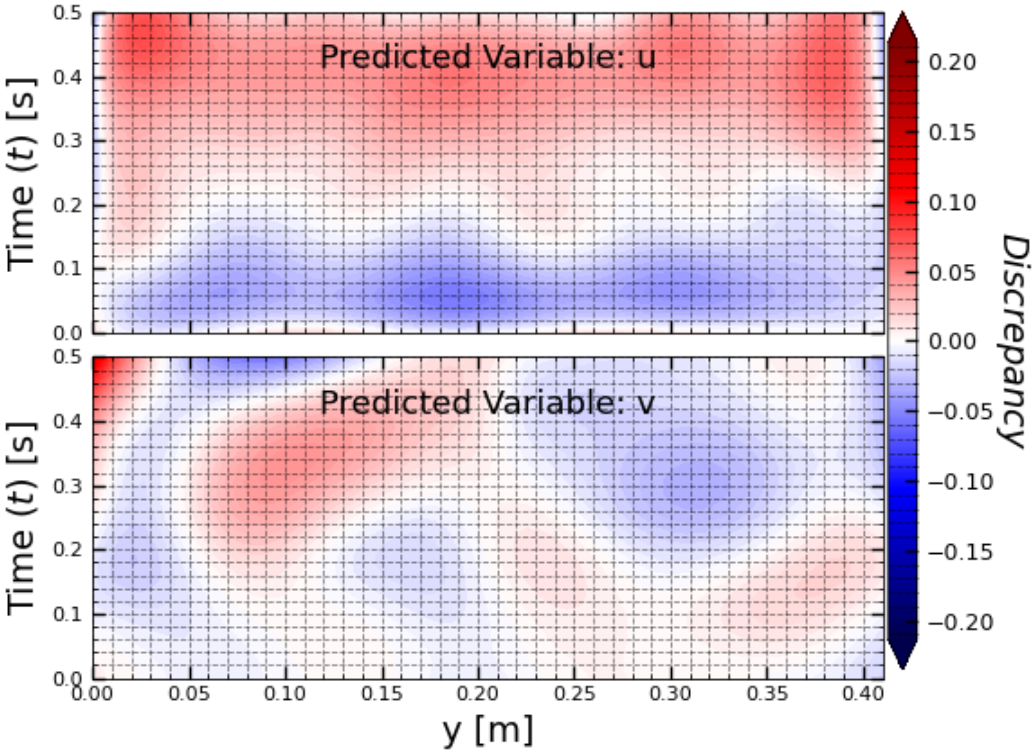

Figure D.4: Inlet discrepancy plot for the time dependent DP.

