Delft University of Technology
Faculty Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Investigating Physics-Informed Neural Networks for Solving PDEs

(Dutch: Physics-Informed Neurale Netwerken voor het Oplossen van PDVs)

In partial fullfilment of the requirements
for the degree of

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

by

ELHAM WASEI

Delft, The Netherlands
August 2020

BSc report APPLIED MATHEMATICS

Investigating Physics-Informed Neural Networks for Solving PDEs
(Dutch: Physics-Informed Neurale Netwerken voor het Oplossen van PDVs)

ELHAM WASEI

Delft University of Technology

Supervisor

Dr. ir. D. Toshniwal

Committee Members

Prof. dr. ir. A.W. Heemink

Dr.ir. D. Toshniwal

August, 2020 Delft

Abstract

Physics-Informed Neural Networks (PINNs) are a new class of numerical methods for solving partial differential equations (PDEs) that have been very promising. In this paper, four different implementations will be tested and compared. These include: the original PINN functional with equal weights for the interior and boundary loss, the same functional with custom weights, and the First Order Least Squares (FOSLS) functional with equal weights and custom weights. These custom weights are chosen to be equal to the optimal weights derived by Oosterlee et al. as well as slightly bigger and smaller. These methods will be applied to the 1D stationary advection-diffusion equation where we vary the difficulty by configuring the diffusion parameter ε. Furthermore, for each method we have done an elaborate parameter study where we varied ε and the number of collocation points. We have found that the weights derived by Oosterlee et al. did not provide accurate results in our specific setting. Instead, equal weights usually performed best. Also, the two functionals turned out to have very similar performance. For the Python code visit **Github**.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Conventional numerical methods are old tools in numerical mathematics. The finite difference method was first used by Leonhard Euler in the 18th century, while the finite element method has been made robust by several decades of research. Still, these numerical methods have their shortcomings, which are exemplified by difficult problems in computational science and engineering. One such problem is the convection dominated convection-diffusion equation (or advection-diffusion). In this chapter we will introduce this equation, give possible pitfalls for conventional numerical methods, and explore reasons why neural networks may be a better solver. In chapter 2, we will explore the basic mechanisms of artificial neural networks. In chapter 3, we explore the implementation of neural networks to PDE's and in chapter 4 we specifically apply it to convection-diffusion. We will explore the results in chapter 5.

1.1 Problem formulation

The advection-diffusion equation describes a wide range of physical phenomena, in which particles, heat, energy or some other physical quantity is moving in space due to two processes: advection and diffusion. Advection refers to the bulk motion of the physical quantity, for example, the transport of pollutants in a river by the flow of the water. Diffusion on the other hand refers to the movement of a physical quantity where its concentration is high to where its concentration is low, which happens because of (in some sense) random movement of molecules.

The general 1D advection-diffusion equation (without sources and sinks) is given by

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\varepsilon \frac{\partial u}{\partial x}\right) - v\frac{\partial u}{\partial x} \tag{1.1}$$

where ε and can depend on x but will not do so in this paper. It is assumed that $\varepsilon > 0$. We want to numerically approximate the stationary solution to this equation on the domain [0,1] with boundary conditions, that is we want to solve

the boundary value problem

$$\begin{aligned} -\varepsilon\frac{d^2u}{dx^2} + v\frac{du}{dx} &= 0 \quad \text{for} \quad x \in (0,1) \\ u(0) = 0, \quad u(1) &= 1 \end{aligned} \tag{1.2}$$

For this problem it is not hard to find the exact solution. One can verify that it is given by

$$u(x) = \frac{1 - e^{vx/\varepsilon}}{1 - e^{v/\varepsilon}} \tag{1.3}$$

Using this exact solution, we can test any numerical approximation by comparing its graph to the graph of (1.3). This benchmark problem can give us insight into how our numerical methods will perform on problems where the exact solution can not be derived analytically. Note that (1.3) is a monotonically increasing function. Any oscillations in a numerical solution are therefore unwanted. Oscillatory solutions also violate an important theorem called the Maximum Principle, which states that local minima in the interior cannot exist. We will see that the finite difference method can encounter this problem.

1.2 Finite Differences

In the Finite Difference Method (FDM) the domain of computation is split using a mesh grid $\{x_0, x_1, ..., x_{N+1}\}$, which can be an equidistant set of points. Due to the Dirichlet boundary conditions, the boundary points will be included in this set, i.e. $x_0 = 0$ and $x_{N+1} = 1$.

We will need to numerically approximate the first and second derivatives of u. This can be done with central difference formulas, which have the nice property that the order of the truncation error is quadratic. They are given by

$$\begin{aligned} u'(x_k) &= \frac{u(x_{k+1}) - u(x_k)}{2\Delta x} + \mathcal{O}(\Delta x^2) \\ u''(x_k) &= \frac{u(x_{k-1}) - 2u(x_k) + u(x_{k+1})}{\Delta x^2} + \mathcal{O}(\Delta x^2). \end{aligned} \tag{1.4}$$

Substituting these expressions into 1.2 and neglecting the error we obtain a system of N equations, as the solution on the boundary points are known. This substitution results in the difference equation

$$-\varepsilon\frac{w_{k-1} - 2w_k + w_{k+1}}{\Delta x^2} + v\frac{w_{k+1} - w_{k-1}}{2\Delta x} = 0 \tag{1.5}$$

where $w_k \approx u(x_k)$ is the numerical approximation. To see why this is problematic, we can try to solve this difference equation exactly. Assume the solution on the interior is of the form $w_k = r^k = r^{k-1}r$. In this case $w_{k-1} = r^{k-1}$ and $w_{k+1} = r^{k-1}r^2$. Substituting these into (1.5), dividing by r^{k-1} (and noting that

$r = 0$ a trivial solution) and doing some algebraic manipulations (where again we can find $r = 1$ to be a trivial solution), we arrive at [9]

$$r = \frac{1 + \frac{v\Delta x}{2\varepsilon}}{1 - \frac{v\Delta x}{2\varepsilon}} \tag{1.6}$$

with $\frac{v\Delta x}{2\varepsilon}$ called the mesh Péclet number Pe. To satisfy the boundaries we can take $w_k = A + Br^k$ with appropriate values for A and B. Oscillatory solutions occur when $r < 0$, i.e. $Pe > 1$. Thus we require that $Pe < 1$. Here we can see that if ε is small, then the number of gridpoints needed can become very large, even computationally prohibitive, especially in higher dimensions.

In conventional numerical analysis, there are several remedies to this problem. These include: using a backward difference scheme for the first derivative (called *upwinding*, and assuming that $v > 0$) and refining the mesh where the solution varies quickly (called the *boundary layer*). However, these remedies have several drawbacks, which invites us to explore neural networks. Firstly, these remedies do not generalize well into higher dimensions. Secondly, note that upwind differences adds artificial diffusion to the problem. This can be seen by analyzing the truncation error when approximating the first derivative and observing a term involving u'' which can be effectively added up to the diffusion term. When training a neural network, the network will add perhaps less artificial diffusion than necessary.

Furthermore, if one needs to approximate the solution to the boundary value problem for different values of the parameters ε and v, then every time a large system of equations needs to be solved. In contrast, when these parameters are inputs to the neural network along with the spatial coordinate, then the network needs to be trained only once and the solution can be extracted for as many values of the parameters as one needs. Another advantage of neural networks is that the solution is a continuous (and sometimes differentiable) function, unlike a solution obtained with the FDM, which is only defined at a finite set of points.

CHAPTER 2

Artificial Neural Networks

In this chapter we discuss the main principles and working mechanisms behind Artificial Neural Networks (ANNs). This explanation is based on [5], which we also refer to for a more detailed study. ANNs have been invented a long time ago but fell out of use due to computational infeasibility at the time. Recently they made a resurgence, making use of advancements in CPU and GPU technologies as well as the availability of large datasets in part due to the Internet, which are needed to train the ANNs. ANNs are somewhat inspired by the workings and constituents of the human brain. Unlike traditional computer programs, ANNs are not explicitly programmed to solve specific problems. Instead, they are programmed to seek patterns in existing solutions to similar problems and learn from them, in order to figure out the solution to a new problem.

