Solving Partial Differential Equations with Neural Networks

Juan B. Pedro Costa

June 29, 2018

Solving Partial Differential Equations with Neural Networks

Trabajo final de Máster
MIARFID
Universitat Politècnica de València

Presentada al departamento:
PRLHT

Tutor:
Roberto Paredes Palacios.

Co-tutor:
Oriol Vinyals.

Abstract

Many scientific and industrial applications require solving Partial Differential Equations (PDEs) to describe the physical phenomena of interest. Some examples can be found in the fields of aerodynamics, astrodynamics, combustion and many others. In some exceptional cases an analytical solution to the PDEs exists, but in the vast majority of the applications some kind of numerical approximation has to be computed.

Numerical methods are based on the idea of domain discretization. The region of interest (usually the space and time domain) is divided in small regions where some kind of form of the solution is assumed (for example, we can consider a constant solution in each sub-region). Then, the global solution, is approximated by putting together the solution for each discretized zone. Different numerical methods exist depending on the form of the approximated solution and the final objective is to find the parameters of the discretized functions that minimize the approximation error given by the PDEs.

In this work, an alternative approach is proposed using neural networks (NNs) as the approximation function for the PDEs. Unlike traditional numerical methods, NNs have the property to be able to approximate any function given enough parameters. Moreover, this solutions are continuous and derivable over the entire domain and, hence, there is no need for discretization. Another advantage that NNs as function approximations provide is the ability to include the free-parameters in the process of finding the solution. As a result, our solution can generalize to a range of situations instead of a particular case, hence avoiding the need of performing new calculations every time a parameter is changed.

In chapter 1 we begin by introducing conservation laws, a kind of PDEs that are used in the field of Computational Fluid Dynamics (CFD). We will restrict our analysis to this kind of PDEs due to the author's background in the fluid simulation field. Numerical methods will be explained in order to present the traditional numerical approach for solving PDEs. Then NNs are introduced and their properties will be explored. Finally, a state of the art study is presented with previous works related to this topic. Our first attempt to solve PDEs with NNs is presented in chapter 2 where the one-dimensional advection equation is presented and solved. Then, the study is extended to the two-dimensional form of the same PDE in chapter 3. In chapter4, the inviscid Smith-Hutton problem is tackled with different boundary conditions. Finally, in chapter 5 an approach to solve higher-order PDEs is presented and tested in the viscous Smith-Hutton problem. This work ends with chapter 6 where the conclusions and possible further work is discussed.

Agradecimientos

...

frase célebre

Contents

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Conservation laws 2
- 1.3 Numerical Methods 3
- 1.4 Neural Networks 5
  - 1.4.1 Error backpropagation 8
- 1.5 State of the art 10
- 1.6 Conclusions 11

2 One-Dimensional Advection Equation **14**
- 2.1 Neural Network 14
  - 2.1.1 Topology 14
  - 2.1.2 Training points 16
  - 2.1.3 Loss function 16
- 2.2 Results 17
  - 2.2.1 Smooth solution 17
  - 2.2.2 Discontinuous solution 17
  - 2.2.3 NN vs FVM 22
- 2.3 Conclusions 22

3 Two-Dimensional Advection Equation **25**
- 3.1 Neural Network 25
  - 3.1.1 Topology 25
  - 3.1.2 Training points 25
  - 3.1.3 Loss function 27
- 3.2 Results 28
  - 3.2.1 Smooth solution 28
  - 3.2.2 Discontinuous solution 28
  - 3.2.3 NN vs FVM 28
- 3.3 Conclusions 33

4 Inviscid Smith-Hutton Problem **37**
4.1 Neural Network 37
4.1.1 Topology 37
4.1.2 Training points 39
4.1.3 Loss function 39
4.2 Results 40
4.3 Conclusions 40

5 Viscous Smith-Hutton Problem **46**
5.1 Neural Network 46
5.1.1 Topology 47
5.1.2 Training points 47
5.1.3 Loss function 48
5.2 Results 48
5.3 Conclusions 50

6 Conclusions & Further Work **51**
6.1 Conclusions 51
6.2 Further Work 51

Nomenclature

...

Chapter 1

Introduction

The main topic of this work is exploring the use of Neural Networks (NNs) as solution approximation to Partial Differential Equations (PDEs). In this chapter, the motivation of this work is discussed. Then, conservation laws (a form of PDEs) are presented and Numerical Methods are explained. Following, Neural Networks (NNs) are introduced and its properties are explored. Finally, a state of the art study is presented with other works found in the literature that also use NNs to solve PDEs.

1.1 Motivation

Many scientific and industrial applications require solving Partial Differential Equations (PDEs) to describe the physical phenomena of interest. Some examples can be found in the fields of aerodynamics, astrodynamics, combustion and many many more. In some exceptional cases an analytical solution to the PDEs exists, but in the vast majority of the applications some kind of numerical approximation has to be computed.

Numerical methods are based on the idea of domain discretization. The region of interest (usually the space and time domain) is divided in small regions where some kind of form of the solution is assumed (for example, we can consider a constant solution in each sub-region). Then, the global solution, is approximated by putting together the solution for each discretized zone. Different numerical methods exist depending on the form of the approximated solution and the final objective is to find the parameters of the discretized functions that minimize the approximation error given by the PDEs.

In this work, an alternative is proposed using neural networks (NNs) as the approximation function for the PDEs. Unlike traditional numerical methods, NNs have the property to be able to approximate any function given enough parameters. Moreover, this solutions are continuous and derivable over the entire domain and, hence, there is no need for discretization.

Another advantage that NNs as function approximations provide is the ability to include the free-parameters in the process of finding the solution. As a result, our solution can generalize to a range of situations instead of a particular case avoiding the need of performing new calculations every time a parameter is changed.

1.2 Conservation laws

Conservation laws arise from physical principles[5]. Consider the simplest fluid dynamics problem, in which a gas is flowing through a one-dimensional pipe with some known velocity $u(x,t)$, which is assumed to vary only with x, the distance along the pipe, and the time t. Let $\rho(x,t)$ be the density of the gas, the function that must be determined. Consider a section of pipe $x_1 < x < x_2$, the total mass of the gas in the pipe is given by

$$m(t) = \int_{x_1}^{x_2} \rho(x,t)dx \tag{1.1}$$

and, if there is no creation or destruction of gas in this section (i.e. no nuclear reactions) the total mass within this section can change only due to the flux or flow of particles through the endpoints of the section at x_1 and x_2.

$$\frac{\partial}{\partial t}\int_{x_1}^{x_2} \rho(x,t)dx = F_1(t) - F_2(t) \tag{1.2}$$

Here, $F_i(t)$ for $i = 1,2$ are the fluxes at the endpoints. This is the basic integral form of a conservation law, and the basis of conservation. Roughly speaking, the variation of mass within the section is the sum of what is entering the section plus what is leaving it. Remember that $\rho(x,t)$ has to be computed, it is therefore required to relate the flux with the variable. In this case, the flux is given by the product of the density and the velocity

$$f(x,t) = u(x,t)\rho(x,t) \tag{1.3}$$

The function $f(x,t)$ is the flux function, and $F_i(t) = f(x_i,t)$. Since the velocity is known, the flux function reads $f(\rho(x,t))$.

$$\frac{\partial}{\partial t}\int_{x_1}^{x_2} \rho(x,t)dx = f(\rho(x_1,t)) - f(\rho(x_2,t)) \tag{1.4}$$

The function $\rho(x,t)$ that satisfies 1.4 cannot been directly found, instead it is transformed into a partial differential equation that can be handled with standard techniques.

$$\frac{\partial}{\partial t}\int_{x_1}^{x_2} \rho(x,t)dx = -\int_{x_1}^{x_2} \frac{\partial}{\partial x} f(\rho(x,t))dx \tag{1.5}$$

Or, manipulating

$$\int_{x_1}^{x_2} \left[\frac{\partial}{\partial t}\rho(x,t) + \frac{\partial}{\partial x} f(\rho(x,t)) \right] dx = 0 \tag{1.6}$$

Since the integral must be zero for all values of x_1 and x_2, the final differential equation is

$$\frac{\partial}{\partial t}\rho(x,t) + \frac{\partial}{\partial x} f(\rho(x,t)) = 0 \tag{1.7}$$

Notice that the derivation of the differential equation requires both $\rho(x,t)$ and $f(\rho(x,t))$ to be smooth functions. This is true in most cases, but discontinuities may appear in the general treatment of compressible fluids. This is an important fact, because when a discontinuity appear the differential conservation equation cannot be used and the integral form must be revisited. Summarizing, the differential equation of a conservation law takes the form

$$\frac{\partial}{\partial t}\phi(x,t) + \frac{\partial}{\partial x} f(\phi(x,t)) = 0 \tag{1.8}$$

Where ϕ is called the conserved variable (like the density in the previous development), and $f(\phi(x,t))$ is the flux function of the conserved variable. The main focus will be to compute $\phi(x,t)$, given the initial and boundary conditions. In order to use a more friendly notation, equation 1.8 is expressed as

$$\phi_t + f(\phi)_x = 0 \tag{1.9}$$

Where the subscripts t and x denote the partial derivatives with respect to the time and space, respectively.

1.3 Numerical Methods

We want numerical methods to be accurate at minimal effort, flexible to solve different problems, easy to maintain and reliable. To construct a numerical method for solving PDEs we need to consider how to represent our solution by an approximate solution and in which sense will the approximate solution satisfy the PDE. We need ways to generate a system of algebraic equations from the well-posed PDE and incorporate boundary conditions. Then solve the system while minimizing unavoidable errors that are introduced in the process. Considering the one-dimensional conservation law 1.9 we discuss basic ideas, advantages and disadvantages of different classical methods.

The finite difference method (FDM) consists on representing the computational domain by a set of collocated points. The solution is represented locally as a polynomial

$$\phi(x,t) = \sum_{i=0}^{2} a_l(t)(x - x^k)^l \qquad f(x,t) = \sum_{i=0}^{2} b_l(t)(z - z^k)^l \tag{1.10}$$

The PDE is satisfied in a point-wise manner

$$\frac{d\phi(x^k,t)}{dt} + \frac{f(x^{k+1},t) - f(x^{k-1},t)}{h^k + h^{k+1}} = 0 \tag{1.11}$$

Local smoothness requirement pose a problem for resolving complex geometries, internal discontinuities and overall grid structure. Finite difference methods are simple to understand, straightforward to implement on structured meshes, high-order accurate, explicit in time and they have an extensive body of theoretical and practical work since the 60s. The main problems is their implementation on complex geometries, non-suitability for discontinuous problems and require grid smoothness.

On the other hand, finite volume methods (FVM) represent the domain with a set of non-overlapping cells, where the solution is represented locally as a cell average

$$\overline{\phi}^k = \frac{1}{h^k} \int_{w^k} \phi^k dx^k \tag{1.12}$$

The PDE is satisfied on conservation form

$$h^k \frac{d\phi^k}{dt} + f(x^{k+1/2},t) - f(x^{k-1/2},t) = 0 \tag{1.13}$$

A flux function needs to be reconstructed on cells interfaces

$$f(x^{k+1/2}) = F(\phi^k, \phi^{k+1}) \tag{1.14}$$

Finite volume methods are robust, support complex geometries, are well suited for hyperbolic problems, methods are local and explicit in time, locally conservative (due to telescopic property) and an extensive theoretical background exists since the 70s. Their main problems are the inability to achieve high-order accuracy on general grids and grid smoothness is required.

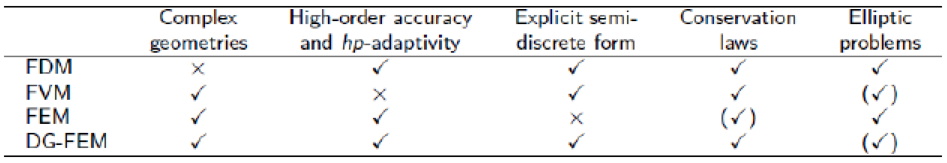
Another formulation, known as finite element method (FEM), consists on representing the domain by non-overlapping elements where the solution is represented globally with piecewise continuous polynomials.

$$\phi(x) = \sum_{i=1}^{K} u(x_k,t) N^k(x) \tag{1.15}$$

The PDE is satisfied in a global manner

$$\int_{\Omega_h} (\phi_t + f_x) N^j(x) dx = 0 \tag{1.16}$$

The semi-discrete scheme is implicit by construction and reduces overall efficiency for explicit time-integration. Finite element methods are robust, support unstructured meshes, are high-order, well suited for elliptic problems (due to the global statement) and extensive theoretical framework exists



	Complex geometries	High-order accuracy and *hp*-adaptivity	Explicit semi-discrete form	Conservation laws	Elliptic problems
FDM	×	✓	✓	✓	✓
FVM	✓	×	✓	✓	(✓)
FEM	✓	✓	×	(✓)	✓
DG-FEM	✓	✓	✓	✓	(✓)

Figure 1.1: Numerical methods comparison.

since the 70s. Their main disadvantages are that they are not well suited for hyperbolic problems (due to directionality) and they are implicit in time (reducing overall efficiency).

An old methodology that is seeing success recently is the so called Discontinuous Galerkin Methods (DGM), and the more advanced $P_N P_M$ methods [6]. This class of methods represent a combination of FEM and FVM that take advantage of the local statement and geometrical flexibility of FVM redefining the cell averaged nature by the local high-order formulation of FEM. Briefly, the computational domain is subdivided into non-overlapping elements as in FVM and FEM. The global solution is represented using local high-order polynomials similar to FEM. Elements are then connected with numerical fluxes at elements interfaces as in FVM. DGM are arbitrary high-order schemes, locally conservative, flexible, explicit in time, locally adaptive ($hp-$refinement) and well-suited for hyperbolic problems. The main problem is its higher computational cost and relative lack of theoretical background compared with the other methods.

1.4 Neural Networks

Neural networks are inspired in the human ability to recognize patterns. They consist of a densely interconnected set of elemental processors with the objective to output some information such as a class label to an object, a vector (regression) or something more structured such as a sentence. We will use NNs to output the solution to a PDE. Nowadays we have available multiple tool boxes with all the functionality that NNs provide already implemented and tested. Some examples are Layers (github.com/RParedesPalacios/Layers), Tensorflow (tensorflow.org) or the one we will use in this work, Pytorch (pytorch.org).

The type of NNs that we are going to explore are also known as Multilayer Perceptron (MLP). They consist of multiple layers of simple Perceptron architecture allowing the representation of complex structures. Activation units are used to overcome the linear limitations if the Preceptron.