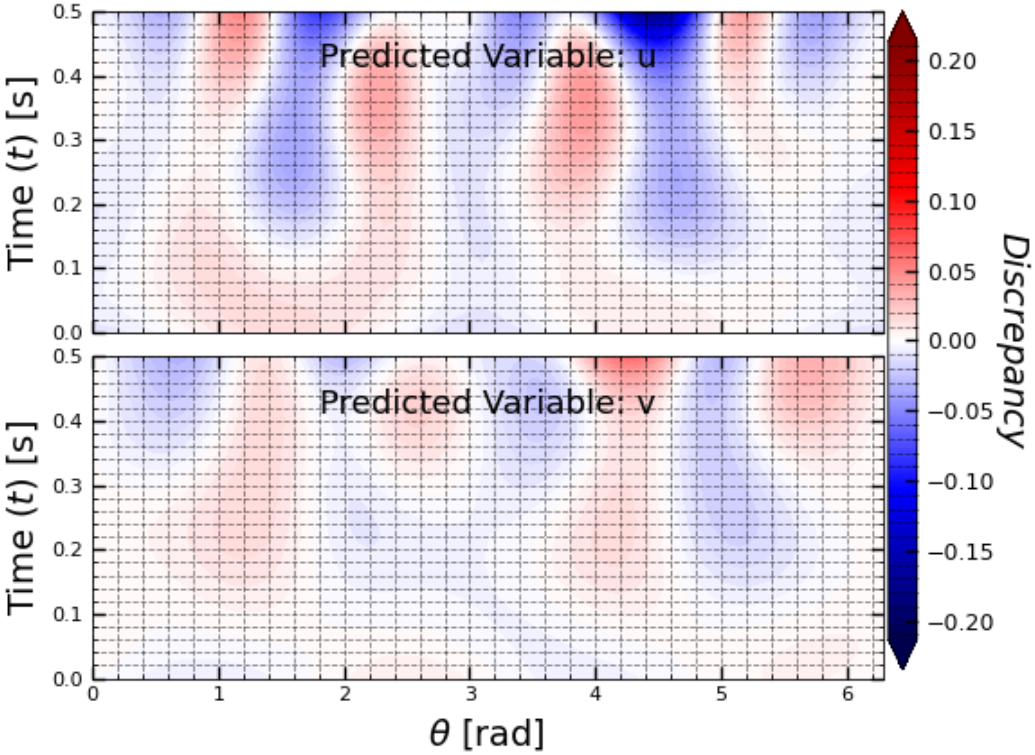

Figure D.5: Non-slip condition at cylinder wall for the time dependent DP

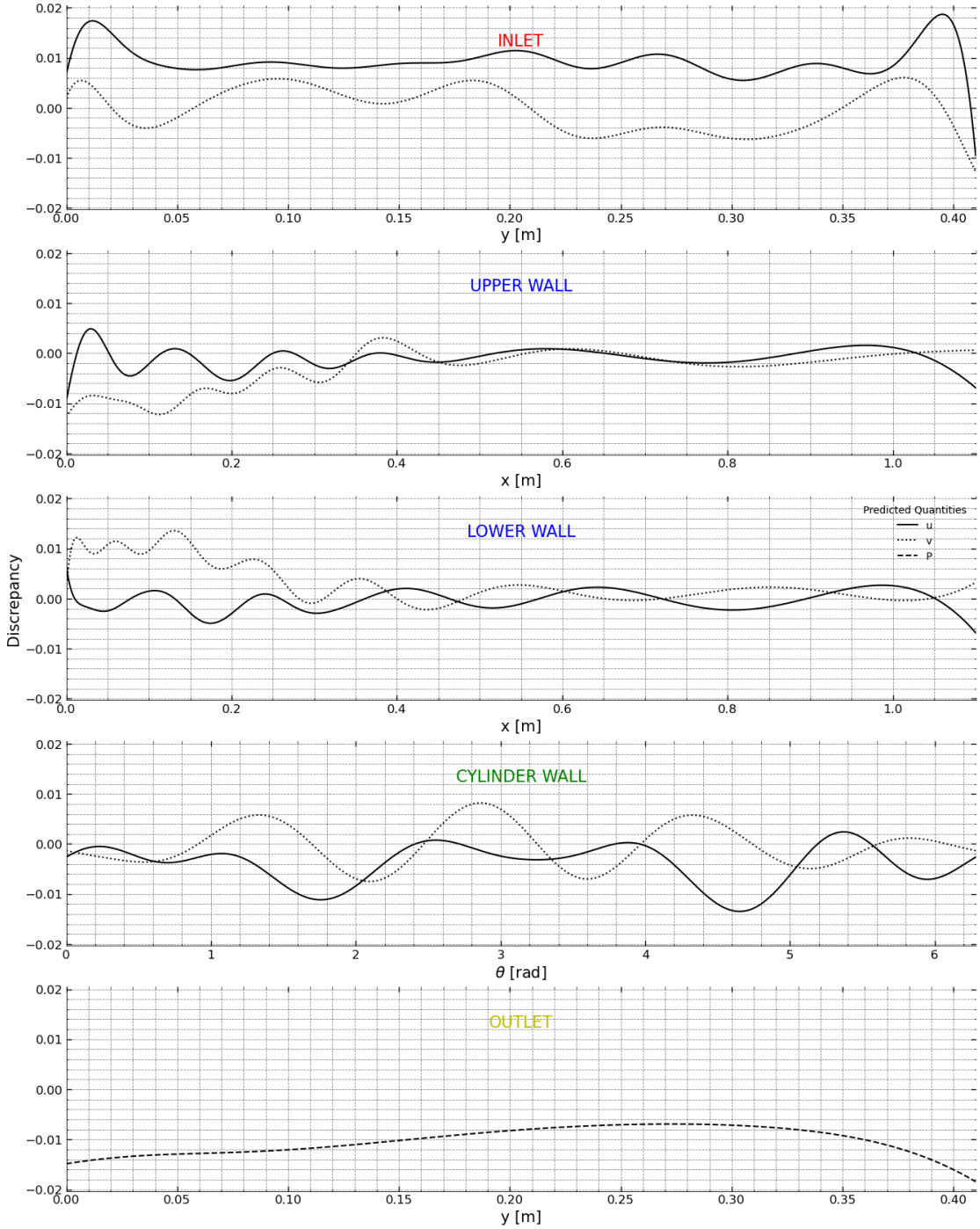

Figure D.6: Discrepancy lines for the time independent DP

Appendix E: Not So Pretty PDE-PINN Examples: Exploring Challenging Results.

Up to this point all the predictive potential of PINNs has been shown through examples that include a wide range of characteristics and applications through many fields but, as with any other scientific research, not all the attempted DP yield the desired outcomes when tried to be solved using PINNs. In this brief appendix a series of unconverged DP are shown with the expectation of further motivating a continuation of the present work, demonstrating that, as explained in the Future Work chapter, is still far from being completed.

E.1 Vibration of 2-Dimensional Membrane

The first failed DP to be shown is regarding the two dimensional oscillatory motion. A, prior thought, easy generalization of the 1D example shown in Sec. 4.2. Two cases were tried out, the rectangular(square) and circular membrane, and in both cases without obtaining good results.

E.1.1 Rectangular membrane

As widely known, the PDE that describes this DP is:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right),$$

where $(x, y) \in [0, L] \times [0, H]$ are the spatial position of the membrane, $t \in [0, 1]$ is the time and u is the displacement of the membrane with respect to the resting state (flat). It was selected $L = H = 1\,m$, $c = 1\,m/s$ and for boundary condition:

$$u(x, y, t) = 0 \quad \forall (x, y) \in \{[0, L] \times [0, H]\}_\partial$$

The general solution to the aforementioned DP is:

$$u(x, y, t) = \sum_{n,m} (a_{nm} \cos(\omega_{nm} t) + b_{nm} \sin(\omega_{nm} t)) \sin(n\pi x) \sin(m\pi y),$$

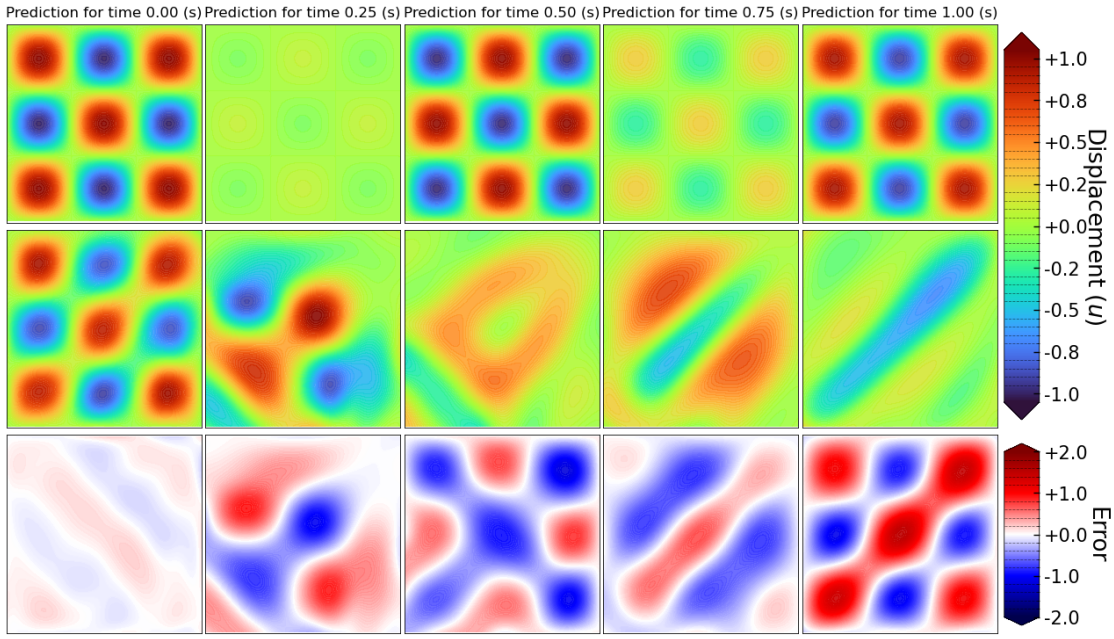

Figure E.1: Solution to the rectangular vibration DP, the first row of panels represent the analytical solution, the second the prediction made by the PINN and the third the estimated discrepancy.