2.1 Main principles

There are many types of sophisticated ANNs but in this paper, only the *multilayer perceptron* is being discussed, which is a *feedforward* ANN, meaning that nodes are not connected in cycles. From a mathematical point of view, an ANN is nothing more than a function which is the composition of many functions. Usually this is a multivariable vector-valued function. It has many parameters and for a given input, there is some desired output. The goal is to fine-tune these parameters so that the output of this function is close to the desired output, for any input in a given domain of computation. How this function is composed can be best illustrated by a directed graph (Fig. 2.1).

Every node (or neuron) in a layer represents a container that receives a number from the previous layer (this is feedforward) and every edge represents a weight that indicates how much influence the previous node has on the next. There are usually multiple hidden layers and if this is not the case then it is simply called a *perceptron*. As mentioned, the entire network represents a composition of functions, with the first function having input $\{x_1, ..., x_n\}$ which is also the input of the neural network itself, and output $\{a_1^{(1)}, ..., a_m^{(1)}\}$, the values of which

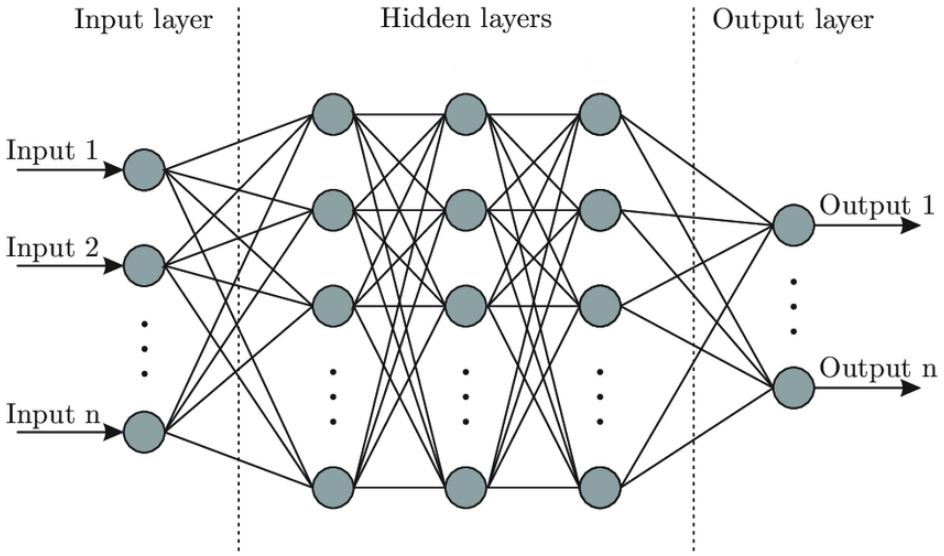

Figure 2.1: Illustration of a multilayer perceptron

are called *activations*. This output is the first hidden layer. The second function has the first hidden layer as its input and has as its output the second hidden layer, etcetera, until the last function outputs the output layer. The superscript denotes in which layer of the network the node is (in this case the first hidden layer) and the subscript denotes the node in that particular layer, with m the number of nodes in the first hidden layer. The first function's output, given the input, are determined as follows.

Let us focus on one node in particular, say $a_j^{(1)}$. What flows into this neuron is a weighted sum of all neurons in the input layer plus some *bias* $b_j^{(1)}$(a scalar), i.e. $\sum_{i=1}^{n} W_{ji}^{(1)} x_i + b_j^{(1)}$ with $W_{ji}^{(1)}$ the weight of the connection from x_i to $a_j^{(1)}$. Next, some *activation function* is applied to this sum to obtain the value of the activation. There are various possibilities, for example the hyperbolic tangent or the sigmoid function defined by

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad x \in (-\infty, \infty).$$

This is a function that is monotonically increasing from 0 to 1. We finally obtain that $a_j^{(1)} = \sigma(\sum_{i=1}^{n} W_{ji}^{(1)} x_i + b_j^{(1)})$. Intuitively, the bias of a neuron indicates how easy it for that neuron to get a high activation. For an arbitrary layer, the activation of the j-th neuron is given by

$$a_j^{(\ell)} = \sigma\left(\sum_{i=1}^{n_{\ell-1}} W_{ji}^{(\ell)} a_i^{(\ell-1)} + b_j^{(\ell)}\right) \tag{2.1}$$

with the zeroth layer corresponding to the input layer, the output layer denoted by L, and $n_{\ell-1}$ denoting the number of neurons in the layer before the ℓ-th layer.

The notation becomes easier when all the neurons in some layer are represented by a column vector and all the weights from one layer to the next represented by a matrix. In this case the activations $\mathbf{a}^{(\ell)}$ of all neurons in layer l can be denoted by

$$\mathbf{a}^{(\ell)} = \sigma\left(W^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}\right). \tag{2.2}$$

2.2 Training the network

We have now presented the mathematical formulation for how particular values for the input neurons flow through the network to create values for the output neurons. Next, we would like to finetune the parameters of this function so that the output is close to the desired output for a given input. This is done using a so called *training set*, which is a set of inputs T for which the desired output is available. Using this training set we can measure how well the network is performing by defining a *loss function*. In many applications the following loss function is used:

$$L(\theta) = \frac{1}{2|T|}\sum_{\mathbf{x}\in T} ||\mathbf{a}^{(\ell)} - \mathbf{y}(\mathbf{x})||^2 \tag{2.3}$$

which is a mean square error. Here θ is the set of all parameters in the network, which is the set of all weights along with the set of all biases and $\mathbf{y}$ is the desired output given some input $\mathbf{x}$. The parameters are initialized in some random manner. Let Θ be the parameter space, which is in fact $\mathbb{R}^{|\theta|}$. The problem at hand is simply to find the values for θ such that the loss function is minimal, that is, to approximate $\arg\min_{\theta\in\Theta} L(\theta)$. This is no easy task due to the high dimensionality of the problem and due to the possibility of L having numerous local minima. However, for sufficiently large neural networks, this does not seem to be a problem, as most local minima are roughly the same quality as the global minimum [3].

One way to approximate this minimum is called *gradient descent*. The gradient

$$\nabla L = \left(\frac{\partial L}{\partial \theta_1}, ..., \frac{\partial L}{\partial \theta_n}\right)^T \tag{2.4}$$

is the direction in which the loss grows fastest. Conversely, $-\nabla L$ is the direction in which the loss diminishes most. A gradient descent iteration is simply moving the solution in this direction as follows. First, pick an initial guess θ_0, which is a vector containing all parameters in the ANN. The update formula is given by

$$\theta_{t+1} = \theta_t - \gamma_t \nabla L(\theta_t) \tag{2.5}$$

where γ_t is some step size that can also depend on the iteration step. Note that this algorithm tries to find local minima of a function instead of global minima. This is not a problem, as previously mentioned.

Gradient descent is a very elementary optimization algorithm that is not used in this paper. Instead, more sophisticated methods are used that are based on gradient descent, in particular *L-BFGS-B*. These are not further elaborated upon. However, these methods still need the gradient of the loss function. To compute this gradient, an advanced technique is needed called *backpropagation*, which is beyond the scope of this text.

CHAPTER 3

Physics Informed Neural Networks

3.1 General Framework

The idea of solving PDE's using artificial intelligence techniques is quite new. For most applications of neural networks, one needs a large training set to achieve an acceptable level of accuracy, because this is needed for the optimization algorithm to converge to the minimum of the loss function. The challenge is to make neural networks effective for solving PDE's while there is not much data. In previous efforts of solving PDE's using neural networks, the PDE itself was not being used when the neural networks learns the solution, which means that a lot of information about the problem is being neglected. Now the key insight is to embed the underlying physics (i.e. the PDE) into the design of the neural network using the loss function. In this way, physically infeasible solutions would be penalized and the neural network will be trained to prevent the output of such solutions. This gives rise to the Physics Informed Neural Networks, or PINNs.