The basic elemental processor of the MLP, also called neuron, is depicted in figure 1.2. The neuron receives a series of values coming from the input layer or the output of other neurons. Each of these values are weighted and

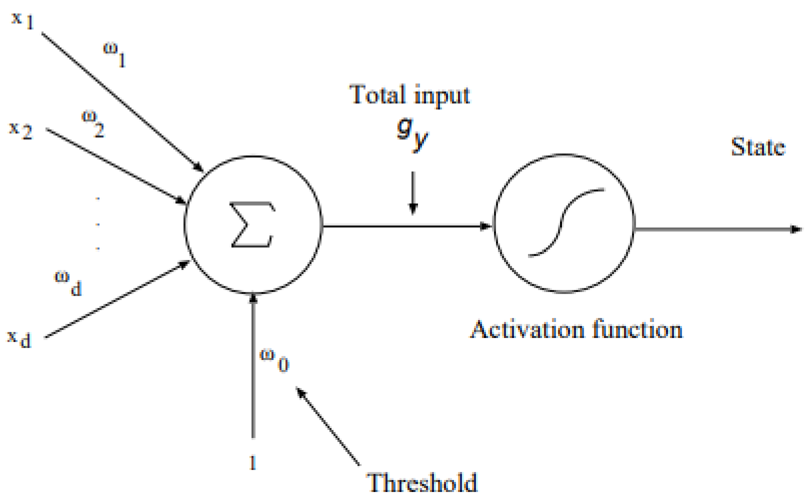


Figure 1.2: Logistic linear discriminant functions.

summed together, and a bias is added. The result is then filtered with an activation function, resulting in the output.

$$s(\mathbf{x}) = f(\sum_{k=1}^{D} w_k x_k + w_0) = f(\mathbf{w}^t \mathbf{x}) \tag{1.17}$$

where $\mathbf{x} = (1, x_1, ..., x_D)$ are the inputs, $\mathbf{w} = (w_0, w_1, ..., w_D)$ are the weights and bias. Multiple activation functions exist, but we will limit ourselves to Relu (rectified linear units) for first order PDEs and Sigmoid for second order, which are defined in equations 1.18 and 1.19 respectively.

$$f(x) = max(0, x) \tag{1.18}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1.19}$$

A two-layer Perceptron is shown in figure 1.3. As it can be seen it consists of 5 input units that are connected to 3 neurons in the first hidden layer (each unit with its own weights and biases). Then, the outputs of these neurons are also connected to three neurons in the output layer. For this simple case we can find the output of our MLP as

$$g(\mathbf{x}, \mathbf{w}) = f(\mathbf{w}_2^t f(\mathbf{w}_1^t \mathbf{x})) \tag{1.20}$$

Our objective will be to find the set of weights that, given as an input the independent variables of the PDE, outputs the solution to the equation. To

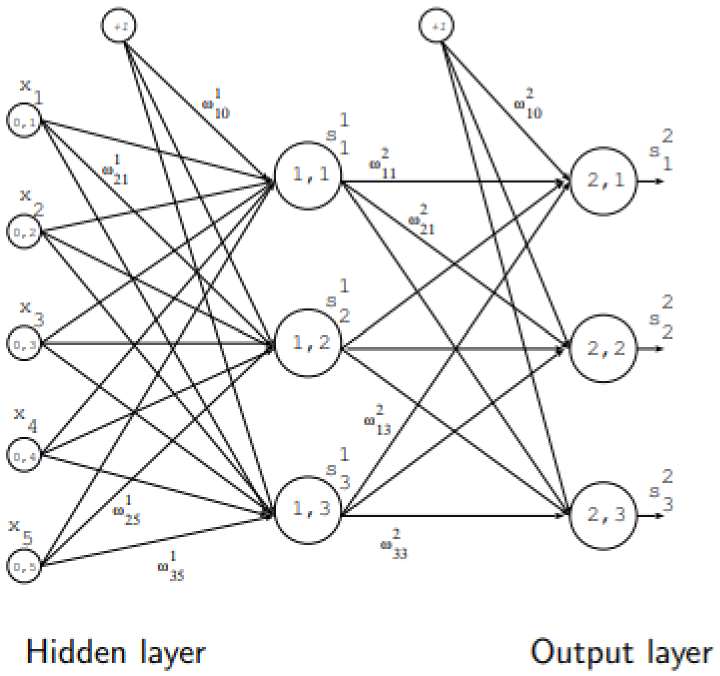


Figure 1.3: A two layer Perceptron.

do so, we use an algorithm known as error backpropagation. Consider a general conservation law

$$\phi_t + f(\phi)_x = 0 \tag{1.21}$$

Where $\phi(x,t)$ is the solution to the PDE and (x,t) are the independent variables. Given a topology of a MLP and $A = (x_1,t_1),...(x_N,t_N)$, we want to search for $\mathbf{w}$ such that minimizes an objective function. For regression problems, such as the one we are interested in, a mean squared error function is used.

$$E_A(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}(L(\mathbf{x}_n,\mathbf{w}))^2 \tag{1.22}$$

where $L(\mathbf{x}_n,\mathbf{w})$ is the loss function. If we know the output of the NN (for example for the initial condition or the boundary conditions) we can define a loss function as

$$L(\mathbf{x}_n,\mathbf{w}) = g(\mathbf{x}_n,\mathbf{w}) - \phi(\mathbf{x}_n) \tag{1.23}$$

with this loss function, the NN will adjust the weights to match our solution. Nevertheless, our interest is to obtain the solution in all the domain. The

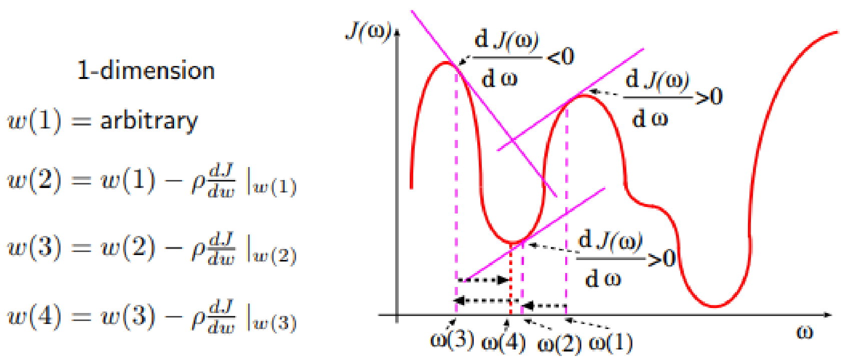

1-dimension
$w(1) = \text{arbitrary}$
$w(2) = w(1) - \rho \frac{dJ}{dw}|_{w(1)}$
$w(3) = w(2) - \rho \frac{dJ}{dw}|_{w(2)}$
$w(4) = w(3) - \rho \frac{dJ}{dw}|_{w(3)}$
$J(\omega)$
$\frac{d\,J(\omega)}{d\,\omega} < 0$
$\frac{d\,J(\omega)}{d\,\omega} > 0$
$\frac{d\,J(\omega)}{d\,\omega} > 0$
ω(3) ω(4) ω(2) ω(1)
ω

Figure 1.4: Error backpropagation example.

loss function that we will use to achieve that objective is the PDE itself.

$$L(\mathbf{x}_n, \mathbf{w}) = \phi_t(\mathbf{x}_n) + f(\phi(\mathbf{x}_n))_x \tag{1.24}$$

If we can ensure that the error introduced with this loss function is close to zero, we can assume that the output of the NN is indeed the solution to our PDE. An issue remains open, and it is how to compute the derivative of the output with respect to the inputs in order to build the loss function. Luckily, the error backpropagation algorithm solves this problem for us.

1.4.1 Error backpropagation

The error backpropagation algorithm consist on computing the error derivative with respect to the weights, allowing a weight update in the direction that minimizes the error

$$\Delta w_{ij}^k = -\rho \frac{\partial E_A}{\partial w_{ij}^k} \tag{1.25}$$

where ρ is the learning rate. An example of the algorithm can be found in figure 1.4. Starting from a random value for the weight, computing the error derivative allows to its update in such a way that after several iterations of the algorithm local minima of the error function are found. Small values of the learning rate result in a slow but steady optimization, while big values can produce the divergence of the process. It will depend ultimately on the search space. The fact that the backpropagation algorithm finds only local minima is one of the disadvantage of this method. Nevertheless it has proved to be reliable and good enough for multiple applications.

As it was previously mentioned we are going to use Pytorch. This framework has already implemented all the required functionality to perform error backpropagation and to compute the derivatives of the NN output with respect to the input variables. An example of the implementation (Pytorch v0.4.0) to solve our simple example is presented now

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
dtype = torch.FloatTensor

# Define a NN with two hidden layers
class Net(nn.Module):
    def __init__(self, H):
        super(Net, self).__init__()
        D_in, H1, D_out = H
        # The linear layer corresponds to the perceptron architecture
        self.linear1 = nn.Linear(D_in, H1)
        self.linear2 = nn.Linear(H1, D_out)
    def forward(self, x):
        # Use relu activation functions for the hidden layer
        x = F.relu(self.linear1(x))
        # No activation is used in the output layer (regression)
        y_pred = self.linear2(x)
        return y_pred

# two inputs, 16 neurons in the hidden layer, 1 output
H = 2, 16, 1
net = Net(H)

# random input (x, t)
x = torch.randn(2)
x = Variable(x.type(dtype), requires_grad=True)

# the output will be a random number
p = net(x)

# compute the output derivative w.r.t inputs
grads, = torch.autograd.grad(p, x,
                grad_outputs=p.data.new(p.shape).fill_(1),
                create_graph=True, only_inputs=True)
dpdx, dpdt = grads[0], grads[1]

# The loss function is the PDE (for example, dpdt + dpdx = 0)
loss = dpdt + dpdx

# create your optimizer with a learning rate
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
```

```
# update the weights
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

If we repeat the process until the loss reaches near-zero values, that means we have been able to find an approximated solution to our PDE. Once again Pytorch helps us with its built-in functionality.

It is important to note that in many cases to close the problem an initial condition and boundary conditions are required. We will need to define the points that belong to these conditions and used separately to train the NN with a different loss. The process to solve our problem will in the end involve as many loss functions as PDEs in the system we want to solve plus all the initial conditions and boundary conditions required to close the problem. In some cases this can result in a lot of functions to be optimized making it very difficult to find an optimal set of weights that match all the requirements.

Another important issue worth discussing is the fact that, when computing derivatives, the derivatives of the activation functions are required. The derivative of the Relu function is 1 or 0. This means that if second-order derivatives are required, Relu activation functions will always result in 0. This is the reason why Sigmoid activation functions are used in this cases, since they have arbitrary order defined derivative.

In the next chapter we are going to explore how the presented method apply in different cases in order to explore its capabilities, performance and limitations.

1.5 State of the art

In this subsection we are going to explore the most relevant works concerning the use of NNs to solve PDEs. Traditional methods such as FEM, FVM and FDM rely on discretizing the domain and weakly solving PDEs over the discretization. This result in discrete solutions with exponential cost growth with the number of points and dimensionality of the problem (a continuous solution would require infinite time). Furthermore, free parameters such as the Reynolds or Mach numbers, the geometry and the boundary conditions require new simulations every time we change them. We will see how using NNs allows us to obtain analytical solutions which are continuous over all the domain, that include all the free-parameters as part of the solution and that once trained are able to give accurate results instantly dramatically accelerating the optimization process.

Chiaramonte and Kiener [2] use a NN with a single hidden layer and a trial solution to obtain continuous differentiable solution to PDEs. They use the independent variables and a bias as input parameters, and one single output with the optimal values of the trial solution that satisfies the PDE.

They test it in the Laplace's equation and a conservation law. The results show relatively small errors when compared to the analytical solutions. The advantage of this method is that the obtained solution is a smooth approximation that can be evaluated and differentiated continuously on the domain. Several areas of improvement include adaptive training set generation to reduce training costs and the study of non-uniform discretizations.

Parisi et al. [4] use an unsupervised approach to train a NN to solve PDEs. They take advantage of the universal approximation capabilities of NNs to postulate them as a solution for a given PDE. A single hidden layer perceptron is used as a generic function. The weights are then found by gradient descent optimisation using the original PDE and a set of sample points as error function, using a genetic algorithm for their initialisation. They compared their solution with a traditional method in an unsteady solid-gas reactor problem which relied in spatial discretization obtaining similar accuracy results but at a fraction of the time since once the NN is trained it can find the solution at any given point instantaneously.

Bayman et al. [1] use a NN to solve NS equations. An analytical solution formed by two parts (one that satisfies boundary conditions and other for the internal domain) is found via optimization in a feed-forward network with two hidden layers. Results obtained with this method for a two-dimensional steady problem show good agreement with existing data with smaller errors compared to traditional numerical methods. Furthermore, the solution generated by the NN can be reused at any time.

In [7] Sirignano and Spiliopoulos develop an algorithm similar to Galerkin methods in order to approximate high-order PDEs. Their method is meshless, and the NN is trained to satisfy the differential operators and boundary conditions using stochastic gradient descent at randomly sampled spatial points. A similar work is presented in Han et al. [3] who also studied the use of NNs to approximate high-dimensional PDEs.

1.6 Conclusions

We can conclude that the advantages that solving PDEs with NNs present, compared to traditional methods, are:

- Analytic solution over the domain, not piecewise discrete (mesh-less).
- Computational complexity does not increase with the number of sampling points.
- Free parameters can be included in the solution, avoiding repeating simulations at different conditions.
- Once the NN is trained, it can be reused to obtain results instantly.

On the other hand, the main disadvantage of this methodology is the expensive training. Nevertheless, its advantages overcome this disadvantage since once the train is completed, the NN can be reused over and over. As an example of the potential of this method we can consider the design of an airfoil. Traditionally, some analytical method would be used to estimate the geometry of the airfoil in order to achieve some performance criteria (a lift or drag value, for example). Then, a scaled model would be built and tested in a wind tunnel to validate the model. Some iterations of this process would be required in order to achieve the initial goal, since simplified models does not account for many phenomena that take place in the real testing (turbulence, acoustics and others). This process is expensive and takes a lot of time. A step forward was introduced with the use of CFD. Some wind tunnel experiments could be replaced by virtual simulations. Thanks to the rise of the computational power available, more sophisticated models could be used such as solving the three-dimensional NS equations. Accurate simulations are still prohibitive nevertheless, and so the process of optimization is still slow. The use of NNs can disrupt the simulation environment as follows:

- We can train a NN to solve the three-dimensional NS equations over an airfoil, obtaining an accurate continuous solution over the entire domain without the need of a mesh (which is the main limiting factor nowadays).
- We can include the design parameters in the solution as NN inputs (Reynolds, Mach, airfoil chord, AoA, etc).
- We can train the NN in an unsupervised way, no train data is required (the solution is obtained requiring it to satisfy the NS system of equations and the boundary conditions).
- Once the solution is found (which may be as expensive as one or several traditional simulations) we can compute any solution at any point with any set of free parameters instantly, dramatically speeding up the process of optimization and also allowing real-time simulations.

Bibliography

[1] M. Baymani, S. Effati, H. Niazmand, and A. Kerayechian. Artificial neural network method for solving the navier stokes equations. 26(4):765–763, 2015.

[2] M. M. Chiaramonte and M. Kiener. Solving differential equations using neural networks.

[3] Jiequn Han, Arnulf Jentzen, and Weinan E. Overcoming the curse of dimensionality: Solving high-dimensional partial differential equations using deep learning. 2017.

[4] Daniel R. Parisi, María C. Mariani, and Miguel A. Laborde. Solving differential equations with unsupervised neural networks. 42(8-9):715–721, 2003.

[5] J. Randall and Leveque. *Finite Volume Methods for Hyperbolic Problems.* Cambridge University Press, 2002.

[6] Lei Shi, Z. J. Wang, Song Fu, and Laiping Shang. PN PM-CPR method for navier stokes equations. 2012.

[7] Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. 2017.

Chapter 2

One-Dimensional Advection Equation

In this chapter the first attempt to solve a PDE with a NN is presented. The one-dimensional advection equation is chosen because it is a simple PDE with analytical solution. This means that we can easily compare the solution provided by our NN with the real one. The one-dimensional advection equation has the form

$$\phi_t + u\phi_x = 0 \tag{2.1}$$

If we compare equations 2.1 and 1.9 we can see that the advection equation is a conservation law where ϕ is the conserved variable and the flux $f(\phi) = u\phi$ where u is a known constant. We can think of the advection equation as the expression that describes the time evolution of a dye in a background velocity field. Given an initial condition $\phi(x, t = 0)$ we can find the analytical solution only by transporting the initial condition in x at speed u.

$$\phi(x, t) = \phi(x - ut, t) \tag{2.2}$$

Let's say, for example, that $\phi(x, t = 0) = sin(2\pi x/L)$ where $x \in [0, L]$. Then, we can find the solution to our PDE at any given time as $\phi(x, t) = sin(2\pi(x-ut)/L)$. See figure 2.1 where the solution for this case is presented at $t = 0.5$ and different values of u. Since $u = x/t$, in the case where $u = 1$ the solution moves in x 1 dimension unit each time unit.