with $\omega_{nm} = \frac{\pi}{2}\sqrt{n^2 + m^2}$[1]. Selecting as initial condition:

$$u(x, y, 0) = sin(3\pi x)sin(3\pi y); \quad \left.\frac{\partial u}{\partial t}\right|_{t=0} = 0$$

It forces the solution, due to the linear independence of the sinusoidal function and as well the nullity of the first derivative of the displacement respect to time at $t = 0$ to be:

$$u(x, y, t) = cos(\omega_{33}t)sin(3\pi x)sin(3\pi y).$$

The selected architecture for the PINN is $[3, 75 \otimes 7, 1]$, with $LBFGS$ optimizer using the optional arguments shown in Sec. 5.2 as well as learning rate scheduler $StepLR$ and arguments $\gamma = 1.5$ and step size 100. Only two CDT were imposed, both of type Dirichlet, the first regarding the boundary condition and the second the initial condition[2]. The results of the training is shown in Fig. E.1, where it can be seen that only the first simulation time step, which involves the initial condition, is close to the analytical solution.

[1] the complete formulation is $\omega_{nm} = \frac{c\pi}{2}\sqrt{\left(\frac{n}{L}\right)^2 + \left(\frac{m}{H}\right)^2}$

[2] not including the first time derivative condition

E.1.2 Circular membrane

Considering now a circular membrane with radius $R = 1$, expressing the oscillatory equations in polar coordinates the following equation is obtained:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} + \frac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2} \right).$$

It was assumed that DP was symmetric and thus independent of θ, meaning that the equation could be reduced and re-written as:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} \right)$$

One can then assume that the solution could be written as $u(r,t) = \mathcal{R}(r)T(t)$, leading to:

$$\begin{aligned} \frac{1}{c^2 T(t)}\frac{\mathrm{d}^2 T}{\mathrm{d}t^2} =& \frac{1}{\mathcal{R}(r)}\left(\frac{\mathrm{d}^2 \mathcal{R}}{\mathrm{d}r^2} + \frac{1}{r}\frac{\mathrm{d}\mathcal{R}}{\mathrm{d}r} \right) \\ \frac{\mathrm{d}^2 T}{\mathrm{d}t^2} =& Kc^2 T(t) \\ r\frac{\mathrm{d}^2 \mathcal{R}}{\mathrm{d}r^2} + \frac{\mathrm{d}\mathcal{R}}{\mathrm{d}r} =& Kr\mathcal{R}(r) \end{aligned}$$

rewritting $K = -\lambda^2$ the time dependent equation is transformed to a simple harmonic oscillator problem, with simple solution $T(t) = acos(c\lambda t) + bsin(c\lambda t)$. The radial equation is the way more complex Bessel differential equation for which the solution is a linear combination of Bessel functions of order 0. Taking into account the the function Y_0 is not defined for $r = 0$ its not plausible to include it as solution to the DP. Taking into account that for $r = R$ $u(R,t) = 0$ the general solution for the equation is:

$$u(r,t) = \sum_n (a_n cos(c\lambda_n t) + b_n sin(c\lambda_n t)) J_0(\lambda_n r)$$

where $\lambda_n = \alpha_n / R$ is the n-th root of the Bessel function, meaning $J_0(\lambda_n R) = 0$. Taking the initial condition:

$$u(r,0) = J_0(\lambda_3 r)$$

Due to linear independence of the Bessel functions the solution to the DP is set to:

$$u(r,t) = cos(c\lambda_3 t) J_0(\lambda_3 r).$$

Assuming $c = 1$ ans using a PINN architecture $[2, 75 \otimes 7, 1]$ with the same arguments of the previous example and as CDT only the initial condition as well

as the boundary condition. The prediction of the model is then presented in Fig. E.2 and, as it can there be seen the PINN DOES NOT PREDICT! multiple trainning strategies were adopted, including separetly predicting $\mathcal{R}$ and T, but neither of then gave a different solution.

E.2 Two-Dimensional Helmholtz Equation

The second failed DP is the two dimensional Helmholtz equation problem, this DP is important in physics because its represents the time-independent wave equation. The following example is taken from Lu et al., 2021a which in turn was extracted from Ihlenburg, 1998. The PDE studied is:

$$\frac{\partial u}{\partial y} + \frac{\partial u}{\partial x} + k^2 u - (-(a_1\pi)^2 - (a_2\pi)^2 + k^2) sin(a_1\pi x) * sin(a_2\pi y),$$

where $x, y \in [-1, 1]$ and $k = a_1 = a_2 = 1$. This means that the domain Ω is the square $[-1, 1]^2$, for points in the boundary the solution of the equation most satisfy:

$$u(x, y) = 0 \qquad\qquad (x, y) \in \partial\Omega.$$

This leads to the solution of the DP being $u(x, y) = sin(a_1\pi x) * sin(a_2\pi y)$. Multiple architecture and mixture of optimizers, learning rate schedulers, data-feeding the model, even different DP, were trained, all of them leading to poor results, the shown in Fig. E.3 corresponds to the last of the training iterations, with an architecture $[2, 80 \otimes 2, 50, 80 \otimes 2, 1]$, cyclic activation function sin, $LFBGS$ optimizer with the previously mentioned arguments. Two CDTs were considered, one concerning the boundary condition and other for data feeding the PINN[3]. As it can be seen in the figure, not even the boundary condition is satisfied[4], the poor estimation, principally in the central region where the solution turns null, motivate to further sample this region as collocation points[5], but even with this decision the PINN could not properly predict the solution to the DP.

E.3 Time dependent Cavity Flow

As previously mentioned in Sec. 5.3.2, the time dependent cavity flow DP does not present a well converged PINN. As with the other DP mentioned in this chapter, multiple architecture configurations were tried out, the results for a

[3]another studied architecture is the one shown in Lu et al., 2021a, $[2, 150 \otimes 3, 1]$ with initial learning rate $\lambda = 10^{-2}$ accompanied with a scheduler "StepLR" with attenuation $\gamma = 0.8$ each 1000 training epochs, also failing to properly predict the solution

[4]although on some of the trained PINNs this was accomplished

[5]maybe another case yet to be tried is to data-feed this section of the PINN separately.