Earlier attempts[6], where Gaussian Process Regression was being used, had certain limitations. Non-linear terms in time had to be linearized and non-linear PDE's were in general hard to solve accurately. In contrast, Raissi et. al have proposed a very general method[7] that can be used to approximate the solution to virtually any PDE. To explain their methods, let us introduce the following. Let a general PDE be given by

$$\begin{aligned} &\frac{\partial u}{\partial t} + \mathcal{N}(u) = 0, \quad \mathbf{x} \in \Omega, \quad t \in [0, T] \\ &\mathcal{B}(t, \mathbf{x}, u) = 0 \end{aligned} \tag{3.1}$$

with $\mathcal{N}$ a differential operator (potentially nonlinear), $u = u(t, \mathbf{x})$ a scalar function of $d + 1$ variables, Ω some subset of $\mathbb{R}^d$ and $\mathcal{B}$ denoting boundary conditions as well as initial conditions. Raissi et al. only considered Dirichlet boundary conditions, but we can generalize this.

To deal numerically with the time derivative, two techniques have been proposed, the *continuous time models* and the *discrete time models*. Note that this

distinction is not relevant for our research since the problem we are investigating has no time derivative at all. However, the methods we will experiment with can be most easily generalized to continuous time models, if one wishes to investigate problems with time derivatives. The difference between these two techniques is the following. In the discrete time models, the time derivative is discretized using a Runge-Kutta scheme, whereas in the continuous time models, no discretization is being used. Instead, the neural network handles the spatial derivates as well as the time derivatives. We will not further elaborate on the discrete time models.

To employ a neural network to solve (3.1), construct it such that the input is $(t, \mathbf{x})$ and the output is $u(t, \mathbf{x})$. That is, if $\mathbf{x} \in \mathbb{R}^d$ then the neural network has d inputs and one output. This network can be used to query the value of u given some point $\mathbf{x}$ in the domain. Furthermore, define $f(t, x)$ as the right hand side of 3.1, that is, $f(t, x) = u_t + \mathcal{N}(u)$. Remarkably, f can be derived from the same network. This is because automatic differentiation can be used to compute the derivative of the output of a neural network with respect to any input. Thus we can compute $\partial u/\partial t$ and $\partial u/\partial x_i \ \ \forall i = 1, .., d$ but also higher order derivates, which allows us to compute f. One way to program this is to use Tensorflow, which is a software library for machine learning applications mainly used in Python. More on implementation chapter 4. To compute for example $\partial u/\partial x_1$, one can use `tf.gradients(u,x1)[0]`, where `tf.gradients(u,x1)` returns a list of tensors for the derivatives of `u` with respect to the tensors in `x1`

Now the key distinction of PINNs in contrast with neural networks in other applications, is how the loss function is chosen. This loss function can be seen as a sum of two mean squared errors and is defined by

$$L(u) = \frac{1}{N_I} \sum_{i=1}^{N_I} f(t_i^I, \mathbf{x}_i^I)^2 + \frac{1}{N_B} \sum_{i=1}^{N_B} \mathcal{B}(t_i^B, \mathbf{x}_i^B, u)^2 \tag{3.2}$$

Here, the first term is evaluated at certain points in the interior of the domain Ω and certain points in time (not the initial time), represented by $\{t_i^I, \mathbf{x}_i^I\}$ for $i = 1, ..., N_I$. The second term is evaluated on certain points on the boundary of the domain for all the discrete points in time and for the initial time at all discrete points in the domain. All these points are called *collocation points*. The neural network will be trained to reduce the loss function on these collocation points. But after training is finished, the solution u can be queried on any point in the domain, so not only these collocation points. Therefore these collocation points should not be confused with a mesh grid in the finite difference method.

Note that when the first term of (3.2) is very small and not the second term, it means that the laws of physics (i.e. the PDE) are more or less obeyed but the initial and boundary conditions are not, and vice versa if it is the other way around. Both terms ought to be small to obtain accurate solutions, but neural networks might find one of the two much harder to minimize then the other, depending on the particular problem. Therefore, it may be a good idea to give

weights to each of the two contributions, where a high weight to the first term may be appropriate when the first term is hard to minimize, for example. This has been thoroughly investigated[8] by Oosterlee et al. First, they investigated whether the loss function given by eq. (3.2) is mathematically justified. Second, they theoretically derived an optimal weight for each of the two terms.

3.2 Mathematical motivation for the custom loss function

To simplify the analysis, Oosterlee et. al considered only time independent PDE's. To this end, we can rewrite (3.1) to

$$\begin{aligned} &\mathcal{N}(\mathbf{x}, u) = 0 \quad \text{in } \Omega \subset \mathbb{R}^d \\ &\mathcal{B}(\mathbf{x}, u) = 0 \quad \text{on } \partial\Omega \end{aligned} \tag{3.3}$$

where again $\mathcal{N}$ is a (nonlinear) differential operator but now $\mathcal{B}$ only denotes boundary conditions. In particular we are interested in $\Omega = [0, 1]$, $\mathcal{N}(\mathbf{x}, u) = -\varepsilon u'' + vu'$, $\mathcal{B}(1, u) = u(1) - 1$ and $\mathcal{B}(0, u) = u(0) - 0$ but we will continue the analysis for the general case, eq. (3.3). Now the loss function simplifies to

$$L(u) = \frac{1}{N_I} \sum_{i=1}^{N_I} \mathcal{N}(\mathbf{x}_i^I, u)^2 + \frac{1}{N_B} \sum_{i=1}^{N_B} \mathcal{B}(\mathbf{x}_i^B, u)^2 \tag{3.4}$$

with $\{x_i^B\}$ now only being points on the boundary of the domain for $i = 1, ..., N_B$ instead of points in time also. An important observation is that the more points we sample from the domain (i.e. taking N_I and N_B larger), the more the two terms of (3.4) tend to the average of $\mathcal{N}^2$ over Ω and $\mathcal{B}^2$ over $\partial\Omega$ respectively. More specifically, we obtain the *loss functional*

$$\hat{L}(u) = \frac{1}{|\Omega|} \int_\Omega \mathcal{N}(\mathbf{x}, u)^2 d\Omega + \frac{1}{|\partial\Omega|} \int_{\partial\Omega} \mathcal{B}(\mathbf{x}, u)^2 d\Gamma \tag{3.5}$$

which is denoted by a hat. Therefore note that (3.4) is actually an approximation of (3.5). Another observation is that if $L(u) = 0$, then u is not necessarily a solution of the PDE, while if $\hat{L}(u) = 0$, then it certainly is. This can be seen by the integrands of (3.5), which are non-negative and only zero when $\mathcal{N} = \mathcal{B} = 0$, while if $L(u) = 0$ then this only ensures that the PDE is satisfied on a finite number of points. This makes $\hat{L}$ more related to the PDE than L, which is useful for mathematical analysis.

Since we've seen that the minimizer of $\hat{L}(u)$ coincides with the solution of the PDE, ideally, we would use it as the loss of the neural network. But since the integrals need to be approximated, we have to settle for (3.4). This raises the question whether it is a good idea at all to minimize (3.4. Oosterlee et al. have argued that it is, as follows.