2.1 Neural Network

2.1.1 Topology

For this case, we are going to use a neural network with 2 input units (x, t) and 1 output unit, $\phi(x, t)$. Different combination of hidden layers and hidden units were studied, and results for a specific topology are presented in the next section. Relu are used as activation functions.

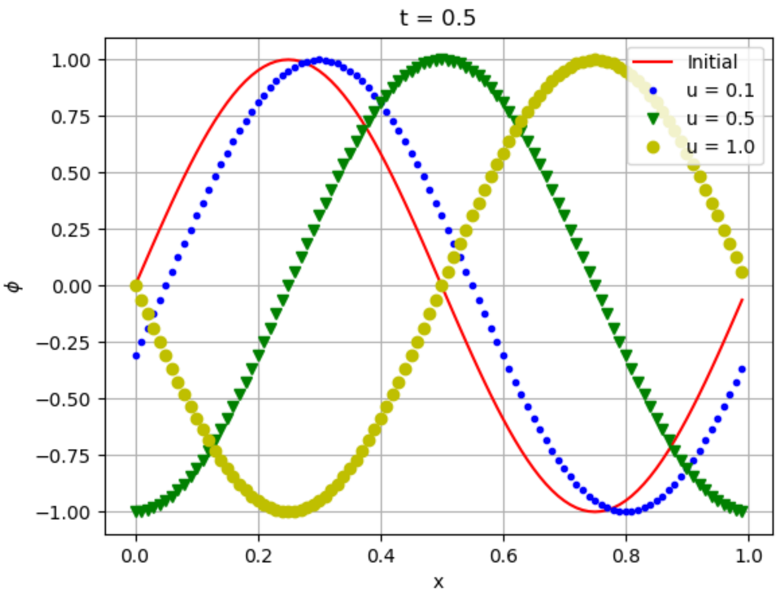


Figure 2.1: Example of the solution to the one-dimensional equation with the initial condition $sin(2\pi x)$.

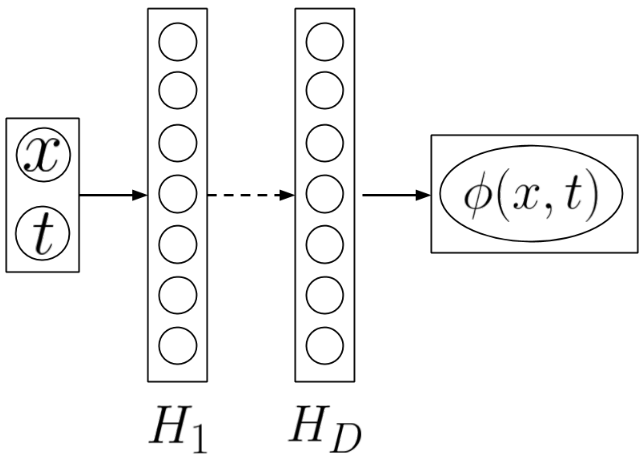


Figure 2.2: NN topology used for solving the one-dimensional advection equation.

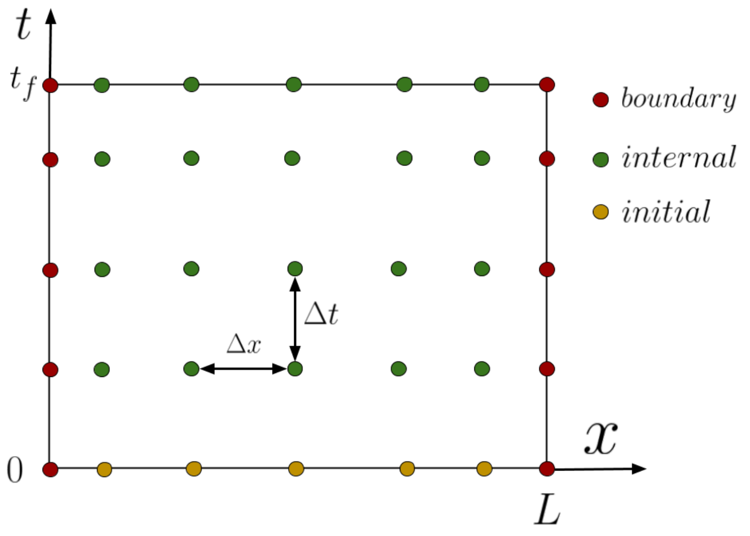


Figure 2.3: Example of training points for $N = 5$ and $M = 4$.

2.1.2 Training points

In order to build the training data we first define a set of points in which we are going to evaluate our solution during the training process. We define $N + 2$ points in x (N interior points and 2 points at the boundaries) and $M+1$ points in t (M points for each time step and 1 for the initial condition). In total, there are $(M + 1)(N + 2)$ points. Then we divide this points in three groups:

- NM points to evaluate the PDE in the internal points.
- N points at $t = 0$ to evaluate the initial condition.
- $2(M + 1)$ points to evaluate periodic boundary conditions.

In order to build periodic boundary conditions we need to match the solution at $x = 0$ with the solution at $x = L$ at any given t. An example of the points generated for the training process can be seen in figure 2.3.

2.1.3 Loss function

Each set of points has their own loss function to minimize, so the final loss function is composed by three different losses:

- The output of the network for the initial condition training points is used along with the known initial condition to build a Mean Square Error loss function to be minimized.

- The outputs of the network for the corresponding periodic boundary conditions training points are used to build a Mean Square Error loss function to be minimized.

- The output of the network for the internal points is derived with respect to the inputs, obtaining ϕ_t and ϕ_x. A loss function that matches the PDE is built. Since $\phi_t + u\phi_x$ has to be 0 for every x and t we simply minimize this loss function.

If the triple loss function reduces to 0 during the training process, we can assume that any result that we obtain with the NN for any arbitrary pair (x, t) included in the training set satisfies the initial condition, boundary conditions and the PDE. Thus, we have an approximate solution for our PDE that is continuous and derivable throughout the entire domain. We can get the solution to the PDE at any desired point with a simple forward pass through the network.

2.2 Results

After some tests we decided to go with a NN with 5 hidden layers. Each layer consists of 32 units with Relu activations. A lot of configurations were studied and here we present results for two different conditions, a sinus and a hat function. All the results presented were obtained in a domain $(x, t) \in [0, 1]\text{x}[0, 1]$ with $u = 1$. The training process was conducted during 5000 epochs with the Adam optimizer and a learning rate of 0.01.

2.2.1 Smooth solution

In figures 2.4 to 2.6 results for the sinus initial condition are presented using a time step of 0.05 and different values of N. Overall, results are in good agreement with the exact solution for the different considered meshes.

2.2.2 Discontinuous solution

In figure 2.7 results for the hat initial condition are presented. As we can see we also obtain moderate good results, but since we are trying to approximate a discontinuous solution the final results are not as good as the previous smooth condition. Nevertheless, in all the cases we are able to obtain a solution for our PDE that is correct in the physical sense (the initial condition is transported at the correct speed) but with an accuracy highly dependent on the loss achieved at the end of the training. Several techniques have been tried to achieve better results, such as re-training the network with a lower learning rate, try to use more hidden layers, etc. We can conclude that the lower the loss achieved during the training process, the better approximation it is obtained.

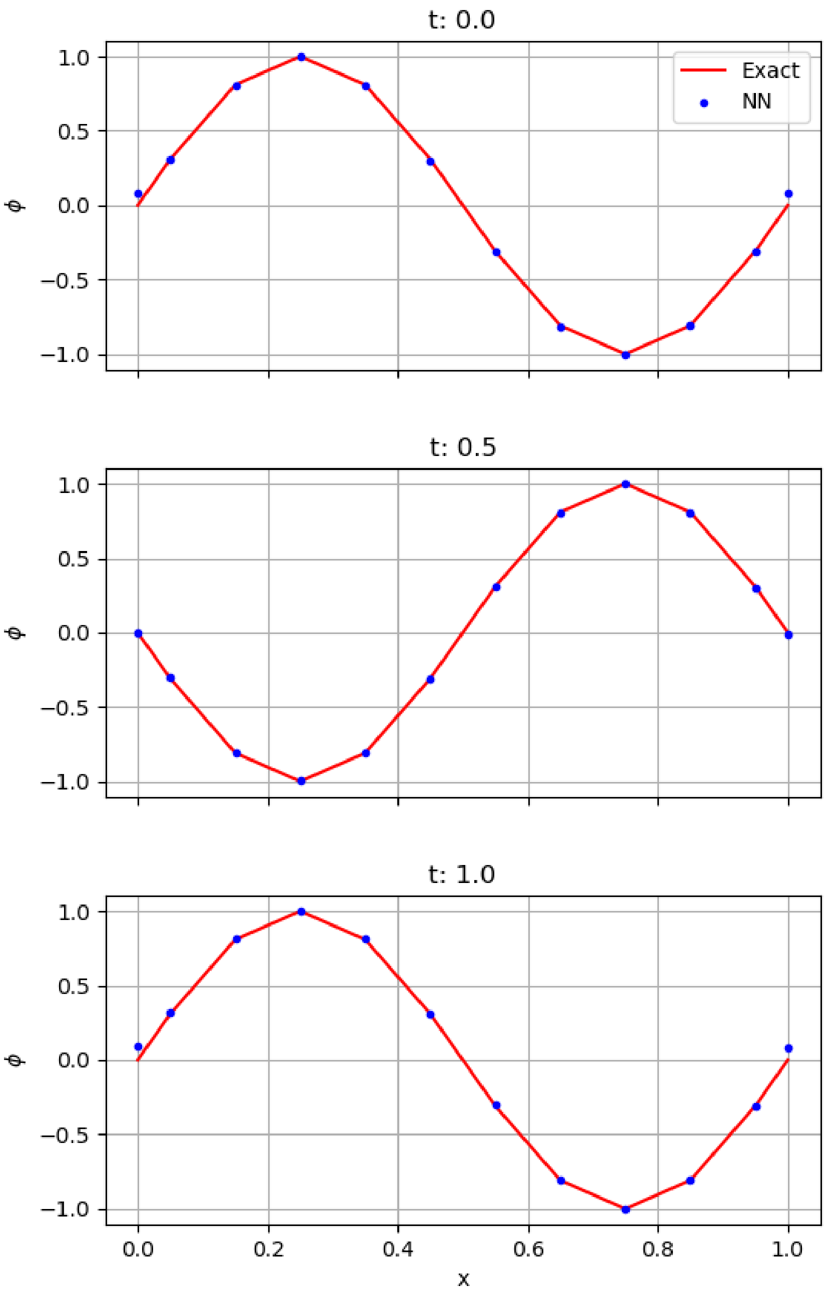


Figure 2.4: Sinus initial condition and $N = 10$.

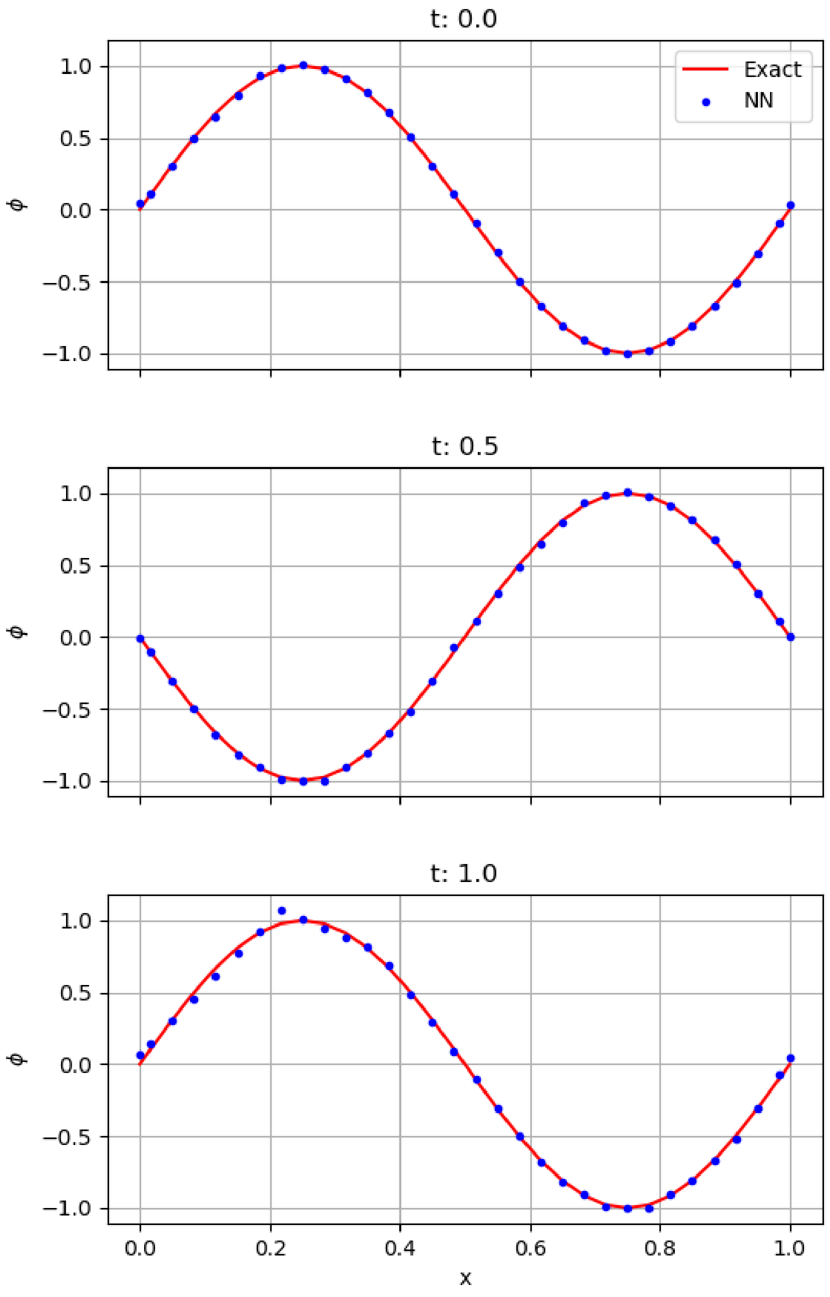


Figure 2.5: Sinus initial condition and $N = 20$.