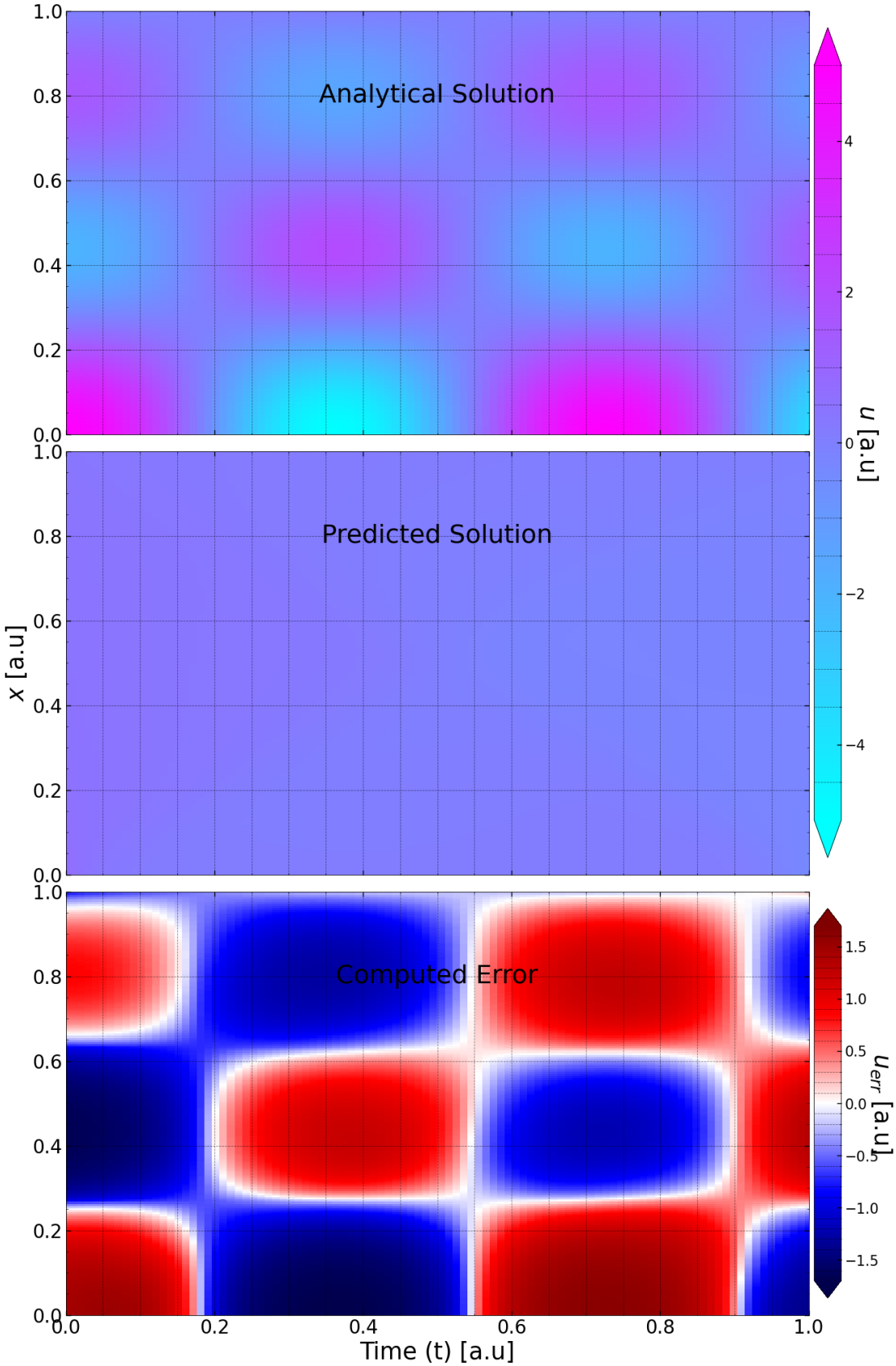

Figure E.2: PINN solution for circular membrane DP, the error is computed as $U_{PINN} - u_{analytical}$