It has been shown (Theorem 1 in [8]) that if boundary value problem (3.3) is well-posed (meaning that a solution exists, is unique and depends continuously on boundary conditions), and if the exact solution is given by $\hat{u}$, then for any $\varepsilon > 0$ there exists a $\delta > 0$ such that

$$\hat{L}(u) < \delta \implies ||u - \hat{u}|| < \varepsilon \tag{3.6}$$

where the authors used the L_1 norm for $||.||$. This means that if some accuracy ε is desired for the computed solution u, one can always find a bound on the loss functional. Therefore, it is likely that if the value of $\hat{L}(u)$ is small, that the computed solution is accurate. Furthermore, it is reasonable to assume that (3.4) is a good approximation for (3.5), given a sufficiently high number of collocation points. Thus, it is also likely that if $L(u)$ is small, that the solution is also accurate.

But eq. (3.6) tells us more. It also justifies using a neural network for the purpose of minimizing the loss $L(u)$. Although neural networks can approximate functions arbitrarily well, they cannot represent all functions exactly. Thus it is unlikely that it finds the exact minimizer of $L(u)$. However, (3.6) tells us that this is not a problem: approximating the minimzer can still result in accurate solutions.

Due to the current discrepancy between the theory of machine learning and the performance of neural networks, it is not clear whether neural networks can indeed reach a sufficiently low value for the loss. This needs to be investigated by doing experiments. One thing that could complicate this, is that the loss functional $\hat{L}(u)$ has local minima. This would translate into local minima in the parameter space Θ of the neural network, which do not need to be of similar quality. It turns out however, that if (3.3) represents a linear PDE (which is the case for the convection-diffusion equation), that not only are there no local minima in the loss functional, but it is even convex[8]. That is, for any two functions $u_1(\mathbf{x})$ and $u_2(\mathbf{x})$ represented by a neural network, and for any $t \in [0, 1]$ the inequality

$$\hat{L}(tu_1 + (1-t)u_2) \leq t\hat{L}(u_1) + (1-t)\hat{L}(u_2)$$

holds. This does not mean that the parameter space Θ has no local minima for the loss function L. In fact, local minima in this parameter space is very common in any application of neural networks, but in many applications this is not problematic. This somewhat justifies the use of local optimization algorithms like L-BFGS-B instead of global ones.

3.3 Interior and boundary loss weights

By building further on the mathematical analysis of the methods by Raissi et al. we can modify those methods. Recall that the two components of the loss function (3.4) may be of different levels of difficulty to minimze. This difference could span orders of magnitude. Thus it makes sense to introduce a weight $\lambda \in (0, 1)$ for the

interior component of the loss function and $1-\lambda$ for the boundary component. To get an idea of what this weight might be, we again need to resort to mathematical analysis of loss functional (3.5). But now we modify this functional te be

$$\hat{L}(u) = \lambda \int_{\Omega} \mathcal{N}(\mathbf{x}, u)^2 d\Omega + (1-\lambda) \int_{\partial\Omega} \mathcal{B}(\mathbf{x}, u)^2 d\Gamma \tag{3.7}$$

where we absorb the constants in front of the integrals in (3.5) into the new loss weight λ.

It is not immediately clear how we should choose the optimal value for λ, or how we even should go about analyzing this. One way to do this, is to choose λ such that there is a strong relation between the value of the loss functional and the accuracy of the computing solution. That is to say, if we have a low loss then this corresponds to a computed solution that is close, in some sense, to the exact solution, and vice versa. In this way, we can be more certain that spending computational resources to obtain lower losses result in more accurate solutions.

One helpful assumption is that if we put more weight on the interior, lets say, then it improves the loss on the interior approximately as much as the loss on the boundary deteriorates. That is, changing the loss weight changes the relative contributions of the losses, but the total loss stays the same.

Probing this relation between the value of the loss functional and the accuracy is possible by using the absolute error as a measure for accuracy, so the L_1 norm. However, this is probably not a good idea. From experimental studies, it has been observed that where the exact solution has large derivatives, the absolute error on these derivatives is large as well. Therefore, if derivatives of the exact solution are very small on most of the domain Ω, then this would be easy for a neural network to approximate without the loss needing to be small. This would result in a weak relation between the loss and this measure of accuracy. Thus we could take into account the derivatives as well to measure accuracy, and in particular some form of relative error. More precisely, we will use the following definition:

Definition 3.3.1. A candidate solution u is called ε-close to the true solution $\hat{u}$ if it satisfies

$$|Du(\mathbf{x}) - D\hat{u}(\mathbf{x})| \leq \varepsilon |D\hat{u}(\mathbf{x})| \tag{3.8}$$

where D can be any mixed partial derivative operator of any order, including the identity. It needs to hold for all D and for all $\mathbf{x} \in \Omega$.

For example,

$$Du(\mathbf{x}) = \frac{\partial^{10} u(\mathbf{x})}{\partial x_1^2 \partial x_2^5 \partial x_d^3}$$

is one such possible D. Now let a general, linear PDE be given by (3.3). For some computed solution, we would ideally like to find a certain bound on the left hand side of (3.8), given a bound on the loss functional. That is, we want to guarantee ε-closeness. Unfortunately, this is currently not proven. However, the reverse is shown to be true[8]. To be precise, it is shown that if u is ε-close to $\hat{u}$, then it holds that

$$\hat{L}_I(u) := \int_\Omega \mathcal{N}(\mathbf{x}, u)^2 d\Omega \leq \varepsilon^2 \int_\Omega N(\mathbf{x}, \hat{u})^2 d\Omega \tag{3.9}$$

and

$$\hat{L}_B(u) := \int_{\partial\Omega} \mathcal{B}(\mathbf{x}, u)^2 d\Gamma \leq \varepsilon^2 \int_{\partial\Omega} B(\mathbf{x}, \hat{u})^2 d\Gamma \tag{3.10}$$

where $N(\mathbf{x}, u)$ is $\mathcal{N}(\mathbf{x}, u)$ but without the source terms (terms that don't depend on u) and each term involving u is taken in absolute value. $B(\mathbf{x}, u)$ is similarly defined. If we now label

$$M_I(\hat{u}) = \int_\Omega N(\mathbf{x}, \hat{u})^2 d\Omega, \tag{3.11}$$

$$M_B(\hat{u}) = \int_{\partial\Omega} B(\mathbf{x}, \hat{u})^2 d\Gamma \tag{3.12}$$

then we obtain that

$$\hat{L}(u) = \lambda \hat{L}_I(u) + (1-\lambda)\hat{L}_B(u) \leq \varepsilon^2 \left[\lambda M_I(\hat{u}) + (1-\lambda) M_B(\hat{u})\right] \tag{3.13}$$

Note that the inequalites given by (3.9), (3.10) and (3.13) give necessary conditions for u to be ε-close to $\hat{u}$, but not sufficient ones. This means that if these inequalities are satisfied, then ε-closeness is possible, for ε given in the above equations, but not guaranteed. Conversely, if one of these is violated, then ε-closeness is being ruled out.

Optimal loss weight

Next, we can use the previous discussion to devise a strategy to obtain an expression for λ that is optimal in some sense. The plan is the following: we can choose λ such that it is possible for u to be ε-close to $\hat{u}$, for as small an ε as possible. That is, given a fixed total loss $\hat{L}(u)$, we can tweak λ such the inequalities given above are satisfied for the smallest possible ε. This ties in with a previous assumption that $\hat{L}(u)$ stays approximately constant as we vary λ. Note that with this strategy we cannot guarantee ε-closeness. We can merely make sure that it is not ruled out, for small ε ideally.

To this end, let u be a candidate solution of a linear PDE so that the loss value $\hat{L}(u)$ is fixed. We want to derive a new necessary condition for ε-closeness in just one equation, involving $\hat{L}(u), \lambda$ and ε. We will see later that (3.13) is not suitable. Then we want to isolate ε on one side of the inequality, to see which values of ε satisfy this condition, of which the smallest one is of interest (i.e.

using equality). Finally, we can vary λ such that ε is smallest.