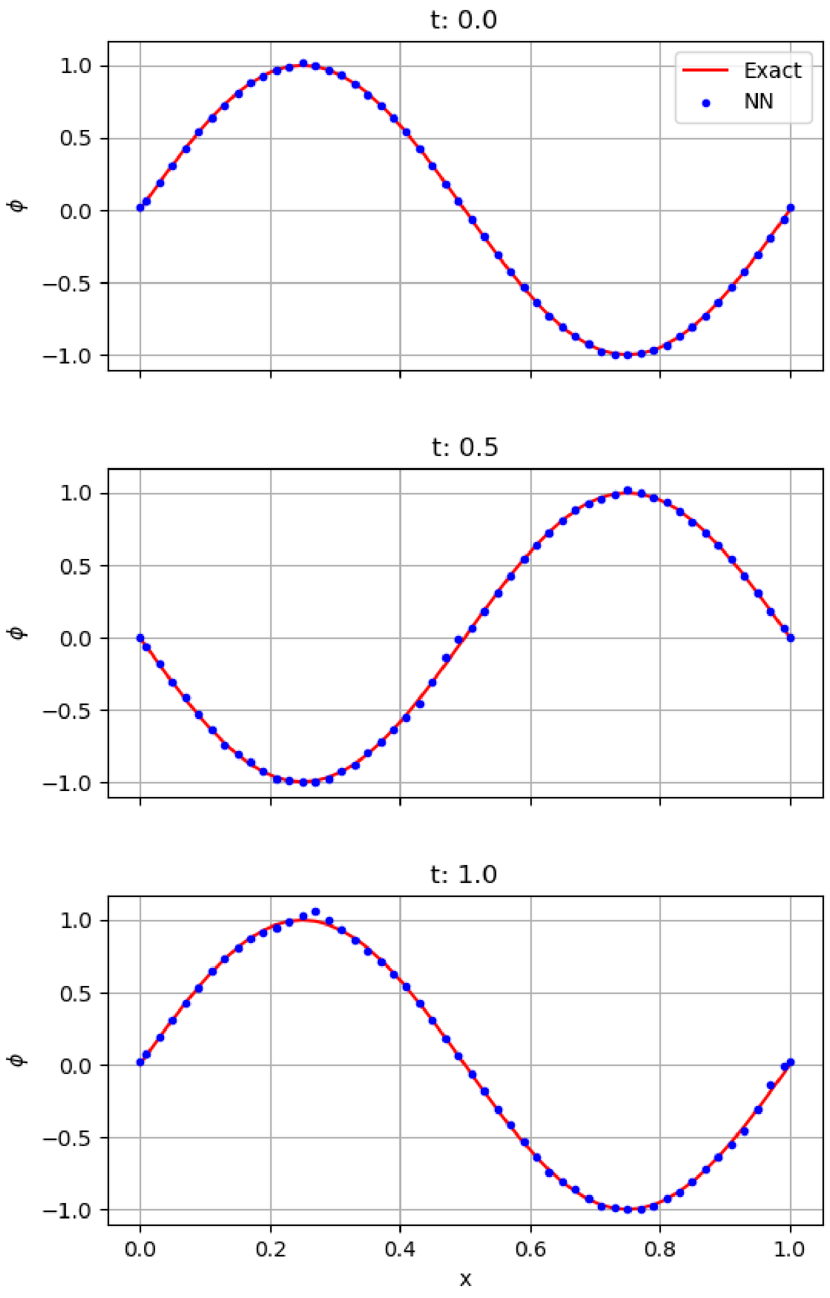


Figure 2.6: Sinus initial condition and $N = 50$.

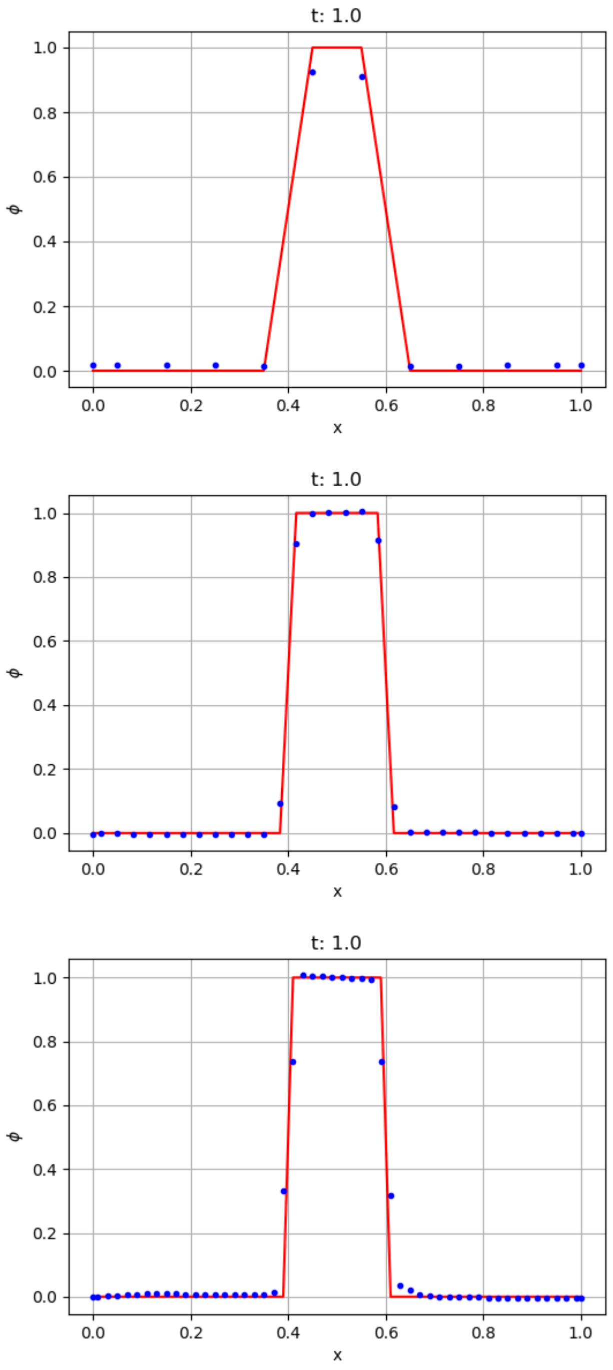


Figure 2.7: Hat initial condition and $N = 10$ (top), 30 (middle) and 50 (bottom).

2.2.3 NN vs FVM

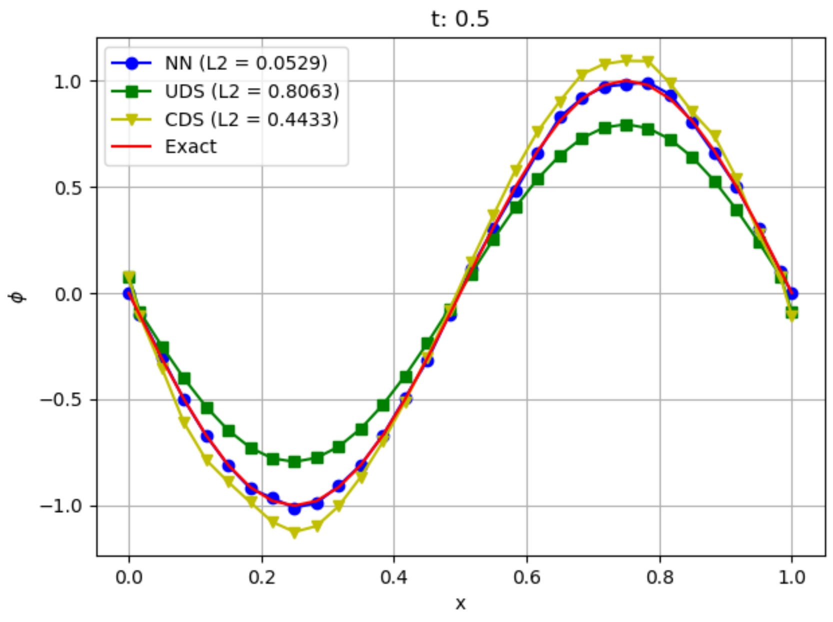
We are going to compare the performance of the presented method with respect to traditional finite volume methods. In figure 2.8 the same results are presented obtained with three different methods. All of them are compared against the exact solution in terms of the $L2$ error. A mesh with 30 elements has been used and the simulation has been run with a time step of 0.01 during 1 time unit and $u = 1$. We have used two different numerical schemes for the finite volume solutions: upwind (UDS) and central (CDS) schemes. Upwind schemes suffer from numerical diffusion, which can be seen in the figure. On the other hand, central schemes suffer from error accumulation which results in the simulation blow-up. Usual techniques used to alleviate this issues consist of high-order spatial numerical schemes to reduce the diffusion of UDS and high-order integration schemes to reduce the error accumulation of the CDS. Without requiring extra tuning, our method is able to obtain a more accurate solution in all the domain. Moreover, we can reuse our NN to obtain results with different meshes while the use of FVM would require new simulations. Also, the solution provided by the NN is derivable in the entire domain, which is not true for the FVM results since they consist of piece-wise functions. An important aspect to be considered is that in order to obtain solutions with FVM the time step was required to be reduced by a factor of 5 due to physical limitations. This constrain is not required for our approach, hence the NN can provide more accurate solutions using bigger time steps.

2.3 Conclusions

In this chapter the one-dimensional advection equation has been solved using a NN for two different initial conditions (smooth and discontinuous). After several experiments, a NN with 5 hidden layers with 32 hidden units in each layer was selected to perform the experiments. Three sets of training points have been constructed to include the initial condition, the boundary conditions and the internal solution. Each set has its own loss function to be minimized.

First of all, we can conclude that our approach to solve the PDE is able to obtain results that are correct in the physical way. The initial condition is transported at the correct velocity as it is predicted by the analytical solution. The error, however, depends on the final loss achieved during the training process. The closer to 0 the loss goes, the less error compared to the analytical solution is observed.

A study of the impact of the number of training points considered has been carried out. It is concluded that, for this simple case, the amount of points used in the spatial domain is not very important since all the meshes achieved a good solution whenever the loss was able to reduce to near zero



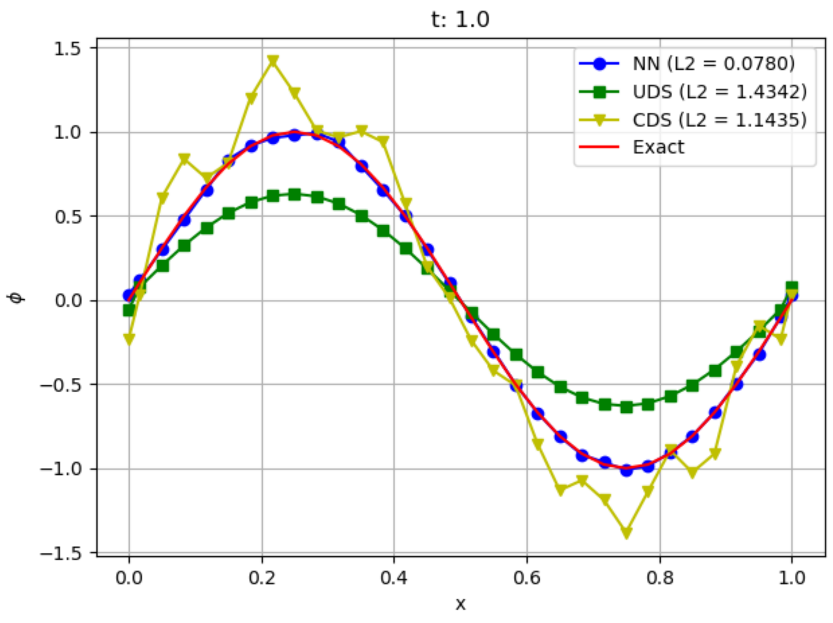


Figure 2.8: Comparative between the NN and FV methods.

values. This fact strengthen our previous conclusion that the accuracy of the solution ultimately depends on the ability of the network to find the optimal weights. In more complicated configuration this fact can be affected by the domain discretization, but that is not the case for the experiments conducted in this chapter.

Finally, a comparative between the presented method and traditional finite volume methods has been presented. The accuracy of the obtained solutions with the different methods has been compared. The NN is able to provide much more accurate solutions than traditional FVM at lower computational cost (big meshes and big time steps). This is due to the fact that FVM are constrained by physical limitations concerning the exchange of information between cells. NNs, on the other hand, are not affected by this constrains giving them the ability to compute more accurate solutions using bigger time steps and less points. The $L2$ error obtained with the NN is approximately the error achieved during the training phase. This means that NN allow to evaluate in real-time the accuracy of the solution while training. Moreover, any technique used to decrease the loss during the training (increase the number of hidden layers or units, for example) directly impacts the accuracy of the solution.

Chapter 3

Two-Dimensional Advection Equation

In this chapter we are going to extend the same analysis than in the previous one but with an additional spatial dimension. The two-dimensional advection equation has the form

$$\phi_t + u\phi_x + v\phi_y = 0 \tag{3.1}$$

Like before, this PDE has analytical solution

$$\phi(x, y, t) = \phi(x - ut, y - vt, t) \tag{3.2}$$

The initial condition is transported over the (x, y) plane at the velocity (u, v). An example of the solution to this PDE with a sinusoidal initial condition can be seen in figure 3.1.

3.1 Neural Network

3.1.1 Topology

Since now we have a second spatial dimension, we are going to use a neural network with 3 input units (x, y, t) and 1 output unit, $\phi(x, y, t)$ (see figure 3.2).

3.1.2 Training points

In order to build the training set we first define a set of points in which we are going to evaluate our solution during the training process. We define N_x+2 points in x (N_x interior points and 2 points at the boundaries), N_y+2 points in y and $M+1$ points in t (M points for each time step and 1 for the initial condition). In total, there are $(N_x+2)(N_y+2)(M+1)$ points. Then we divide this points in three groups:

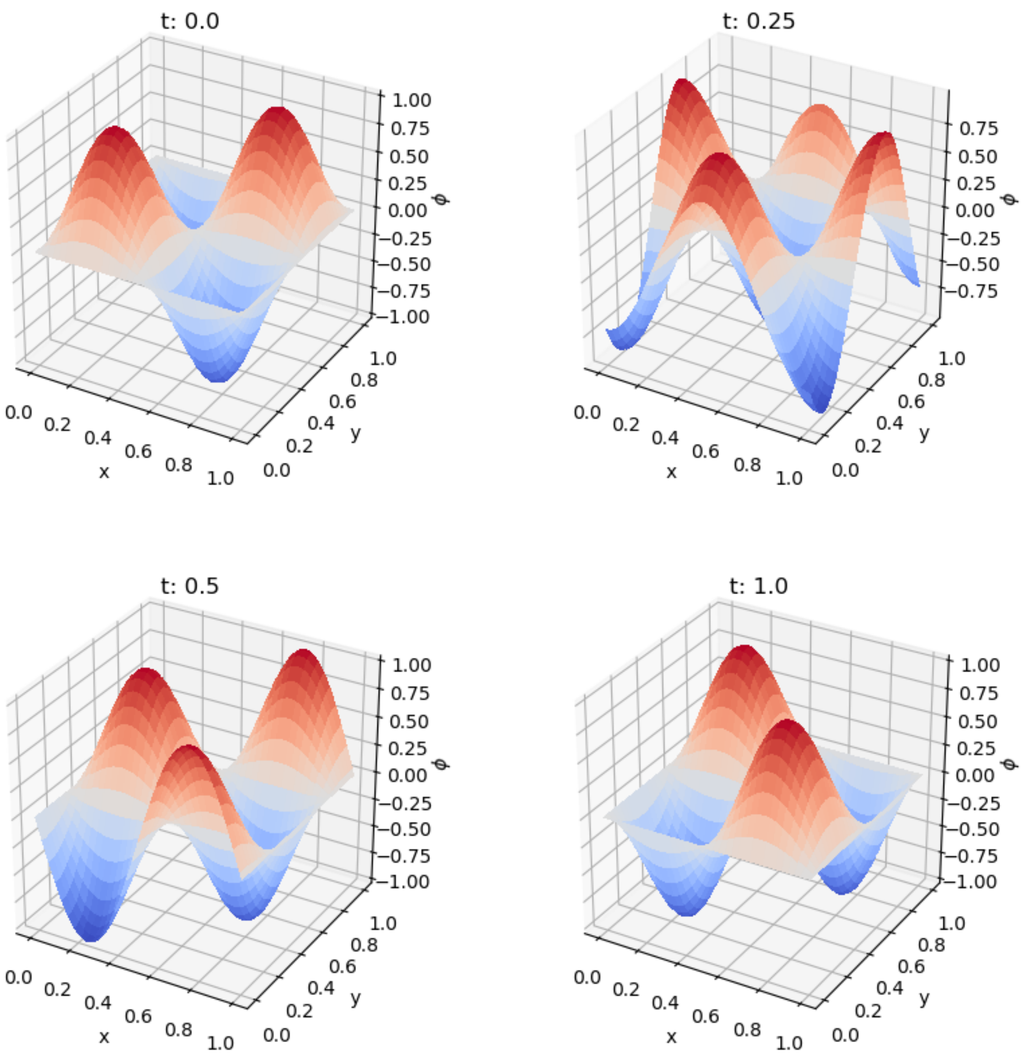


Figure 3.1: Example of the solution to the two-dimensional equation with the initial condition $\phi(x, y, 0) = sin(2\pi x) * sin(2\pi y)$ and $(u, v) = (1.0, -0.5)$.