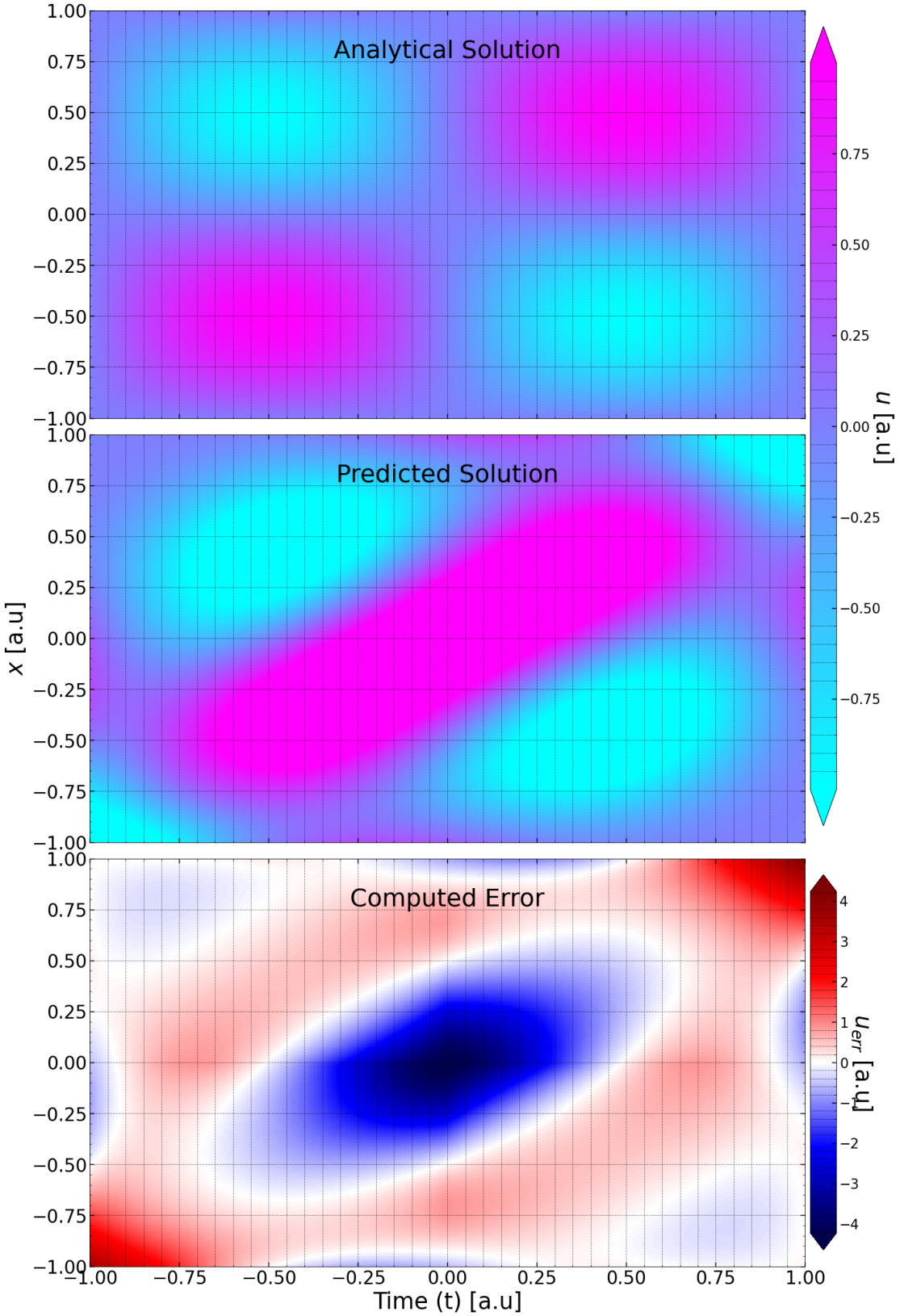

Figure E.3: Demonstration of the predicted solution to the Helmholtz equation DP using PINNs

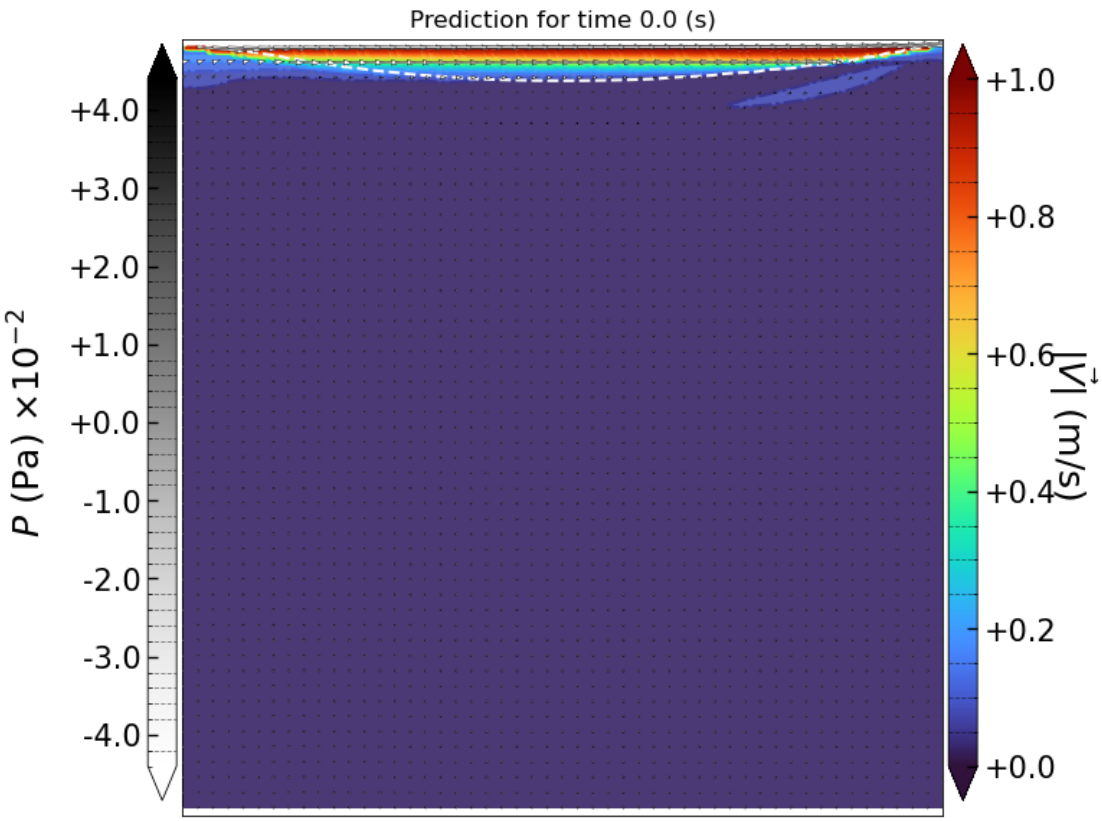

Figure E.4: Time step example of the training results for the full domain $(x, y) \in [0, 1]^2$.

time step example of the whole domain of the DP are shown in Fig. E.4, in Fig. E.5 a zoomed version capturing the domain $(x, y) \in [0, 1] \times [0.9, 1]$. The used architecture is $[3, 45 \otimes 8, 5]$, with LBFGS optimizer, using the parameters previously shown, as well as learning rate optimizer StepLR with $\gamma = 1.5$ and step size 100. As it can be observed in the in the figure, throughout the different panels (which represent different time steps for the simulation), the predicted model those not present the expected evolution, as the one seen in Fig. 5.6.

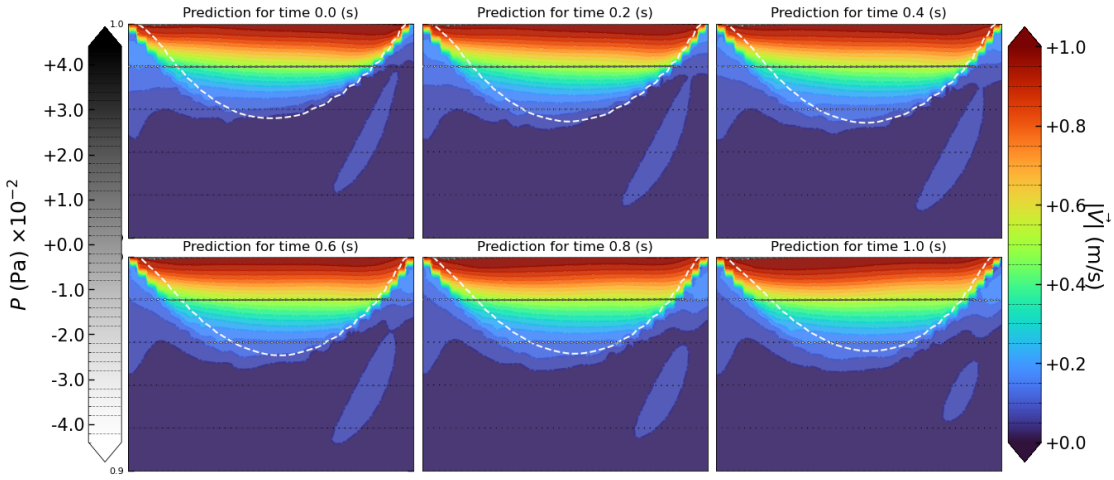

Figure E.5: Zoomed-Cavity flow time dependent DP using 2D Kovasznay flow formulation of the Navier-Stokes equations.