To do this, note that

$$\hat{L}_I(u) = \frac{1}{\lambda}\left[\hat{L}(u) - (1-\lambda)\hat{L}_B(u)\right] \leq \frac{1}{\lambda}\hat{L}(u) \tag{3.14}$$

since the boundary loss $\hat{L}_B(u)$ is non-negative. Similarly, we find

$$\hat{L}_B(u) \leq \frac{1}{1-\lambda}\hat{L}(u) \tag{3.15}$$

which gives rise to new necessary conditions

$$\hat{L}_I(u) \leq \frac{1}{\lambda}\hat{L}(u) \leq \varepsilon^2 M_I(\hat{u}) \implies \hat{L}(u) \leq \varepsilon^2 \lambda M_I(\hat{u}) \tag{3.16}$$

$$\hat{L}_B(u) \leq \frac{1}{1-\lambda}\hat{L}(u) \leq \varepsilon^2 M_B(\hat{u}) \implies \hat{L}(u) \leq \varepsilon^2 (1-\lambda) M_B(\hat{u}) \tag{3.17}$$

and they can be combined into

$$\hat{L}(u) \leq \varepsilon^2 \min\{\lambda M_I(\hat{u}), (1-\lambda)M_B(\hat{u})\}. \tag{3.18}$$

Next, we can isolate ε:

$$\varepsilon \geq \sqrt{\frac{\hat{L}(u)}{\min\{\lambda M_I(\hat{u}), (1-\lambda)M_B(\hat{u})\}}} \tag{3.19}$$

which represents all ε for which u may possibly be ε-close to $\hat{u}$. As stated, we are interested in the smallest such ε, so we can use equality. Finally, we can minimize ε under this constraint, by varying λ. But minimizing

$$\sqrt{\frac{\hat{L}(u)}{\min\{\lambda M_I(\hat{u}), (1-\lambda)M_B(\hat{u})\}}}$$

is equivalent to minimizing

$$\frac{1}{\min\{\lambda M_I(\hat{u}), (1-\lambda)M_B(\hat{u})\}} = \max\left\{\frac{1}{\lambda M_I(\hat{u})}, \frac{1}{(1-\lambda)M_B(u)}\right\}.$$

The smallest maximum occurs when both arguments are equal, since the arguments change continuously with λ. Thus we finally obtain

$$\lambda M_I(\hat{u}) = (1-\lambda)M_B(\hat{u}) \tag{3.20}$$

which implies that

$$\lambda = \frac{M_B(\hat{u})}{M_B(\hat{u}) + M_I(\hat{u})}. \tag{3.21}$$

which is the optimal loss weight in a sense that we just described. It should be noted that information about the exact solution is needed to compute this λ, which

is often not available. However, we can also estimate this λ by replacing $\hat{u}$ with the approximated solution u in the right hand side of (3.21). This approximated value for (3.21) can be iteratively updated after each iteration of training the neural network. Depending on the boundary conditions of the particular problem though, it may be possible to compute $M_B(\hat{u})$ in advance.
However, using this heuristic for λ can present problems for certain PDEs, in particular, those that admit a trivial solution on the interior of the domain. That is, a trivial solution $u = 0$ that solves the PDE but does not necessarily obey the boundary conditions. This is because $M_I(u = 0) = 0$ and $\hat{L}_I(u = 0) = 0$ which implicates that

$$\hat{L}(u = 0) = \lambda \hat{L}_I(u = 0) + (1 - \lambda)\hat{L}_B(u = 0) = 0. \tag{3.22}$$

Thus, using this heuristic for λ, the loss can be completely minimized with the trivial solution. To remedy this, the authors used a scaling factor[8] for the modified loss functional which they called *magnitude normalization*. In this paper this heuristic will not be investigated, as we know the exact solution for our toy problem.

3.4 Deep Least Squares

The idea of translating a PDE into a minimization problem, namely finding the minimum of some loss functional, is certainly not new. A method that is more advanced than the finite difference method, called the least squares finite element method (FEM), extensively makes use of such a functional. The functional given by (3.5) can be seen as a *least squares* (LS) formulation, since the problem is formulated as minimizing the integral or sum of some squared function.

Lazarov et al. (1994) have studied[1] a different functional though, that is somewhat related to (3.5), in the context of FEM. There they studied second order (elliptic) PDEs and decomposed it into a system of first order PDEs. Then, they assembled a least square functional similar to (3.5), which they called the First Order System Least Squares (FOSLS) functional. However, (3.5) is the square of an L_2 norm which they did not use. Instead, they used more advanced norms related to Sobolev spaces, which is beyond the scope of this text. Moreover, they did not include a term to enforce the boundary conditions into the functional. This was not problematic back then, as finite element methods are not as challenged with the boundary conditions as artificial neural networks.

While Lazarov et al. have extensively analysed this FOSLS functional theoretically, Chen et al. have proposed to apply these ideas to PINNs[2]. Also, in [2] they summarized the most relevant theoretical results from ([1]). First, we will introduce this theoretical FOSLS functional. Then, we can modify it into a loss function that an artificial neural network is able to work with. Now let us introduce the problem that these methods have been applied to.

Problem formulation

Let Ω again be a bounded domain in $\mathbb{R}^d$ with boundary $\partial\Omega = \Gamma_D \cup \Gamma_N$ with Γ_D the part of the boundary where a Dirichlet boundary condition will be imposed and Γ_N the part with a Neumann boundary condition. Let a second order partial differential equation be given by

$$-\nabla \cdot (A\nabla u) + Xu = f \quad \text{in } \Omega \tag{3.23}$$

and with boundary conditions

$$u = g_D \quad \text{on } \Gamma_D \tag{3.24}$$
$$-\mathbf{n} \cdot A\nabla u = g_N \quad \text{on } \Gamma_N \tag{3.25}$$

where f, g_D and g_N are given functions, $A(x)$ is a symmetric matrix of functions and X is a linear differential operator of order at most one (for example, $Xu = 4 * \partial u/\partial x_1$). Now if f, g_D and g_N are sufficiently well behaved functions, then it is known that this problem is well-posed. One of those regularity conditions is that f needs to be in $L^2(\Omega)$, the space of square integrable functions over domain Ω. The other two (g_D and g_N) need to be in certain spaces that are more generalized Sobolev spaces.

Next, we can transform the second order PDE into a system of first order PDEs by introducing the variable $\boldsymbol{\sigma} = -A\nabla u$. Hence we obtain the new formulation:

$$\nabla \cdot \boldsymbol{\sigma} + Xu = f \quad \text{in } \Omega \tag{3.26}$$
$$\boldsymbol{\sigma} + A\nabla u = \mathbf{0} \quad \text{in } \Omega \tag{3.27}$$

with boundary conditions

$$u = g_D \quad \text{on } \Gamma_D \tag{3.28}$$
$$\mathbf{n} \cdot \boldsymbol{\sigma} = g_N \quad \text{on } \Gamma_N \tag{3.29}$$

FOSLS formulation

For this problem, the FOSLS functional, as obtained from [1], is given by

$$\tilde{\mathcal{G}}(u, \boldsymbol{\sigma}, \mathbf{f}) = ||\nabla \cdot \boldsymbol{\sigma} + Xu - f||^2 + ||A^{-1/2}\boldsymbol{\sigma} + A^{1/2}\nabla u||^2 \tag{3.30}$$

with $\mathbf{f} = (f, g_D, g_N)$ representing the data and $||.||$ represents the L_2 norm. The second term enforces (3.27) by multiplying that equation by $A^{-1/2}$ and taking the norm of the residual. The power of matrix A should be interpreted as follows. If A is a symmetric positive definite matrix, then there exists a matrix B such that $A = BB^T$. This is the Cholesky decomposition. In this case, $A^{1/2}$ is defined as this matrix B and $A^{-1/2} = B^{-1}$.

The FOSLS formulation of the original second order PDE is now to find the minimum of this functional for sufficiently well behaved u and $\boldsymbol{\sigma}$. Note that this

functional does not explicitly enforce the boundary conditions. From theoretical results in [1] (coercivity and boundedness of (3.30)) it can be concluded that this minimization problem is well-posed.