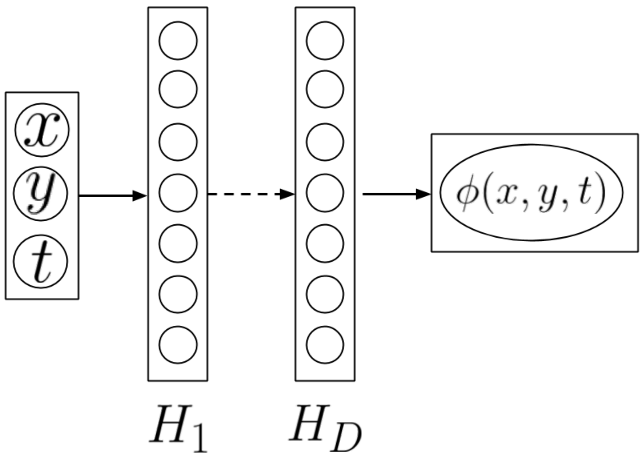


Figure 3.2: NN topology used for solving the two-dimensional advection equation.

- $N_x N_y M$ points to evaluate the PDE in the internal points.
- $N_x N_y$ points at $t = 0$ to evaluate the initial condition.
- $(2N_x + 2N_y)(M + 1)$ points to evaluate periodic boundary conditions.

In order to build periodic boundary conditions we need to match the solution at $x = 0$ with the solution at $x = L_x$ at any given t, and the same for $y = 0$ and $y = L_y$.

3.1.3 Loss function

Each set of points has their own loss function to minimize, so the final loss function is composed by three different losses:

- The output of the network for the initial condition training points is used along with the known initial condition to build a Mean Square Error loss function to be minimized.
- The outputs of the network for the corresponding periodic boundary conditions training points are used to build a Mean Square Error loss function to be minimized.
- The output of the network for the internal points is derived with respect to the inputs, obtaining ϕ_t, ϕ_x and ϕ_y. A loss function that matches the PDE is built. Since $\phi_t + u\phi_x + v\phi_y$ has to be 0 for every x, y and t we simply minimize this loss function.

If the triple loss function reduces to 0 during the training process, we can assume that any result that we obtain with the NN for any arbitrary point (x, y, t) included in the training set satisfies the initial condition, boundary conditions and the PDE. Thus, we have an approximate solution for our PDE that is continuous and derivable throughout the entire domain. We can get the solution to the PDE at any desired point with a simple forward pass through the network.

3.2 Results

After some tests we decided to go with a NN with 5 hidden layers. Each layer consists of 512 units with Relu activations. A lot of configurations were studied and here we present results for two different conditions, a sinus and a hat function. All the results presented were obtained in a domain $(x, y, t) \in [0, 1]$x$[0, 1]$x$[0, 1]$. The training process was conducted during several epochs with the Adam optimizer and learning rates between 0.0001 and 0.001. As opposed to the training process of the one-dimensional advection equation, mini batch training was used for this case. The reason can be found in the considerable increase in the number of points required to solve the PDE. Each batch, the entire initial condition and boundary conditions are considered in the loss function and only 100 internal points are used. This is done because the amount of internal points can be several order of magnitudes bigger than the other two training sets. If all the points are considered, the NN overfit to the internal solution while not fitting to the initial and boundary conditions.

3.2.1 Smooth solution

In figure 3.3 results for a 30 x 30 mesh are presented using a time step of 0.01 for a sinus initial condition with $(u, v) = (1, -0.5)$. In figure 3.4 same results are prestend but for a 50 x 50 mesh. In all cases we can see how the NN is able to reproduce results in good agreement with the exact solution.

3.2.2 Discontinuous solution

In figures 3.6 and 3.6 results for a 30 x 30 and a 50 x 50 mesh are presented, respectively, using a hat initial condition with $(u, v) = (1, -1)$ and a time step of 0.01. In all cases we can see how the NN is able to reproduce results in good agreement with the exact solution.

3.2.3 NN vs FVM

As we did in the previous chapter, we are going to compare the performance of the presented method with respect to traditional finite volume methods.

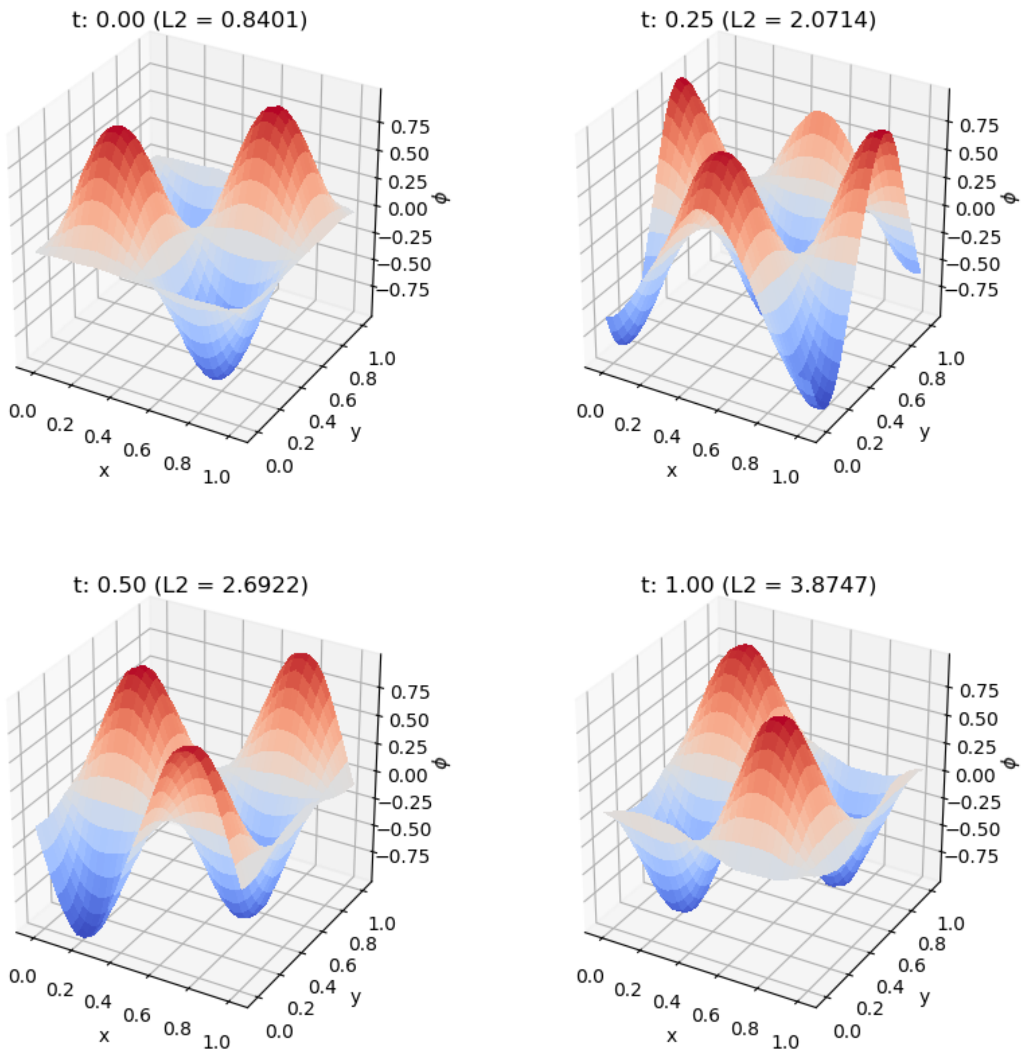


Figure 3.3: Results with a 30x30 mesh.

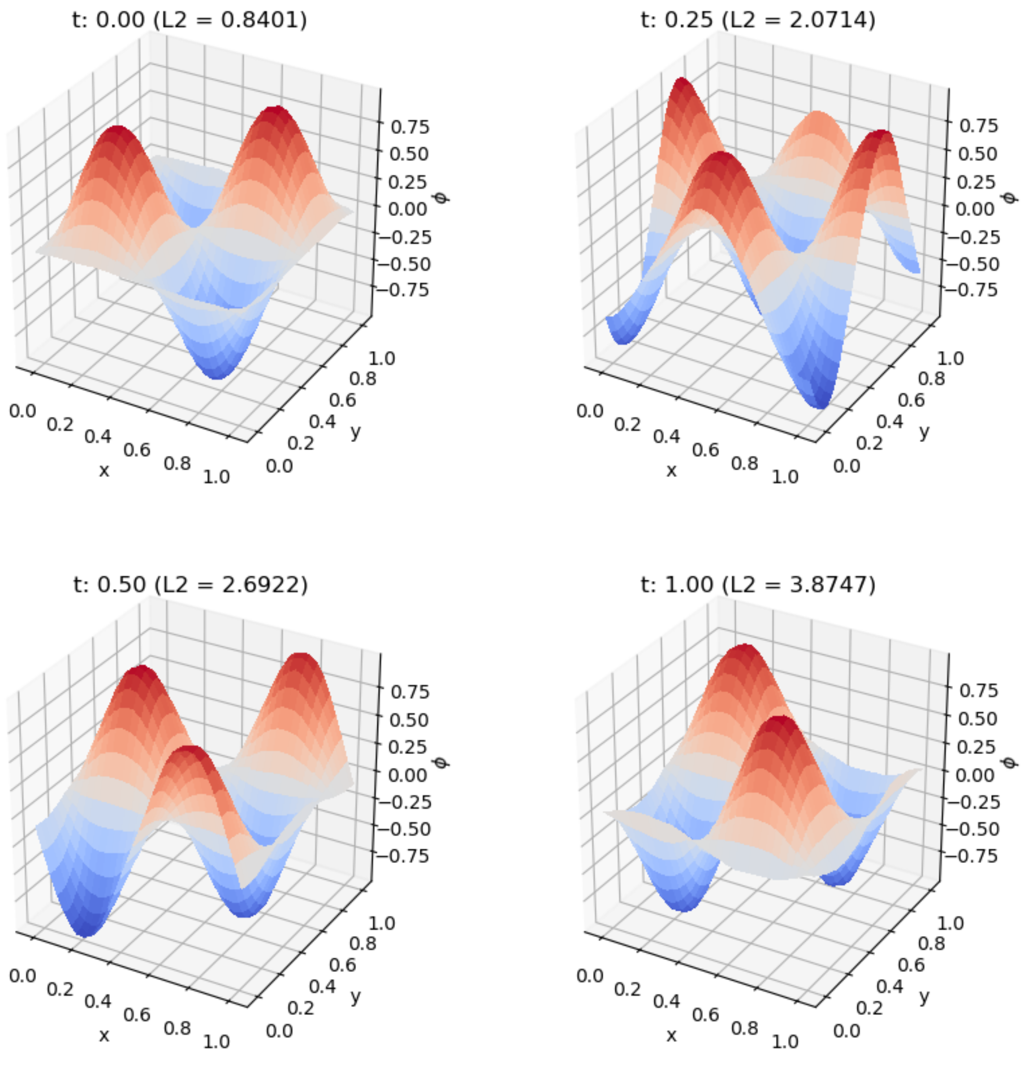


Figure 3.4: Results with a 50x50 mesh.

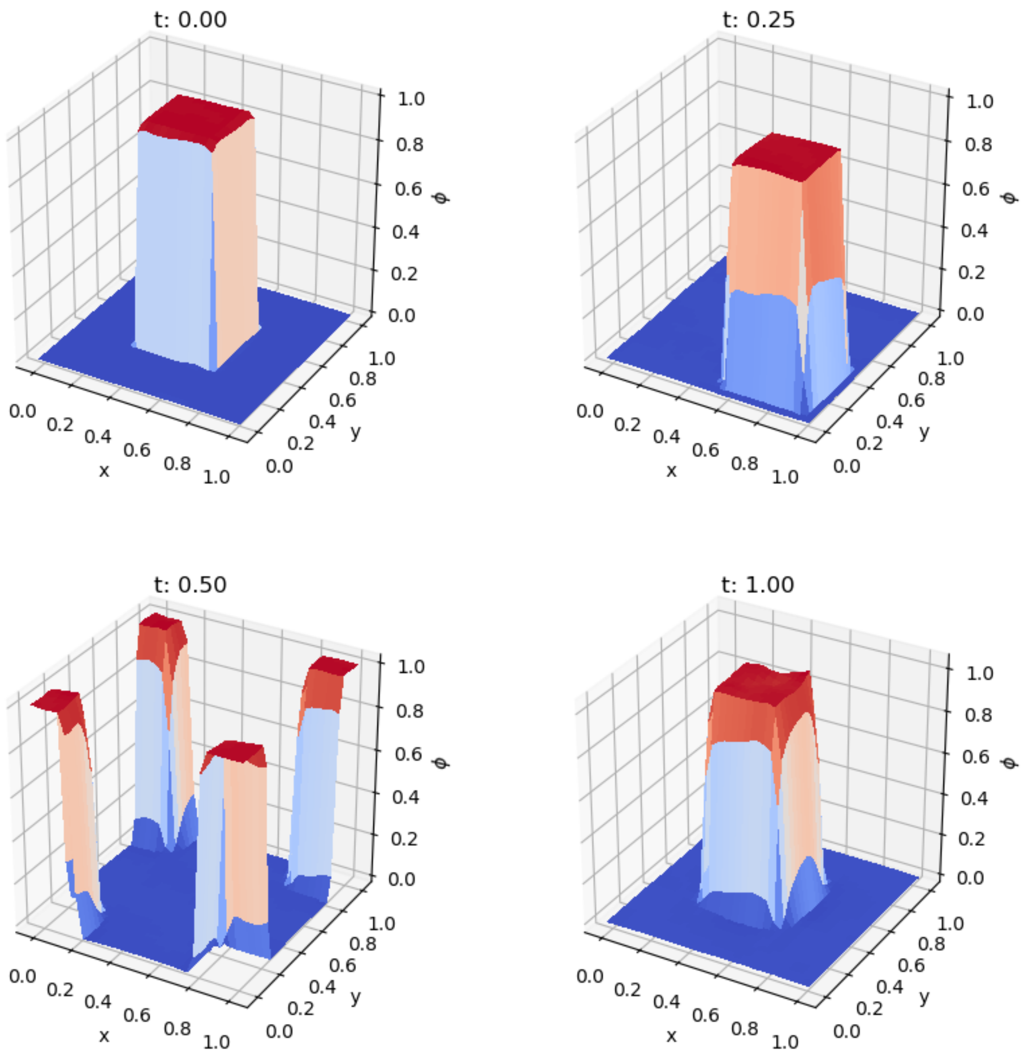


Figure 3.5: Results with a 30x30 mesh.