Bibliography

Abbasi, Jassem and Pål Østebø Andersen (2022). *Physical Activation Functions (PAFs): An Approach for More Efficient Induction of Physics into Physics-Informed Neural Networks (PINNs)*. arXiv: 2205.14630 [cs.LG].

Ali (2014). *How to generate random colors in matplotlib?* URL: https://stackoverflow.com/questions/14720331/how-to-generate-random-colors-in-matplotlib.

Amari, Shun-ichi (1993). "Backpropagation and stochastic gradient descent method". In: *Neurocomputing* 5.4-5, pp. 185–196.

Bai, Yuexing, Temuer Chaolu, and Sudao Bilige (2022). "The application of improved physics-informed neural network (IPINN) method in finance". In: *Nonlinear Dynamics* 107.4, pp. 3655–3667.

Basov, Suren (2007). *Partial Differential Equations in Economics and Finance*. Nova Publishers.

Becker, Adib A (1992). *The boundary element method in engineering: a complete course*. McGraw-Hill Companies.

Bhustali, Prateek (2021). *Physics-Informed-Neural-Networks*. https://github.com/omniscientoctopus/Physics-Informed-Neural-Networks/blob/main/PyTorch/Burgers'%20Equation/Burgers.ipynb.

Bienz, C et al. (2003). "In Front of the Grid–CFD at SAUBER PETRONAS F1 Leading the Aerodynamic Development". In: *1st European Automotive CFD Conference, Bingen, Germany*.

Bottou, Léon (2012). "Stochastic gradient descent tricks". In: *Neural Networks: Tricks of the Trade: Second Edition*, pp. 421–436.

Brauer, Fred, Carlos Castillo-Chavez, and Carlos Castillo-Chavez (2012). *Mathematical models in population biology and epidemiology*. Vol. 2. 40. Springer.

Bullwinkel, Blake et al. (2022). *DEQGAN: Learning the Loss Function for PINNs with Generative Adversarial Networks*. arXiv: 2209.07081 [cs.LG].

Caldwell, J, P Wanless, and AE Cook (1981). "A finite element approach to Burgers' equation". In: *Applied Mathematical Modelling* 5.3, pp. 189–193.

Chandrupatla, T.R. and A.D. Belegundu (2014). *Introduction to Finite Elements in Engineering: International Edition*. Pearson Education. ISBN: 9781292014029. URL: https://books.google.com.co/books?id=yQWpBwAAQBAJ.

Chen, Mark et al. (2020). "Generative pretraining from pixels". In: *International conference on machine learning*. PMLR, pp. 1691–1703.

Chen, Ricky T. Q. (2018). *torchdiffeq*. URL: https://github.com/rtqichen/torchdiffeq.

Chiu, Pao-Hsiung (2018). "An improved divergence-free-condition compensated method for solving incompressible flows on collocated grids". In: *Computers Fluids* 162, pp. 39–54. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2017.12.005. URL: https://www.sciencedirect.com/science/article/pii/S0045793017304413.

Cobbe, Karl et al. (Oct. 2021). "Training Verifiers to Solve Math Word Problems". In: *arXiv e-prints*, arXiv:2110.14168, arXiv:2110.14168. DOI: 10.48550/arXiv.2110.14168. arXiv: 2110.14168 [cs.LG].

Cullen, M. J. P., J. Norbury, and R. J. Purser (1991). "Generalised Lagrangian Solutions for Atmospheric and Oceanic Flows". In: *SIAM Journal on Applied Mathematics* 51.1, pp. 20–31. DOI: 10.1137/0151002. eprint: https://doi.org/10.1137/0151002. URL: https://doi.org/10.1137/0151002.

Cunningham, Pádraig, Matthieu Cord, and Sarah Jane Delany (2008). "Supervised learning". In: *Machine learning techniques for multimedia: case studies on organization and retrieval*, pp. 21–49.

Desai, Shaan (2021). "Physics-informed neural networks for data-efficient learning". PhD thesis. University of Oxford.

Edgar, Thomas F, David M Himmelblau, Leon S Lasdon, et al. (2001). *Optimization of chemical processes*.

Faroughi, Salah A et al. (2022). *Physics-informed Neural Networks with Periodic Activation Functions for Solute Transport in Heterogeneous Porous Media*. arXiv: 2212.08965 [cs.LG].

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, pp. 249–256.

Gonze, Xavier et al. (2020). "The Abinitproject: Impact, environment and recent developments". In: *Computer Physics Communications* 248, p. 107042. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2019.107042. URL: https://www.sciencedirect.com/science/article/pii/S0010465519303741.

Goswami, Somdatta et al. (2022). *Neural operator learning of heterogeneous mechanobiological insults contributing to aortic aneurysms*. arXiv: 2205.03780 [cs.LG].

Gu, Xiangyu, Pier Davide Ciampa, and Björn Nagel (2018). "An automated CFD analysis workflow in overall aircraft design applications". In: *CEAS Aeronautical Journal* 9, pp. 3–13.

Harold, J, G Kushner, and George Yin (1997). "Stochastic approximation and recursive algorithm and applications". In: *Application of Mathematics* 35.

Hesthaven, Jan S, Sigal Gottlieb, and David Gottlieb (2007). *Spectral methods for time-dependent problems*. Vol. 21. Cambridge University Press.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5, pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1990). "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks". In: *Neural Networks* 3.5, pp. 551–560. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(90)90005-6. URL: https://www.sciencedirect.com/science/article/pii/0893608090900056.