As stated before, (3.30) is not as suitable for PINNs as for the FEM, since the boundary conditions are not explicitly enforced. These boundary terms are added to the loss functional $\tilde{\mathcal{G}}$ to obtain $\mathcal{G}$. Due to the regularity constraints of the solution and of the data, for these extra terms a different norm is used. The modified functional becomes

$$\begin{aligned}\mathcal{G}(u, \boldsymbol{\sigma}, \mathbf{f}) = ||\nabla \cdot \boldsymbol{\sigma} + Xu - f||^2 + ||A^{-1/2}\boldsymbol{\sigma} + A^{1/2}\nabla u||^2 \\ + \alpha_D ||u - g_D||^2_{1/2} + \alpha_N ||\mathbf{n} \cdot \boldsymbol{\sigma} - g_N||^2_{-1/2}\end{aligned} \tag{3.31}$$

where $||.||_{1/2}$ and $||.||_{-1/2}$ are norms on Sobolev spaces. Again it follows from functional analysis that minimizing this funtional is a well-posed problem. Here, α_D and α_N are scaling factors. These might be needed, because the norms are not of the same scale, meaning, they could differ orders of magnitude. This is because the functions inside those norms are constrained by different levels of regularity.

Due to the regularity constraints on the solutions u and $\boldsymbol{\sigma}$, not every activation function can be used inside the neural network. In particular, for regular PINNs the activation function is not allowed to be piecewise linear. This excludes the use of the so called ReLU function, which is defined as $\max\{0, x\}$ for $x \in (-\infty, \infty)$, which is a very popular choice in state-of-the-art deep learning methods. For FOSLS, this restriction is not present, but we still will not consider the ReLU function.

Deep FOSLS

Now we have set the theoretical foundation to apply deep learning to this problem. An artificial neural network can be used to approximate the functional given by (3.31) and use this approximation as its loss function. In [2], this approximation is done differently than in previous sections. Here, the approximation looks more like a Riemann sum. To this end, the domain of computation Ω is split into a partition of open sets $\mathcal{T}$, meaning that the union of the closure of its elements is $\overline{\Omega}$. Similarly, let $\mathcal{E}_D$ and $\mathcal{E}_N$ be a partition of Γ_D and Γ_N respectively. Let $\mathbf{x}_T$ be an interior point of $T \in \mathcal{T}$ and let $\mathbf{x}_E$ be an interior point on a boundary part $E \in \mathcal{E}_D$ or $E \in \mathcal{E}_N$. Let $|.|$ denote the volume or measure of such an interior or

boundary part. The loss function is now given by

$$\begin{aligned} L(u, \boldsymbol{\sigma}, \mathbf{f}) = & \sum_{T \in \mathcal{T}} \left[\nabla \cdot \boldsymbol{\sigma}(\mathbf{x}_T) + Xu(\mathbf{x}_T) - f\right]^2 |T| \\ & + \sum_{T \in \mathcal{T}} \left[A^{-1/2}\boldsymbol{\sigma}(\mathbf{x}_T) + A^{1/2}\nabla u(\mathbf{x}_T)\right]^2 |T| \\ & + \alpha_D \sum_{E \in \mathcal{E}_D} [u(\mathbf{x}_T) - g_D]^2 |E| \\ & + \alpha_N \sum_{E \in \mathcal{E}_N} [\mathbf{n} \cdot \boldsymbol{\sigma}(\mathbf{x}_T) - g_N]^2 |E|^3 \end{aligned} \tag{3.32}$$

Note that the third and fourth term might need local scaling factors when the norms of (3.31) are approximated by weighted L_2 norms.

The first two terms correspond to the PDE, while the last two terms enforce the boundary conditions. As such, we can experiment with different weights. We can give the first two some weight λ and the last two the weight $1 - \lambda$. It is not clear whether the optimal weights derived in [8] will provide good results for this FOSLS formulation. This is because in [8] they analyzed a different loss functional, and that analysis does not directly translate over to this formulation. Thus, to rigorously analyze optimal weights for the FOSLS functional, we would have to start from scratch. Still, it is interesting to see if (3.21) happens to work out for this FOSLS functional.

In addition, it is possible to give all four terms separate weights, but that is a suggestion for further research. As guidance, we will only state that from some limited experimentation, we have found that if a Dirichlet boundary condition on σ is imposed at $x = 1$ instead of the Dirichlet condition for u, then sometimes σ does not seem to be the derivative of u. This suggests that a higher weight should be given to the fourth term of eq. (3.32), which enforces the relationship between σ and u. We will not elaborate on these experiments.

CHAPTER 4

Implementation

In this chapter we discuss how the theory in the previous chapter should be applied to the convection diffusion equation, in particular boundary value problem (1.2). In addition, we will discuss how to implement this application in Python code. This discussion will take place along all the methods previously discussed. There will be two classes of Python implementations. For general PINNs as described by Raissi et al. we have used DeepXDE, which is a Python library developed by Lu et al.[4] This library is also used to implement custom weights for the interior and boundary losses for the general PINN. To implement the FOSLS functional along with capabilities for custom weights, we have written our own code, which is loosely based on code written by Raissi et al. Since we are interested in convection-dominated convection-diffusion, we can take $v = 1$ for simplicity and gradually decrease ε to increase the difficulty of the problem, in eq. (1.2).

4.1 General PINNs and custom weights for convection-diffusion

In section 3.2, after eq. (3.4), we have formulated what the operators $\mathcal{N}$ and $\mathcal{B}$ are for our particular problem. It then follows that the general loss function for 1D convection-diffusion is given by

$$L(u) = \lambda\Big(\frac{1}{N}\sum_{i=1}^{N}[-\varepsilon u''(x_i) + u'(x_i)]^2\Big) + (1-\lambda)\Big([u(1)-1]^2 + u(0)^2\Big) \tag{4.1}$$

with N the number of collocation points on the interior of $[0, 1]$ and $\{x_i : i = 1, .., N\}$ the set of all these points. These points can be chosen to be equidistant, but in the DeepXDE library they are randomly sampled. Eq. (4.1) represents the unweighted loss function (or equivalently, with $\lambda = 1/2$). We will now derive the theoretically optimal weight according to eq. (3.21).

Note that for the exact solution u of boundary value problem (1.2) the following expressions hold:

$$u(x) = \frac{e^{x/\varepsilon} - 1}{e^{1/\varepsilon} - 1} \tag{4.2}$$

$$u'(x) = \frac{1}{\varepsilon} \frac{e^{x/\varepsilon}}{e^{1/\varepsilon} - 1} \tag{4.3}$$

$$u''(x) = \frac{1}{\varepsilon^2} \frac{e^{x/\varepsilon}}{e^{1/\varepsilon} - 1} \tag{4.4}$$

Note that, as explained after eq (3.10), for our problem $\mathcal{N}(x, u) = -\varepsilon u''(x) + u'(x)$ so that

$$N(x, u) = \varepsilon u''(x) + u'(x) = \frac{2}{\varepsilon} \frac{e^{x/\varepsilon}}{e^{1/\varepsilon} - 1} := \xi e^{x/\varepsilon} \tag{4.5}$$

with ξ defined as $(2/\varepsilon)/(e^{1/\varepsilon} - 1)$. Next, it follows from eq (3.11) that

$$M_I(u) = \xi^2 \int_0^1 e^{2x/\varepsilon} dx = \xi^2 \frac{\varepsilon}{2} \left[e^{2x/\varepsilon} \right]_0^1 = \xi^2 \frac{\varepsilon}{2} \left(e^{2/\varepsilon} - 1 \right) \tag{4.6}$$

Similarly, for our problem, $\mathcal{B}(1, u) = u(1) - 1$ and $\mathcal{B}(0, u) = u(0) - 0$ so that $B(1, u) = u(1)$ and $B(0, u) = u(0) = 0$. Note that boundary integrals on one dimensial domains simply become evaluations at the boundary, so that eq. (3.12) becomes

$$M_B(u) = \int_{\partial\Omega} B(\mathbf{x}, u)^2 d\Gamma = (u(1))^2 + (u(0))^2 = 1 \tag{4.7}$$

For a given value of ε, we are now ready to evaluate λ using eq. (3.21). Note that for very small ε, the interior weight λ is also very small, which may result in the underlying physics being ignored. On the other hand, for thin boundary layers, the solution looks like a trivial solution for most of the domain. By doing algebraic manipulations it is easy to see that for small ε, $\lambda \approx \varepsilon/2$. Fortunately, the weighted and unweighted loss function can easily be implemented in DeepXDE.