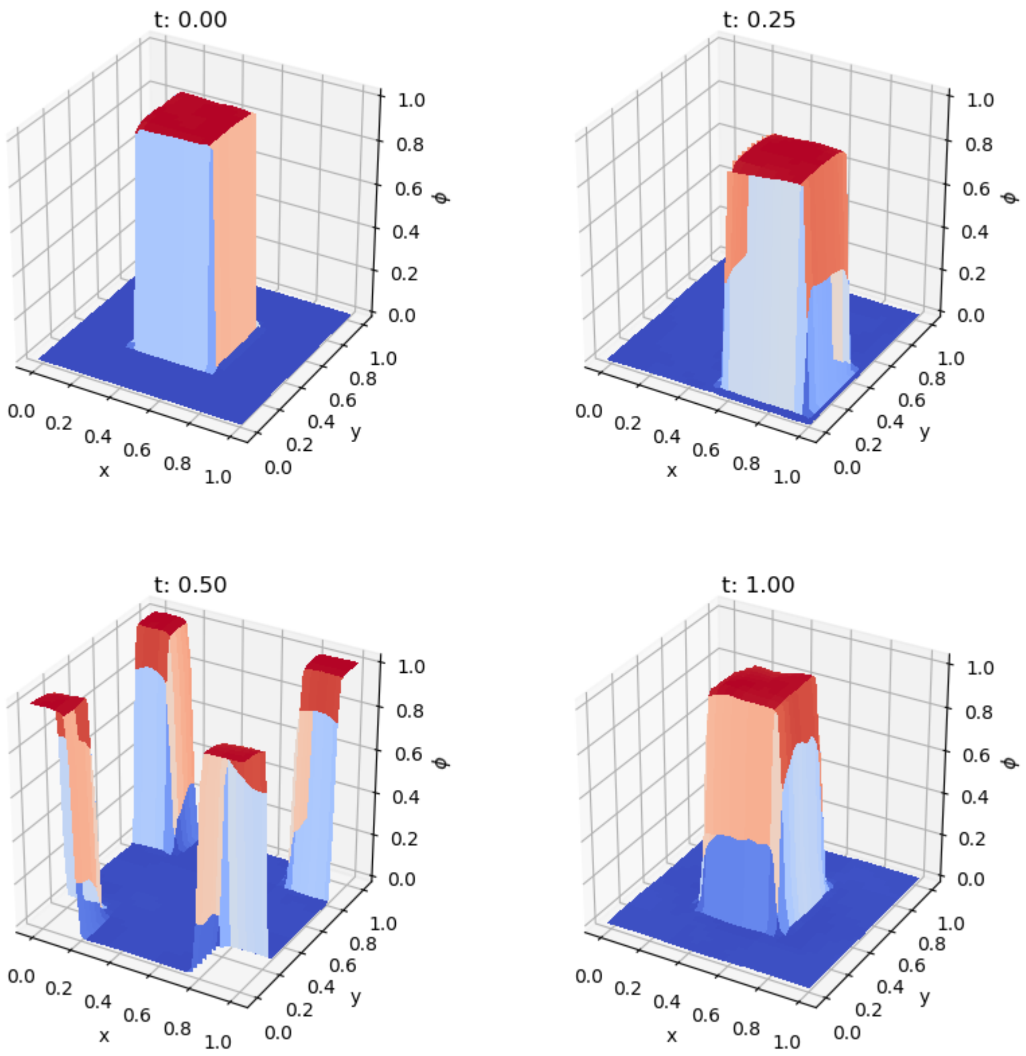


Figure 3.6: Results with a 50x50 mesh.

In figures 3.7 and 3.8 the same results that the one in figure 3.3 are presented but obtained with FVM. All of them are compared against the exact solution in terms of the $L2$ error. A mesh with 30x30 elements has been used and the simulation has been run with a time step of 0.01 during 1 time unit and $(u, v) = (1, -0.5)$. We have used two different numerical schemes for the finite volume solutions: upwind (UDS) and central (CDS) schemes. Without requiring extra tuning, our method is able to obtain a more accurate solution in all the domain. Moreover, we can reuse our NN to obtain results with different meshes while the use of FVM would require new simulations. Also, the solution provided by the NN is derivable in the entire domain, which is not true for the FVM results since they consist of piece-wise functions. The difference between methods, however, is not as big as in the one-dimensional case.

3.3 Conclusions

In this chapter the two-dimensional advection equation has been solved using a NN for two different initial conditions (smooth and discontinuous). After several experiments, a NN with 5 hidden layers with 512 hidden units in each layer was selected to perform the experiments. Three sets of training points have been constructed to include the initial condition, the boundary conditions and the internal solution. Each set has its own loss function to be minimized.

First of all, we can conclude that our approach to solve the PDE is able to obtain results that are correct in the physical way. The initial condition is transported at the correct velocity as it is predicted by the analytical solution. The error, however, depends on the final loss achieved during the training process. The closer to 0 the loss goes, the less error compared to the analytical solution is observed.

A study of the impact of the number of training points considered has been carried out. It is concluded that, for this simple case, the amount of points used in the spatial domain is not very important since all the meshes achieved a good solution whenever the loss was able to reduce to near zero values. This fact strengthen our previous conclusion that the accuracy of the solution ultimately depends on the ability of the network to find the optimal weights. In more complicated configuration this fact can be affected by the domain discretization, but that is not the case for the experiments conducted in this chapter.

Finally, a comparative between the presented method and traditional finite volume methods has been presented. The accuracy of the obtained solutions with the different methods has been compared. The NN is able to provide much more accurate solutions than traditional FVM at lower computational cost (big meshes and big time steps). This is due to the fact

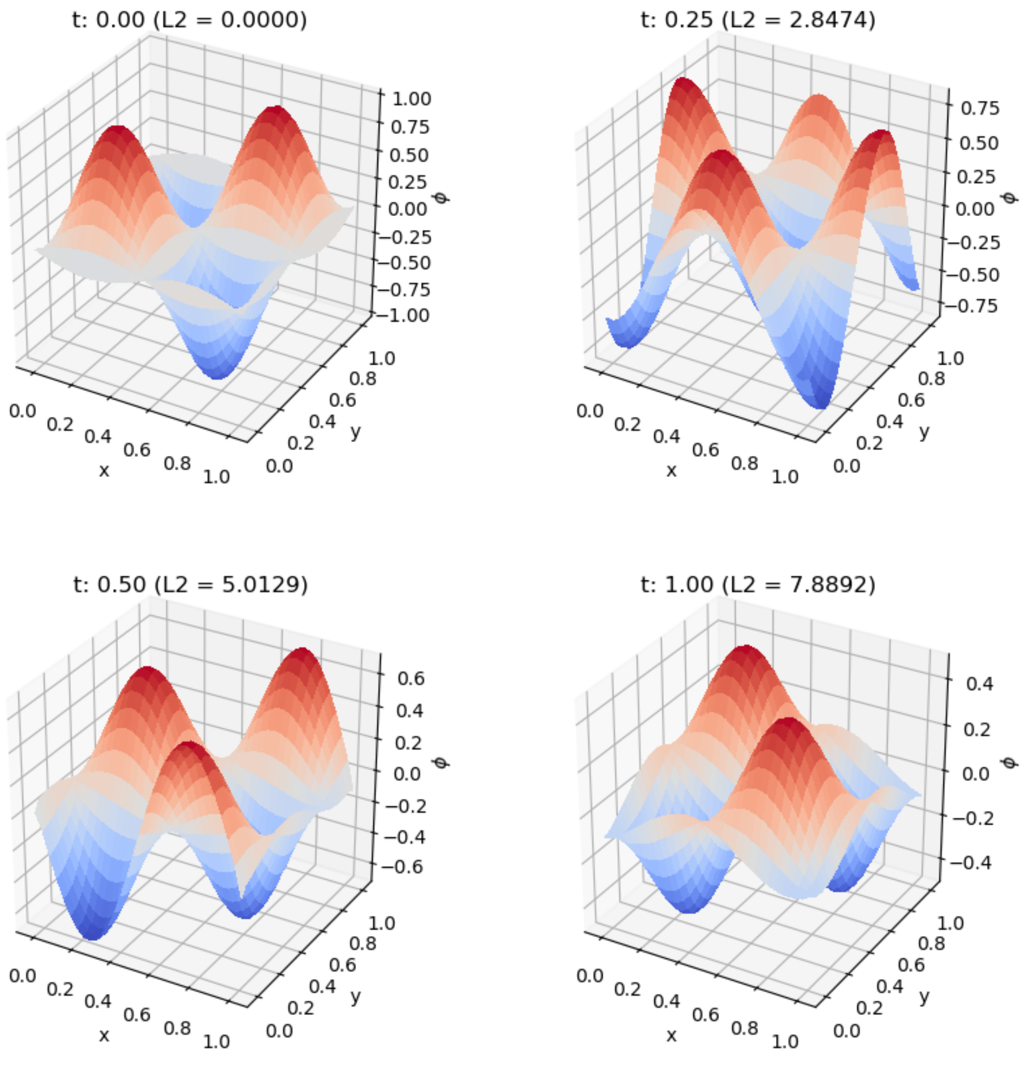


Figure 3.7: Results with UDS.

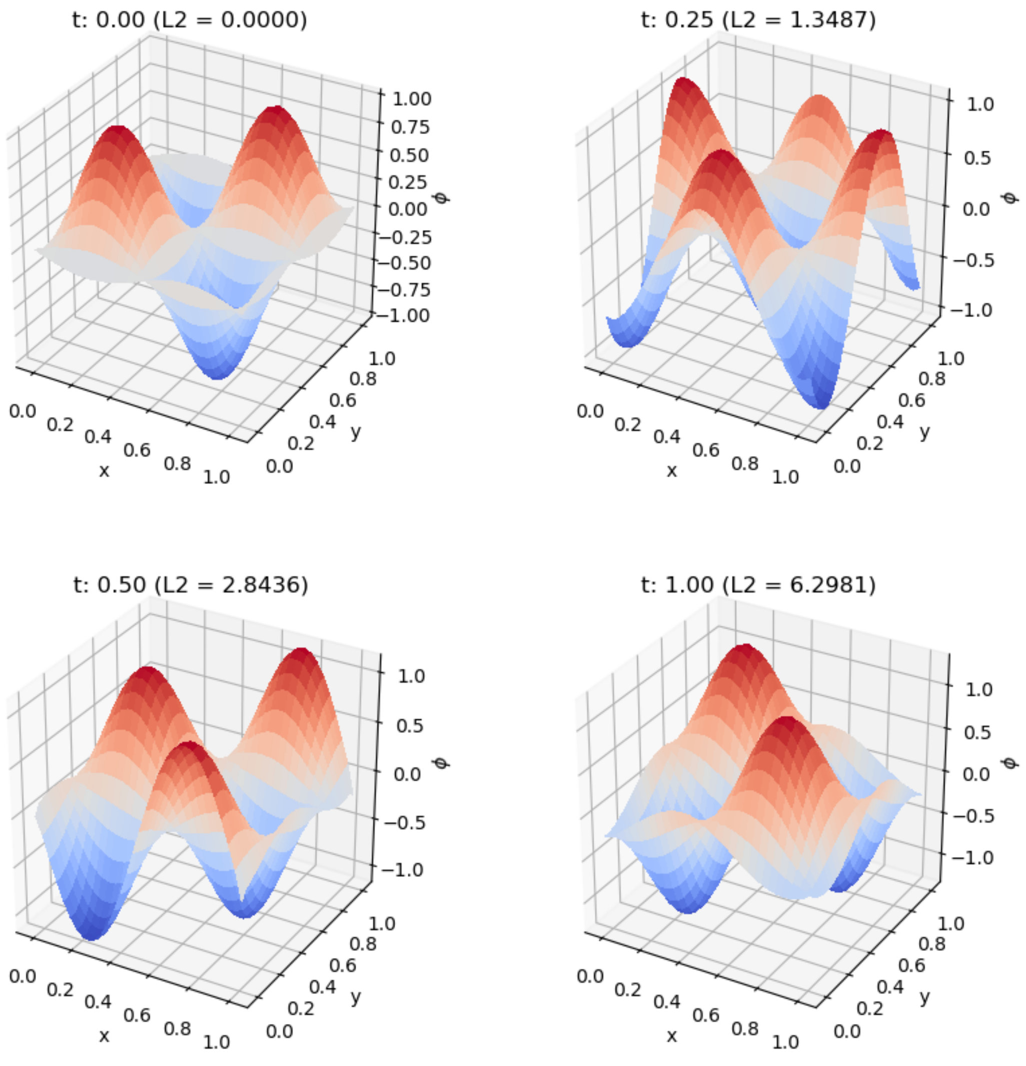


Figure 3.8: Results with CDS.

that FVM are constrained by physical limitations concerning the exchange of information between cells. NNs, on the other hand, are not affected by this constrains giving them the ability to compute more accurate solutions using bigger time steps and less points. The $L2$ error obtained with the NN is approximately the error achieved during the training phase. This means that NN allow to evaluate in real-time the accuracy of the solution while training. Moreover, any technique used to decrease the loss during the training (increase the number of hidden layers or units, for example) directly impacts the accuracy of the solution.

Chapter 4

Inviscid Smith-Hutton Problem

In this chapter we are going to solve the inviscid Smith Hutton problem. As it can be seen in figure 4.1 a rectangular domain of $(x, y) = [-1, 1]\text{x}[0, 1]$ is considered. Now, the velocity field is not constant but a function of x and y. Dirichlet boundary conditions are considered everywhere except for the outlet, where a Neumann boundary condition is specified. The resulting flow field makes the inlet condition to travel through the domain until it reaches the outlet, when a steady state is achieved. In this problem we are interested in the steady solution, hence we are going to solve the following PDE:

$$(u(x, y)\phi)_x + (v(x, y)\phi)_y = 0 \tag{4.1}$$

where the temporal term has been neglected (in a steady flow, $\phi_t = 0$).

4.1 Neural Network

The boundary conditions, and therefore the final solution, in the problem are parametrized as a function of the parameter α. We are going to use this value as an additional input to our network in order to evaluate the generalization properties of the presented method. Out objective is to train the network for several values of α and then evaluate the accuracy of the solution in new values not seen in training. This means that, if we achieve our goal, we could easily study a wide range of values in order to find optimal values where the traditional approach would require to perform a new simulation for each value.

4.1.1 Topology

We are going to use a neural network with 3 input units (x, y, α) and 1 output unit, $\phi(x, y, \alpha)$ as depicted in figure 4.2.

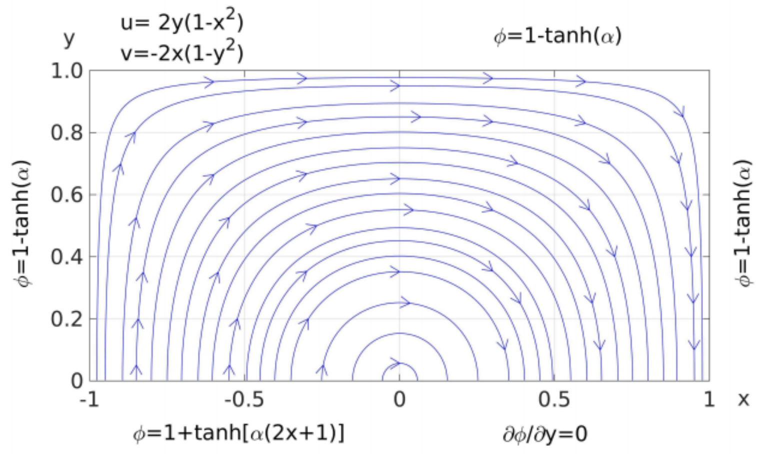


Figure 4.1: Smith-Hutton problem.