Hughes, T.J.R. (2012). *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover Civil and Mechanical Engineering. Dover Publications. ISBN: 9780486135021. URL: https://books.google.com.co/books?id=cHH2n_qBK0IC.

Ihlenburg, Frank (1998). "Finite Element Error Analysis and Control for Helmholtz Problems". In: *Finite element analysis of acoustic scattering*. Springer, 138–139.

Ingle, Shantanu and Madhuri Phute (2016). "Tesla autopilot: semi autonomous driving, an uptick for future autonomy". In: *International Research Journal of Engineering and Technology* 3.9, pp. 369–372.

Jadon, Shruti (2018). "Introduction to different activation functions for deep learning". In: *Medium, Augmenting Humanity* 16.

Jagtap, Ameya D., Kenji Kawaguchi, and George Em Karniadakis (2020). "Adaptive activation functions accelerate convergence in deep and physics-informed neural networks". In: *Journal of Computational Physics* 404, p. 109136. DOI: 10.1016/j.jcp.2019.109136. URL: https://doi.org/10.1016%2Fj.jcp.2019.109136.

Karpatne, Anuj et al. (2017). "Theory-Guided Data Science: A New Paradigm for Scientific Discovery from Data". In: *IEEE Transactions on Knowledge and Data Engineering* 29.10, pp. 2318–2331. DOI: 10.1109/tkde.2017.2720168. URL: https://doi.org/10.1109%2Ftkde.2017.2720168.

Kidger, Patrick (2022). "On neural differential equations". In: *arXiv preprint arXiv:2202.02435*.

Kingma, Diederik P. and Jimmy Ba (Dec. 2014). "Adam: A Method for Stochastic Optimization". In: *arXiv e-prints*, arXiv:1412.6980, arXiv:1412.6980. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980 [cs.LG].

Kovasznay, L. I. G. (1948). "Laminar flow behind a two-dimensional grid". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 44.1, 58–62. DOI: 10.1017/S0305004100023999.

Larsson, T, T Sato, and B Ullbrand (2005). "Supercomputing in F1–Unlocking the Power of CFD". In: *2nd European Automotive CFD Conference*, pp. 29–30.

Lemay, Samuel (2023). "Molecular catalysts for the hydrogen evolution reaction: a first-principles study". In: *Bulletin of the American Physical Society*.

Leo, Tornqvist, Vartia Pentti, and Vartia Yrjo O. (1985). "How Should Relative Changes Be Measured?" In: *The American Statistician* 39.1, pp. 43–46. ISSN: 00031305. URL: http://www.jstor.org/stable/2683905 (visited on 06/21/2023).

Loh, Wei-Liem (1996). "On Latin hypercube sampling". In: *The annals of statistics* 24.5, pp. 2058–2080.

Lu, Lu et al. (2021a). "DeepXDE: A deep learning library for solving differential equations". In: *SIAM Review* 63.1, pp. 208–228. DOI: 10.1137/19M1274067.

Lu, Lu et al. (2021b). "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators". In: *Nature Machine Intelligence* 3.3, pp. 218–229. DOI: 10.1038/s42256-021-00302-5. URL: https://doi.org/10.1038%2Fs42256-021-00302-5.

Markidis, Stefano (2022). "On physics-informed neural networks for quantum computers". In: *Frontiers in Applied Mathematics and Statistics* 8. DOI: 10.3389/fams.2022.1036711. URL: https://doi.org/10.3389%2Ffams.2022.1036711.

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5, pp. 115–133.

Moya, Luz Marina and Edixon Rojas (2021). *Ecuaciones diferenciales ordinarias: técnicas de resolución*. Universidad Nacional de Colombia.

Nadeau, Olivier and Gabriel Antonius (2023). "Numerical modeling of hydrogen absorption in metal hydrides". In: *Bulletin of the American Physical Society*.

Nagle, R. Kent et al. (2005). *Ecuaciones diferenciales y problemas con valores en la Frontera*. Pearson Educacion de Mexico.

Okada39 (2020). *pinn_cavity*. https://github.com/okada39/pinn_cavity.

OpenAI (Mar. 2023). "GPT-4 Technical Report". In: *arXiv e-prints*, arXiv:2303.08774, arXiv:2303.08774. DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774 [cs.CL].

Polyanin, Andrei D. (2016). *Handbook of Linear Partial Differential Equations for engineers and scientists*. Chapman and Hall.

Polyanin, Andrei D. and V. F. Zaitsev (1995). *Handbook of exact solutions for ordinary differential equations*. CRC Press.

Raissi, M., P. Perdikaris, and G.E. Karniadakis (2019). "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378, pp. 686–707. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045. URL: https://www.sciencedirect.com/science/article/pii/S0021999118307125.

Raissi, Maziar, Paris Perdikaris, and George Em Karniadakis (2017a). *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*. arXiv: 1711.10561 [cs.AI].

— (2017b). *Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations*. arXiv: 1711.10566 [cs.AI].

Rao, Chengping, Hao Sun, and Yang Liu (2020). "Physics-informed deep learning for incompressible laminar flows". In: *Theoretical and Applied Mechanics Letters* 10.3, pp. 207–212. DOI: 10.1016/j.taml.2020.01.039. URL: https://doi.org/10.1016%2Fj.taml.2020.01.039.

Ren, Pu et al. (2022a). *PhyCRNet*. https://github.com/isds-neu/PhyCRNet/tree/main.

— (2022b). "PhyCRNet: Physics-informed convolutional-recurrent network for solving spatiotemporal PDEs". In: *Computer Methods in Applied Mechanics and*

Engineering 389, p. 114399. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2021.114399. URL: https://www.sciencedirect.com/science/article/pii/S0045782521006514.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, p. 386.

— (1961). *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. Cornell Aeronautical Lab Inc Buffalo NY.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). "Learning representations by back-propagating errors". In: *nature* 323.6088, pp. 533–536.

Saint-Cirgue, Guillaume (2021). *LE PERCEPTRON - DEEP LEARNING (O2)*. Machine Learnia, Youtube. URL: https://www.youtube.com/watch?v=VlMm4VZ6lk4.

Saputro, Dewi Retno Sari and Purnami Widyaningsih (2017). "Limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method for the parameter estimation on geographically weighted ordinal logistic regression model (GWOLR)". In: *AIP Conference Proceedings*. Vol. 1868. 1. AIP Publishing LLC, p. 040009.