4.2 DeepXDE

DeepXDE is a Python library that makes it easy for researchers and students, who are not so skilled in either Python programming or machine learning, to implement artificial neural networks for solving time-dependent and time-independent PDEs. This is accomplished by encoding the neural network, the loss function and the training process in different files, that the user does not need to modify. The user simply needs to write one Python script where the user states the problem to be solved and what parameters should be used, such as the number of collocation points, the desired neural network architecture, the activation function and the loss weighs for the interior and boundary losses. To accommodate students, the author has written several example files where various PDEs are being solved.

There it can be seen that the PDE should be formulated in the general form $\mathcal{N}(\mathbf{x}, u) = 0$ and that the Python function `pde(x,u)` returns $\mathcal{N}(\mathbf{x}, u)$. Furthermore, the `func` function can be used to specify the boundary conditions by stating the exact solution, but it also suffices to return an arbitrary function that coincides at the boundary. But the exact solution is needed if you want to use this function to plot the exact solution. However we do not recommend this. It is easier to seperately code the exact solution for plotting purposes to take care of OverFlow errors. In particular, while the exact solution $u(x) \in [0, 1]$ for any $x \in [0, 1]$, it is the fraction of two very large numbers (for small ε), as in eq. (4.2). Python has trouble doing arithmetic with large numbers. To remedy this, the `decimal` library can be used. It is cumbersome to implement this in the `func` function as this function is being used multiple times by the algorithm, with arrays of different lengths for the input `x` each time.

Furthermore, one needs to create a `deepxde.Model` object, that represents the problem to be solved and the neural network architecture used. Then, `model.compile` needs to be called, which has an optional argument called `loss_weights`. Here the custom loss weights for the interior and boundary loss can be implemented, by passing to this argument a 1×2 Numpy array containing [interior weight, boundary weight].

After training is done, one can call `model.predict(x,operator=pde)` to obtain an array of residuals, for the array of interior points `x`. To obtain the computed derivative of u, a new Python function can be implemented that returns this derivative, and this can be called by the operator argument of `model.predict`.

As configurable as DeepXDE is, it also has its limitations. It cannot be used to implement the FOSLS functional because we need a special kind of neural network structure, as we will see. We can only increase the size of the network, but not modify its sparsity structure. It is also not easy to modify the loss function more than just scaling it, which is another reason why it is hard to implement FOSLS. Finally, it is not possible to modify the interior and boundary weights during training. This would be needed to implement approximations for λ in eq. (3.21) or to use magnitude normalization (see section 3.3), but this is not relevant for this paper.

4.3 Implementing the FOSLS functional

For our boundary value problem we need to find an appropriate expression for the loss function given by eq. (3.32). To this end, lets translate eq. (3.23) for 1D convection-diffusion. In this case $A = \varepsilon$ and the nabla operators are simply first derivatives, $\partial/\partial x$. Next, $Xu = \partial u/\partial x$ and $f = 0$. It follows that in 1D, σ is a scalar and is given by $\sigma(x) = -\varepsilon u'(x)$. Furthermore, $\Gamma_D = \{0, 1\}$ and $\Gamma_N = \varnothing$ so

that the fourth term of (3.32) vanishes. The powers of matrix A can be replaced by ordinary scalar powers of ε. If the interval $[0, 1]$ is split by an equidistant grid of points, with distances Δx (including the endpoints), then $|T|$ can be replaced by Δx in the loss function. As for $|E|$, this factor is not needed since the third term of (3.32) will represent an approximation of the zero-th dimensional integral embedded in the norm of the third term of (3.31). Instead, we can for this term just use evaluations at the boundary. In conclusion, the weighted loss function becomes

$$\begin{aligned} L(u,\sigma) = \lambda &\left(\sum_{i=1}^{N} \Big[\sigma'(x_i) + u'(x_i)\Big]^2 \Delta x + \sum_{i=1}^{N} \Big[\frac{1}{\sqrt{\varepsilon}}\sigma(x_i) + \sqrt{\varepsilon}u'(x_i)\Big]^2 \Delta x \right) \\ &+ (1-\lambda)\left(\Big[u(1) - 1\Big]^2 + u(0)^2 \right) \end{aligned} \tag{4.8}$$

To test the equally weighted boundary and interior loss, we can take $\lambda = 1/2$.

Neural network for FOSLS

The authors of [2] argued that since u and $\boldsymbol{\sigma}$ are independent, an efficient strategy is to implement a branched neural network, so that training the two outputs can occur independently. The branched network can have the following form.

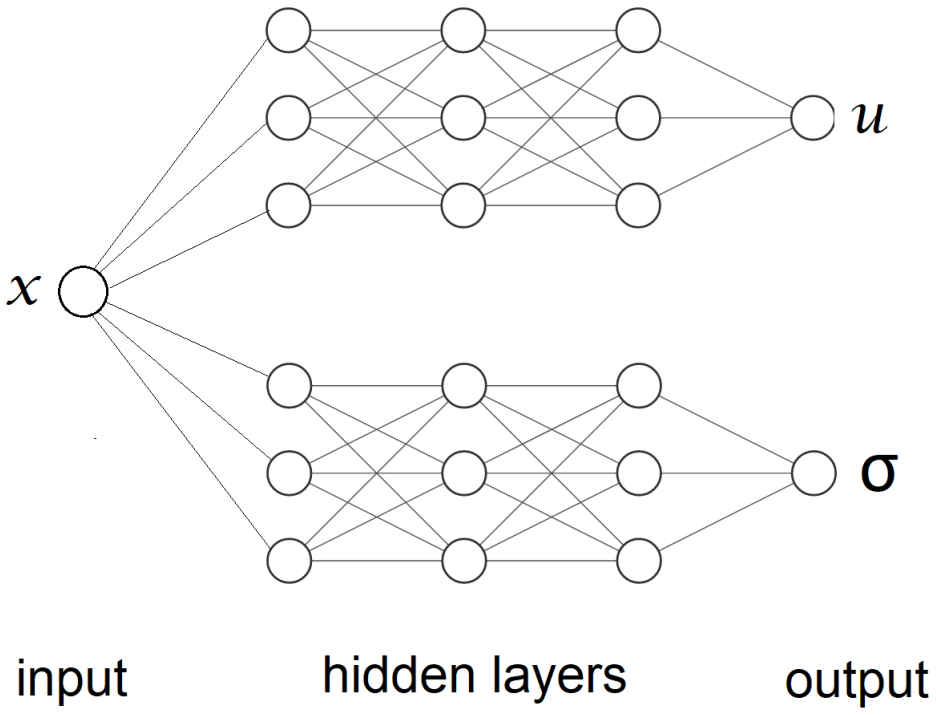

Figure 4.1: A branched neural network to implement the FOSLS functional.

The number of hidden layers and the number of neurons per hidden layer can be adjusted. The authors used four hidden layers. In our implementation, we

always use a neural network that is symmetric in the upper and lower branch. Note that the network is *dense* to the first hidden layer, meaning that every neuron from the input layer is connected to every neuron in the second layer. This is also the case if we choose two inputs: one for x and one for ε. There are no connections between the branches to keep the training separate. Within each branch, the layers are again densely connected.