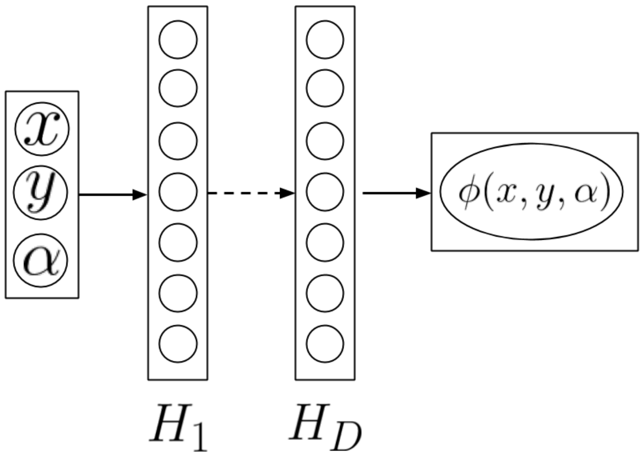


Figure 4.2: NN topology used for solving the inviscid Smith Hutton problem.

4.1.2 Training points

In order to build the training set we first define a set of points in which we are going to evaluate our solution during the training process. We define N_x+2 points in x (N_x interior points and 2 points at the boundaries), N_y+2 points in y and M different values of α. In total, there are $(N_x+2)(N_y+2)M$ points. Then we divide this points in three groups:

- N_xN_yM points to evaluate the PDE in the internal points.
- $(3N_x/2+2N_y)M$ points to evaluate Dirichlet boundary conditions.
- $(N_x/2)M$ points to evaluate Neumann boundary conditions.

Neumann boundary conditions are used at the points with $y=0$ and $x>0$, Dirichlet boundary conditions are used elsewhere.

4.1.3 Loss function

Each set of points has their own loss function to minimize, so the final loss function is composed by three different losses:

- The output of the network for the Dirichlet boundary condition training points is used along with the known value to build a Mean Square Error loss function to be minimized.
- The outputs of the network for the Neumann boundary conditions training points are derived with respect to the inputs in order to obtain ϕ_y. Then, since $\phi_y=0$ for all the points at the boundary we use the derivative as the loss function to be minimized.
- The output of the network for the internal points is also derived with respect to the inputs, obtaining $(u\phi)_x$ and $(v\phi)_y$. A loss function that matches the PDE is built. Since $(u\phi)_x+(v\phi)_y$ has to be 0 for every x and y we simply minimize this loss function.

If the triple loss function reduces to 0 during the training process, we can assume that any result that we obtain with the NN for any arbitrary point (x, y, α) included in the training set satisfies the boundary conditions and the PDE. Thus, we have an approximate solution for our PDE that is continuous and derivable throughout the entire domain. We can obtain the solution to the PDE at any desired point with a simple forward pass through the network.

4.2 Results

Results are obtained for a 60x30 grid and 4 different values of α. A 5 layer NN with 1024 hidden units in each layer is used. The training was carried out with a learning rate of 0.001 through several epochs, and several retraining cycles at lower learning rates in order to achieve better results. In figure 4.3 we can see the steady state flow fields for each value of α considered in the training process. Also, the bottom boundary profiles are presented to evaluate the ability of the NN to reproduce the inlet function at the outlet. The $L2$ error is computed at the outlet profile to asses the accuracy of the method. As it can be seen, the results agree quite well with the expected behavior, although we can observe some degree of error at the higher α values. Results for other values of α not used in the training are presented in figure 4.4 to show the generalization property of the NN. Despite observing a good tendency on the results, they do not match as well as the previous cases (which is obvious since the NN is not trained for these new parameters). We can observe that the profile at the outlet matches well the inlet values which means that the PDE is well resolved. However, the inaccuracy in reproducing the correct inlet value is causing the overall result to be less accurate. If we include these new cases in the training set we can observe how results improve for all the cases (figures 4.5 and 4.6) suggesting that the more cases we take into account during training, the better our NN generalize.

This results are very powerful since we are able to obtain the solution at any α inside the range used in the training process. A simple forward to the NN gives us the resulting field and hence we could very quickly study results at different values of α to study optimal conditions. Approximate values can be obtained with this simple process, and then the points of interest can be used to retrain the model in order to obtain much more accurate results around this zones. The described process could also be automatic using, for example, a genetic algorithm replacing long and possibly erroneous simulations with a simple forward pass through the trained NN.

4.3 Conclusions

In this chapter the two-dimensional advection equation has been solved using a NN in the inviscid Smith Hutton problem. The problem uses a semi-circular velocity field to propagate a profile defined at the inlet boundary condition and through the domain until it reaches the outlet. If the problem is solved correctly, the profile observed at the outlet should match exactly the one in the inlet. The problem is parametrized with a constant α. We have included this parameter as an input to our NN in order to obtain the solution at any condition within a considered range. After several experi-

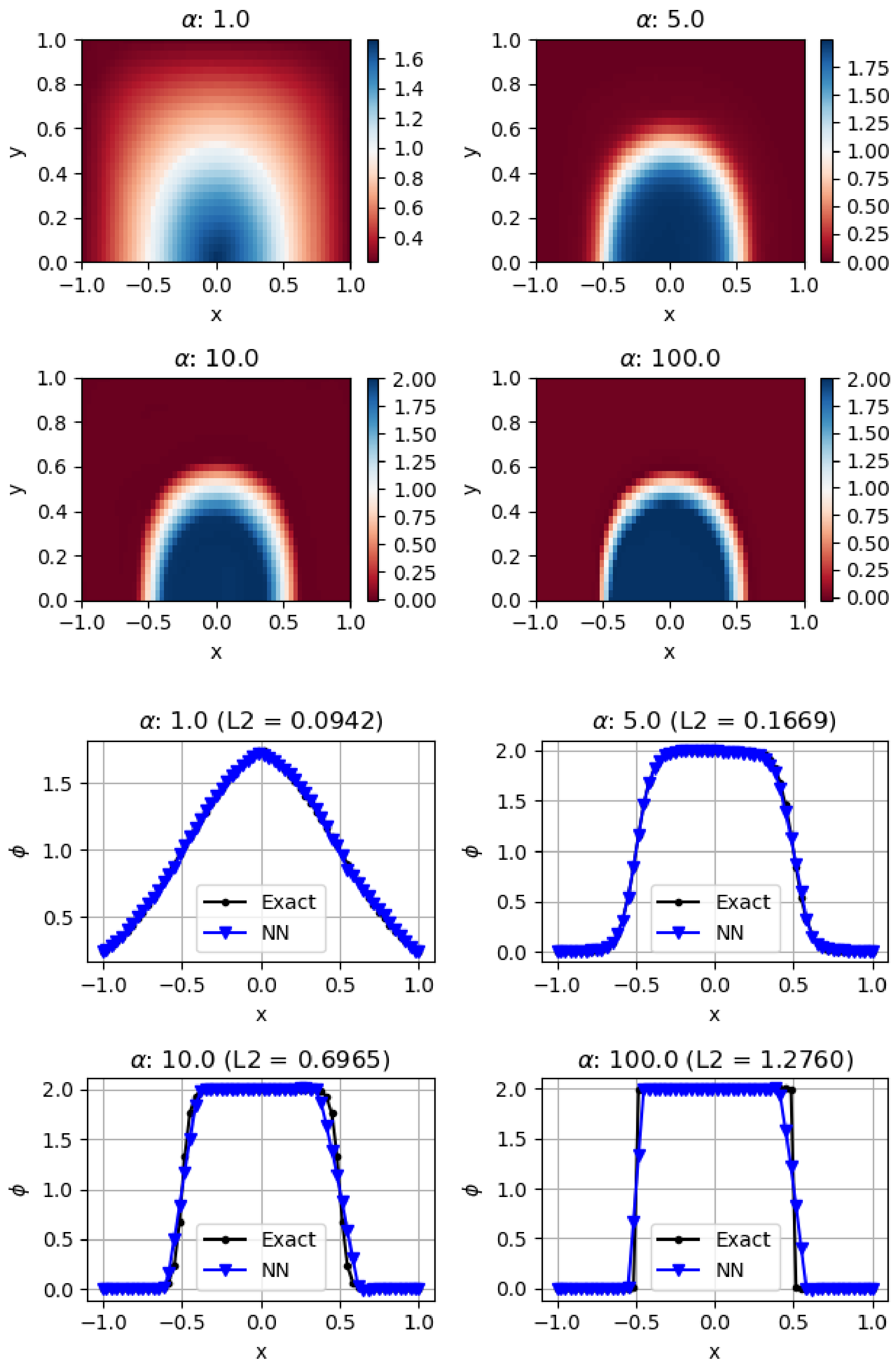


Figure 4.3: Smith Hutton results. Steady state fields (top) and bottom boundary profiles (bottom).

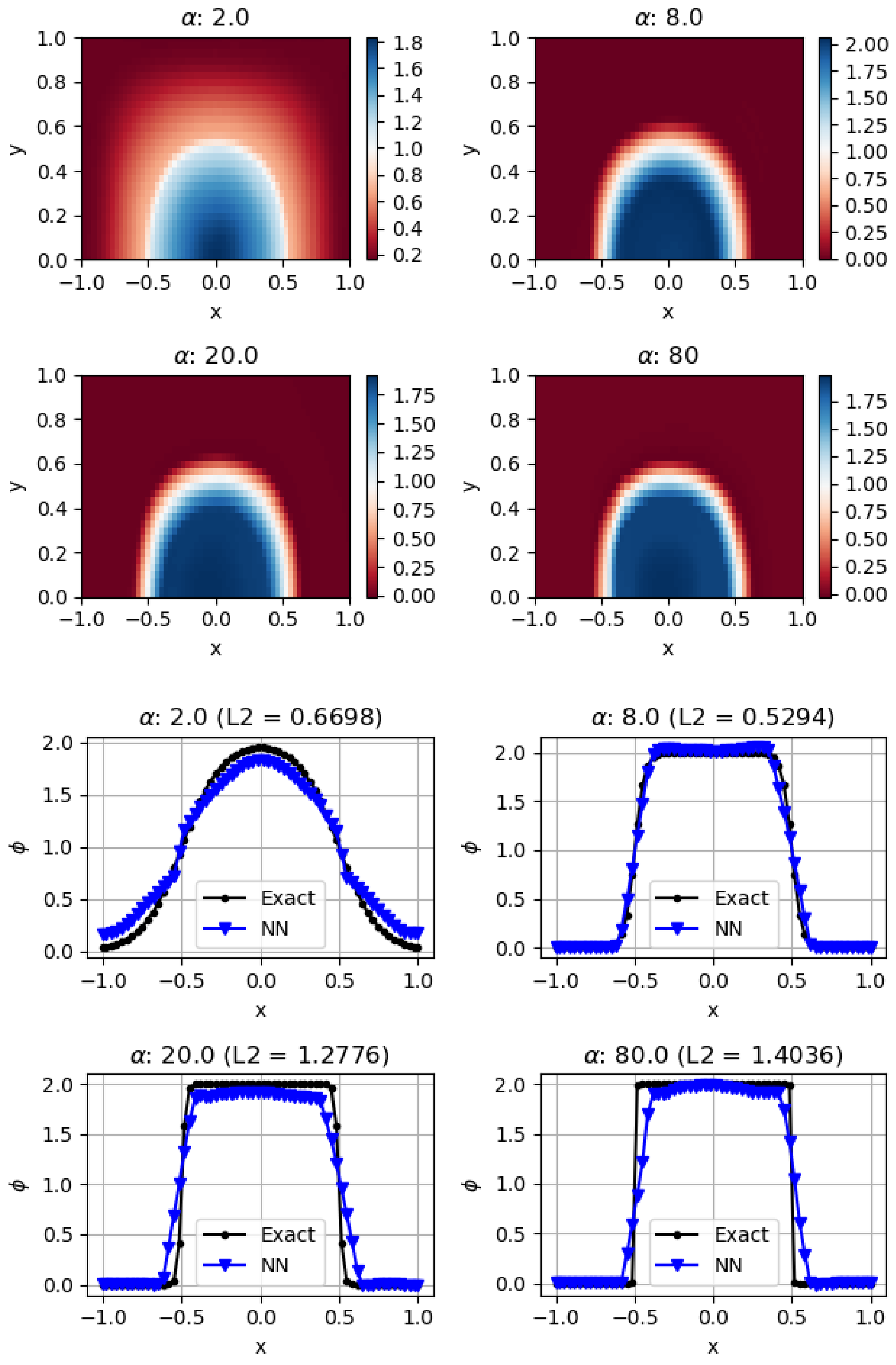


Figure 4.4: Smith Hutton results. Steady state fields (top) and bottom boundary profiles (bottom).

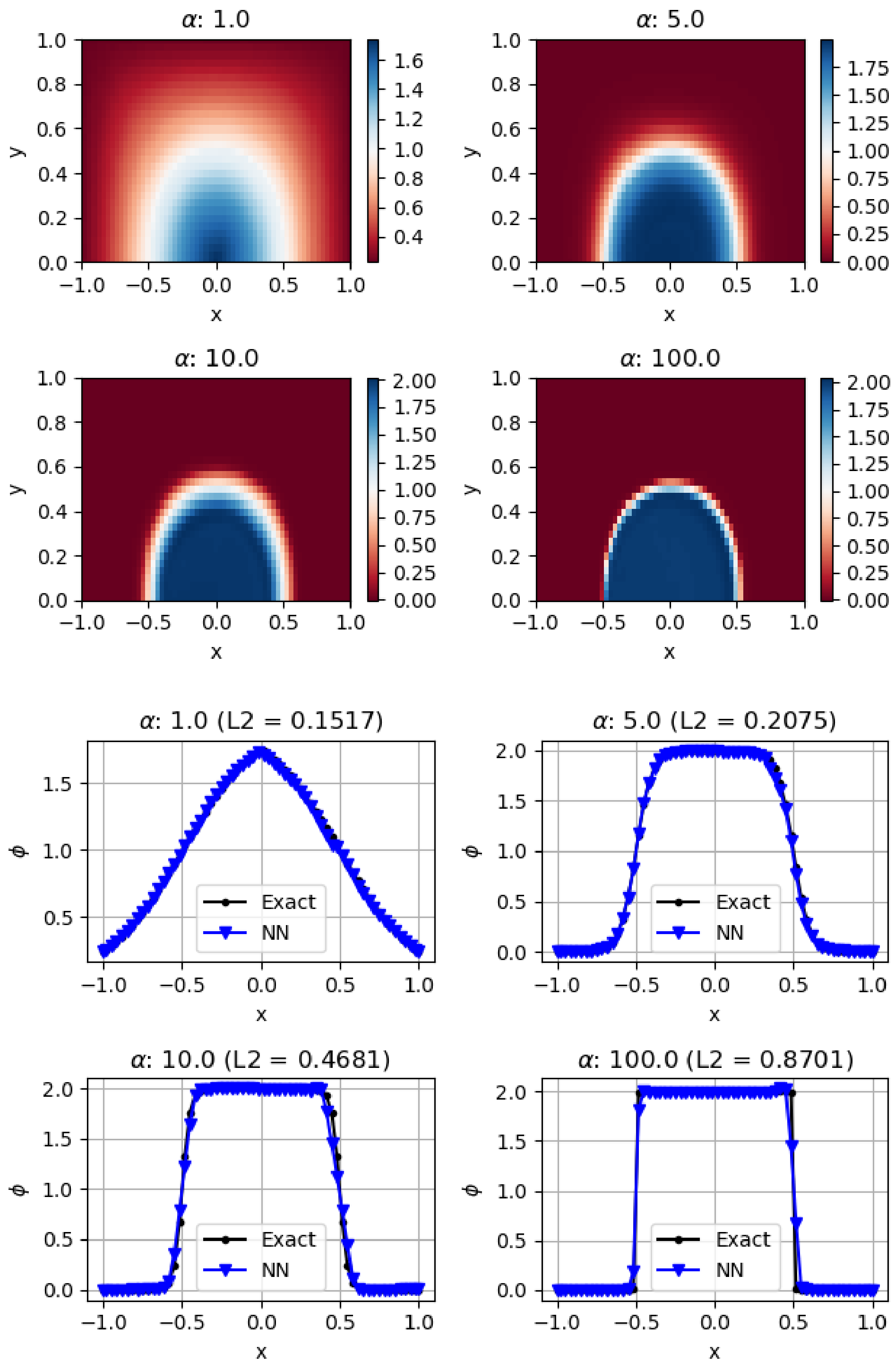


Figure 4.5: Smith Hutton results. Steady state fields (top) and bottom boundary profiles (bottom).