Sazli, Murat H (2006). "A brief review of feed-forward neural networks". In: *Communications Faculty of Sciences University of Ankara Series A2-A3 Physical Sciences and Engineering* 50.01.

Schäfer, Violetta (2022). "Generalization of Physics-Informed Neural Networks for Various Boundary and Initial Conditions". PhD thesis. Technische Universität Kaiserslautern.

Sepúlveda Soto, Alfonso (2009). *Física matemática*. Ciencia y Tecnología. Universidad de Antioquia. ISBN: 9789587143072. URL: https://books.google.com.co/books?id=GVllwgJN0UwC.

Shah, Karan et al. (2022). *Physics-Informed Neural Networks as Solvers for the Time-Dependent Schrödinger Equation*. arXiv: 2210.12522 [quant-ph].

Shao, Yunqi et al. (2020). "PiNN: A Python Library for Building Atomic Neural Networks of Molecules and Materials". In: *J. Chem. Inf. Model.* 60.3, pp. 1184–1193. DOI: 10.1021/acs.jcim.9b00994.

Shen, J., T. Tang, and L.L. Wang (2011). *Spectral Methods: Algorithms, Analysis and Applications*. Springer Series in Computational Mathematics. Springer Berlin Heidelberg. ISBN: 9783540710417. URL: https://books.google.com.co/books?id=QmEBMK3tqkYC.

Sitzmann, Vincent et al. (June 2020). "Implicit Neural Representations with Periodic Activation Functions". In: *arXiv e-prints*, arXiv:2006.09661, arXiv:2006.09661. DOI: 10.48550/arXiv.2006.09661. arXiv: 2006.09661 [cs.CV].

Smith, Gordon D (1985). *Numerical solution of partial differential equations: finite difference methods*. Oxford university press.

Snell, R.S. (2010). *Clinical Neuroanatomy*. Clinical Neuroanatomy. Wolters Kluwer Health/Lippincott Williams & Wilkins. ISBN: 9780781794275. URL: https://books.google.com.co/books?id=ABPmvroyrD0C.

Stein, Michael (1987). "Large sample properties of simulations using Latin hypercube sampling". In: *Technometrics* 29.2, pp. 143–151.

Teshima, Takeshi et al. (2020). *Universal Approximation Property of Neural Ordinary Differential Equations*. arXiv: 2012.02414 [cs.LG].

Thuerey, Nils et al. (2021). "Physics-based deep learning". In: *arXiv preprint arXiv:2109.05237*.

Toscano, Juan Diego (2022). *Learning-Python-Physics-Informed-Machine-Learning-PINNs-DeepONets*. https://github.com/jdtoscano94/Learning-Python-Physics-Informed-Machine-Learning-PINNs-DeepONets/tree/main.

Vadyala, Shashank Reddy and Sai Nethra Betgeri (2023). "General implementation of quantum physics-informed neural networks". In: *Array* 18, p. 100287. ISSN: 2590-0056. DOI: https://doi.org/10.1016/j.array.2023.100287. URL: https://www.sciencedirect.com/science/article/pii/S2590005623000127.

Vartia, Yrjö O. (1976). *Relative Changes and Index Numbers*. ETLA A 4. The Research Institute of the Finnish Economy. URL: https://ideas.repec.org/b/rif/abooks/4.html.

Waldvogel, Jörg (1983). "The Period in the Volterra-Lotka Predator-Prey Model". In: *SIAM Journal on Numerical Analysis* 20.6, pp. 1264–1272. ISSN: 00361429. URL: http://www.jstor.org/stable/2157163 (visited on 05/28/2023).

Wang, Li and Zhenya Yan (2021). "Data-driven rogue waves and parameter discovery in the defocusing nonlinear Schrödinger equation with a potential using the PINN deep learning". In: *Physics Letters A* 404, p. 127408.

Wang, Ruifei et al. (2021). "A review on slip boundary conditions at the nanoscale: recent development and applications". In: *Beilstein Journal of Nanotechnology* 12, pp. 1237–1251. ISSN: 2190-4286. DOI: 10.3762/bjnano.12.91. URL: https://doi.org/10.3762/bjnano.12.91.

Wang, ZJ (2014). "High-order computational fluid dynamics tools for aircraft design". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 372.2022, p. 20130318.

Wikipedia contributors (2023). *Lotka–Volterra equations — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-June-2023]. URL: https://en.wikipedia.org/w/index.php?title=Lotka%E2%80%93Volterra_equations&oldid=1156327731.

Wong, Jian Cheng et al. (2022). "Learning in Sinusoidal Spaces with Physics-Informed Neural Networks". In: *IEEE Transactions on Artificial Intelligence*, pp. 1–15. DOI: 10.1109/tai.2022.3192362. URL: https://doi.org/10.1109%2Ftai.2022.3192362.

Xiang, Zixue et al. (2021). *Self-adaptive loss balanced Physics-informed neural networks for the incompressible Navier-Stokes equations*. arXiv: 2104.06217 [physics.flu-dyn].

Yadav, N., A. Yadav, and M. Kumar (2015). *An Introduction to Neural Network Methods for Differential Equations*. SpringerBriefs in Applied Sciences and Technology. Springer Netherlands. ISBN: 9789401798167. URL: https://books.google.com.co/books?id=9aDdBgAAQBAJ.

Yadav, Neha, Anupam Yadav, Manoj Kumar, et al. (2015). *An introduction to neural network methods for differential equations*. Vol. 1. Springer.

Yang, Liu, Dongkun Zhang, and George Em Karniadakis (2020). "Physics-informed generative adversarial networks for stochastic differential equations". In: *SIAM Journal on Scientific Computing* 42.1, A292–A317.

Zhang, Guang-Tao et al. (2021). *Deep learning based on mixed-variable physics informed neural network for solving fluid dynamics without simulation data*. arXiv: 2111.09086 [physics.flu-dyn].

Zhou, Pei-bai (1993). "Finite Difference Method". In: *Numerical Analysis of Electromagnetic Fields*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 63–94. ISBN: 978-3-642-50319-1. DOI: 10.1007/978-3-642-50319-1_3. URL: https://doi.org/10.1007/978-3-642-50319-1_3.

Zhu, Ciyou et al. (1997). "Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization". In: *ACM Transactions on mathematical software (TOMS)* 23.4, pp. 550–560.