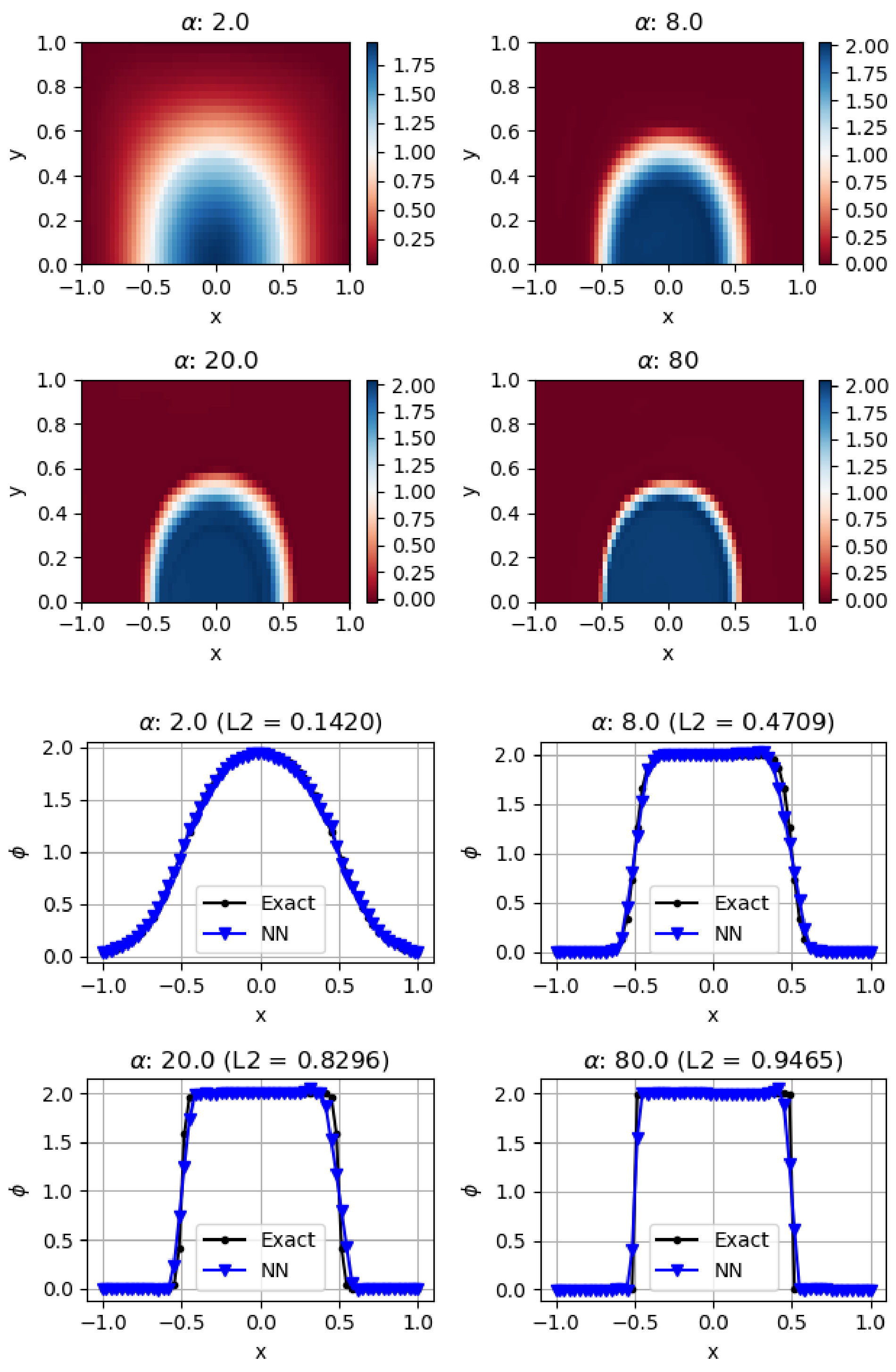


Figure 4.6: Smith Hutton results. Steady state fields (top) and bottom boundary profiles (bottom).

ments, a NN with 5 hidden layers with 1024 hidden units in each layer was selected to perform the experiments. Three sets of training points have been constructed to include the Dirichlet and Neumann boundary conditions and the internal solution. Each set has its own loss function to be minimized.

We have obtained good results when training the NN with 4 different values of α. Once the NN is trained we have obtained results in a different set of α's. Despite obtaining moderate good results, the accuracy of the unseen cases was not as good as the seen cases. The source of error has been identified in the inaccuracy on the reproduction of the inlet profile. The solution at the outlet matched the inlet condition, despite being erroneous.

Then, we decided to perform a second experiment including all the considered values of α. In this case, the overall accuracy was improved for all the cases. This fact suggests that in order to obtain better results, a wide range of configurations has to be considered. At the end, a trade off between training accuracy and computational cost would determine the amount of training points to be used.

To conclude with this chapter we have proposed an approach to apply the presented method for fast optimization studies. Depending on the number of different conditions to study, training a NN can be faster than numerically solving each one of them. Furthermore, once the network is trained we can use it to obtain accurate solutions for other conditions not seen during the training process. This approach can dramatically speed up the optimization process of a particular problem. Finally, we can retrain the same network in the particular optimal areas to improve the accuracy.

Chapter 5

Viscous Smith-Hutton Problem

In this chapter we are going to solve the viscous Smith Hutton problem. The problem configuration is exactly the same than the one presented in the previous chapter (see figure 4.1), but now we are going to introduce diffusive effects solving the two-dimensional advection-diffusion PDE:

$$(u(x,y)\phi)_x + (v(x,y)\phi)_y = \Gamma(\phi_{xx} + \phi_{yy}) \tag{5.1}$$

where ϕ_{xx} and ϕ_{yy} are the second-order derivatives of ϕ with respect to x and y, respectively, and Γ is the diffusive constant. In contrast to all the cases presented in the previous chapters, we cannot obtain an analytical solution to our PDE (except when $\Gamma \to 0$). For small values of Γ we would expect to observe a similar behavior than the inviscid Smith Hutton problem. For big Γ values, however, the inlet profile will be diffused over the domain. This phenomena results in a smoothing of the outlet profile when compared to its inviscid solution.

5.1 Neural Network

The first aspect worth mentioning is the fact that we are going the need the second-order derivative of the network output with respect to the inputs. As mentioned before, during the derivation process the second-order derivative of the activation function is required. Up until now we have used Relu activation functions in all of our experiments. Nevertheless, due to its definition, the second-order derivative of Relu is always 0. This means that if we compute the second-order derivatives of the NN output with respect to the inputs using Relu activation functions we will always obtain 0 values. To avoid this problem and be able to solve second-order PDEs we have to use alternative activation functions that have defined and not null second-order

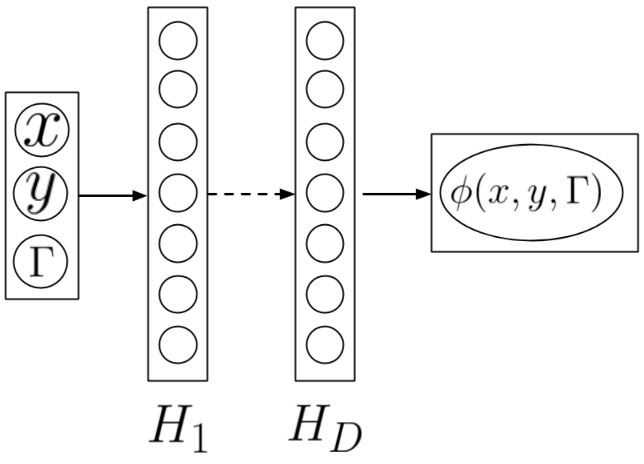


Figure 5.1: NN topology used for solving the viscous Smith Hutton problem.

derivatives. In the present work, we are going to use Sigmoid activation functions to achieve that goal.

In the previous chapter we observed how we can include the free-parameters of the problem as inputs to the network in order to solve our PDE in a wide range of conditions and not only in a particular case, with the benefits that it implies. In this chapter the diffusive constant will be used as an additional input. This means that, if we solve the problem, our NN will be able to provide the solution to the PDE at any value of Γ. In a traditional numerical approach, for each value a complete new simulation would be required.

5.1.1 Topology

We are going to use a neural network with 3 input units (x, y, Γ) and 1 output unit, $\phi(x, y, \Gamma)$ as depicted in figure 5.1.

5.1.2 Training points

In order to build the training set we first define a set of points in which we are going to evaluate our solution during the training process. We define N_x+2 points in x (N_x interior points and 2 points at the boundaries), N_y+2 points in y and M different values of Γ. In total, there are $(N_x+2)(N_y+2)M$ points. Then we divide this points in three groups:

- N_xN_yM points to evaluate the PDE in the internal points.
- $(3N_x/2 + 2N_y)M$ points to evaluate Dirichlet boundary conditions.

- $(N_x/2)M$ points to evaluate Neumann boundary conditions.

Neumann boundary conditions are used at the points with $y = 0$ and $x > 0$, Dirichlet boundary conditions are used elsewhere.

5.1.3 Loss function

Each set of points has their own loss function to minimize, so the final loss function is composed by three different losses:

- The output of the network for the Dirichlet boundary condition training points is used along with the known value to build a Mean Square Error loss function to be minimized.
- The outputs of the network for the Neumann boundary conditions training points are derived with respect to the inputs in order to obtain ϕ_y. Then, since $\phi_y = 0$ for all the points at the boundary we use the derivative as the loss function to be minimized.
- The output of the network for the internal points is also derived with respect to the inputs, obtaining $(u\phi)_x$, $(v\phi)_y$, ϕ_x and ϕ_y. Then, a second derivation is performed to obtain ϕ_{xx} and ϕ_{yy}. A loss function that matches the PDE is built. Since $(u\phi)_x + (v\phi)_y - \Gamma(\phi_{xx} + \phi_{yy})$ has to be 0 for every x and y we simply minimize this loss function.

If the triple loss function reduces to 0 during the training process, we can assume that any result that we obtain with the NN for any arbitrary point (x, y, Γ) included in the training set satisfies the boundary conditions and the PDE. Thus, we have an approximate solution for our PDE that is continuous and derivable throughout the entire domain. We can obtain the solution to the PDE at any desired point with a simple forward pass through the network.

5.2 Results

Results are obtained for a 60x30 grid and 3 different values of Γ. A 4 layer NN with 1024 hidden units in each layer and Sigmoid activation functions is used. The training was carried out with a learning rate of 0.001 through several epochs, and several retraining cycles at lower learning rates in order to achieve better results. In figure 5.2 we can see the steady state flow fields for each value of Γ considered in the training process. Also, the bottom boundary profiles are presented and compared with numerical results available in the literature. As it can be seen, the results agree quite well with the expected behavior.

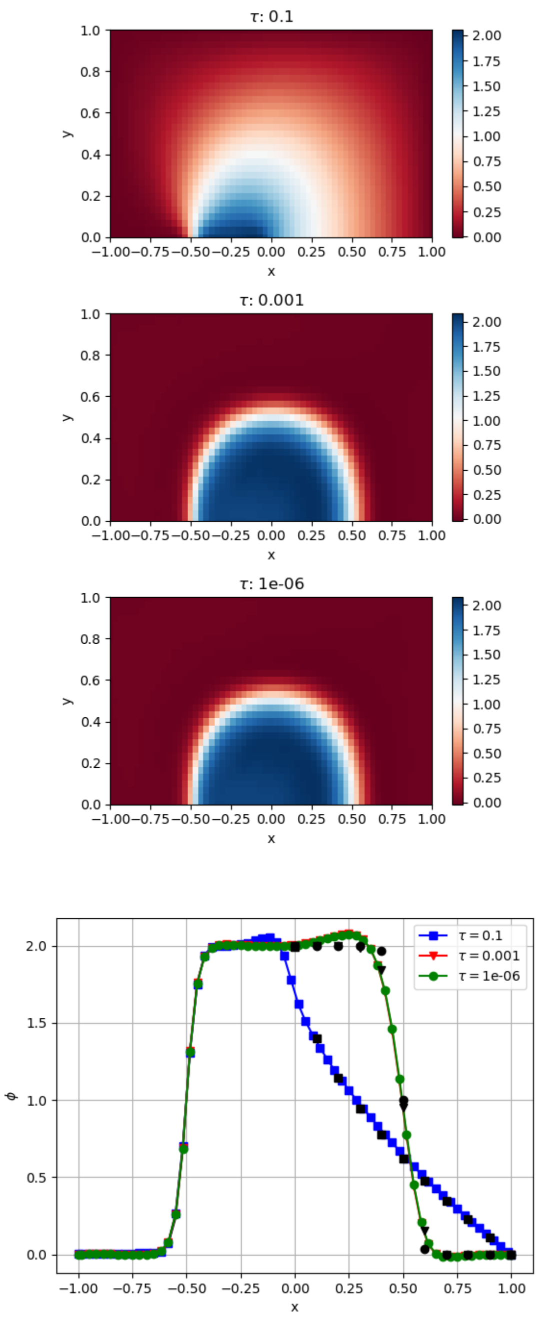


Figure 5.2: Smith Hutton results. Steady state fields (top) and bottom boundary profiles (bottom).

5.3 Conclusions

In this chapter the two-dimensional advection-diffusion equation has been solved using a NN in the viscous Smith Hutton problem. The problem uses a semi-circular velocity field to propagate a profile defined at the inlet boundary condition and through the domain until it reaches the outlet. In contrast to the inviscid Smith Hutton problem we cannot find an analytical solution for this problem except when the diffusive constant, Γ, is close to 0. We have included this constant as an input to our NN in order to obtain the solution at any condition within a considered range. After several experiments, a NN with 4 hidden layers with 1024 hidden units in each layer and Sigmoid activation functions was selected to perform the experiments. Relu activation functions are not useful for second-order PDE since their second-order derivative is always 0. Three sets of training points have been constructed to include the Dirichlet and Neumann boundary conditions and the internal solution. Each set has its own loss function to be minimized.

We have obtained good results when training the NN with 3 different values of Γ. They match quite well with reference results found in the literature.

We could also include α like in the previous chapter to obtain an even more general solution to our PDE allowing the study of a very large range of cases.

Chapter 6

Conclusions & Further Work

6.1 Conclusions

...

6.2 Further Work

...