Recall from chapter 2 that a network diagram merely represents a composition of matrix multiplications and activation functions. Normally, neural networks are densely connected which result in dense weight matrices. For branched networks, the weight matrix from one hidden layer to the next is a block diagonal matrix with four blocks. Each of the four blocks has the same number of rows and columns and can be non-square. This also holds for the matrix corresponding to the last hidden layer to the output layer. This obviously doesn't hold for the first weight matrix (from input to first hidden layer) which is a dense matrix.

These weight matrices, along with the bias vectors, are normally passed over to the optimizer as the variables to be optimized. We cannot do this with block diagonal matrices however, since there is no guarantee that after optimization, the matrices stay sparsely structured. Instead, the variables passed to the optimizer should be the upper left and lower right blocks of the weight matrices. After each iteration of training, these blocks should be reassembled into a block diagonal matrix to do the matrix multiplications and compute u and σ, as well as the loss function. In Tensorflow, this can be done by horizontally concatenating the upper left block with a matrix consisting of zeros of the same dimension. To do this use `tf.concat([W1,0],1)`. The same should be done for the lower right block. Finally, we can vertically concatenate these wide matrices with `tf.concat([A,B],0)`.

4.4 General remarks

As stated before, it is problematic to compute $e^{1/\varepsilon}$ if ε is small. This can be fixed by using the Decimal library. Not only should this be done when computing the exact solution, but also when computing λ using eq. (3.21).

From experiments it has become clear that the parameter space Θ is difficult for the optimizer L-BFGS-B to navigate through. It seems like the loss function is relatively flat for a large portion of the landscape, while there might be a metaphorical sink somewhere that is difficult to find. One way to observe this is to run the same code a few times but not obtaining the same results. Another way it became clear was when sometimes the value of the loss barely changes for thousands of iterations, when suddenly it does find the sink and it descends rapidly. Therefore, the hyperparameters of the optimizer should be configured such that the training process does not stop too early.

We have observed that it is good practice to not limit the maximum number of iterations below 15000. Furthermore, the algorithm can stop when the norm of the projected gradient of the loss function gets below a certain threshold, called the `gtol` parameter. This should be avoided though, as it turns out that better solutions can be obtained if we continue the training process. This can be done by setting the `gtol` parameter of L-BFGS-B, which is the threshold for the gradient under which training stops, to 1e-8. Setting this too low might not be a good idea, as rounding errors from decimal expansions to binary expansions can start to play a role. This helped to greatly reduce the unpredictability of the outcome. In DeepXDE this is done by modifying the `train.py` file.

Another aspect that can cause L-BFGS-B to terminate early, is using single-precision floating point data. Instead, double-precision should be used. This is easily done in the DeepXDE library by modifying the `config.py` file. Without DeepXDE, one needs to initialize all variables with datatype `tf.float64`. With single precision, none of the methods presented in this paper managed to create a good solution for ε as high as 0.01. However with double-precision, we were able to take ε orders of magnitude lower and still get good results.

CHAPTER 5

Experiments

In this chapter we will compare all the methods described in chapter 3. For notational convenience, the general loss function given by (4.1) we will label L_G and the FOSLS loss function given by (4.8) we will label L_F. Let us denote the optimal weight given by (3.21) by $\hat{\lambda}$. Since this value for λ is optimal in a vague sense, we will also use the weight $\lambda = 50\hat{\lambda}$ and $\lambda = \hat{\lambda}/50$ for both L_G and L_F. We will also call this the supra-optimal and infra-optimal weight, respectively. Recall that it is even less clear if $\hat{\lambda}$ is optimal for L_F since the optimality was derived under different settings. For all experiments we will investigate the computed solution u as well as the computed du/dx.

To measure the quality of a solution, we can compare the plot to the plot of the exact solution. But we can also compute the relative L_2 error, which is computed as

$$\text{Relative } L_2\text{-error} = \frac{||u - \hat{u}||}{||\hat{u}||} \tag{5.1}$$

where a thousand equidistant points are used on [0,1] to approximate these norms.

5.1 Performance of the general loss function

From preliminary testing it became clear that the hyperbolic tangent function outperformed the sigmoid function, thus only tanh is used in this paper. This may explain why the next results are better than the ones in [8]. There, the unweighted loss L_G caused an L_2 error in the order of 1e0 for $\varepsilon < 10^{-2}$ while here the error is one order of magnitude smaller for $\varepsilon = 10^{-4}$ (Table 5.1). For L_G, we take a neural network with 4 hidden layers and 20 neurons per layer. For L_F, we also take 4 hidden layers but 14 neurons per layer per branch. This amounts to approximately an equal number of parameters for both networks (1321 for L_G and 1346 for L_F).

The following tests are conducted by varying magnitude of ε and the number of collocation points N in the interior of [0,1] (Table 5.1 and A.1). In Fig. 5.1

we see that the error of 0.89 corresponds to a thin boundary layer like the exact solution. But if we look close (Fig. 5.2) we see that this boundary layer is not as thin as desired. The derivative has not been approximated well. This can also be seen in Table A.1. An error of 1 for σ usually means that σ is close to the zero function. It is remarkable that taking too many collocation points can negatively impact the solution (Fig. 5.3). There it can be seen that neither the boundary conditions are being obeyed nor the PDE. A partial solution is to give more weight to the boundary conditions, as in Fig. 5.4 to satisfy those. However this does not solve the fact that the PDE is neglected. From Table 5.1 it seems that $N = 128$ is ideal.

	Unweighted, L_G, error u				
ε	64	128	256	512	1024
1e-5	3.15e+01	8.92e-01	1.59e+01	1.59e+01	1.59e+01
1e-4	1.82e+00	3.83e-01	1.59e+01	1.59e+01	1.59e+01
1e-3	1.72e+00	1.90e+00	1.48e+01	1.48e+01	1.48e+01
1e-2	6.66e+00	3.24e-05	9.45e-06	1.04e-05	2.24e-05
1e-1	2.39e-05	2.47e-05	2.17e-05	3.59e-05	5.59e-05

Table 5.1: Relative L_2 error for u using loss L_G with $\lambda = 1/2$ for various numbers of collocation points.

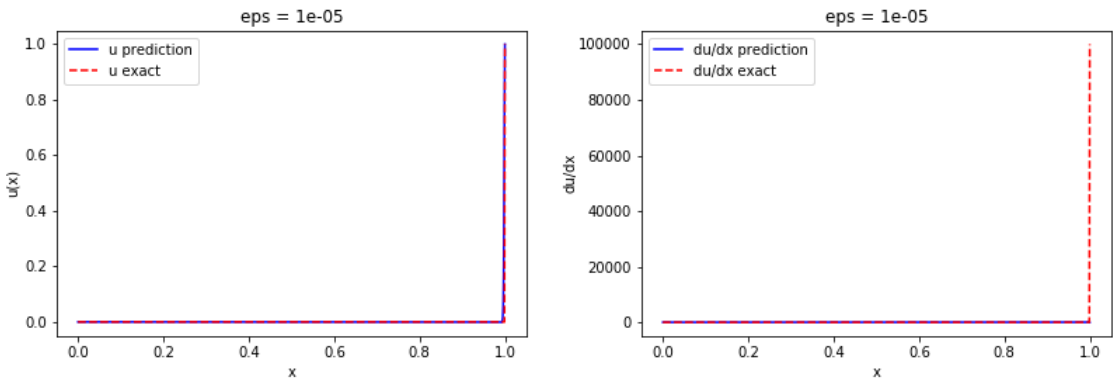

Figure 5.1: The unweighted loss L_G for $\varepsilon = 10^{-5}$ and $N = 128$

For the optimally weighted loss L_G, it seems the method is very fragile for small diffusion (Table A.2 and A.3). However for $\varepsilon = 10^{-1}$ the solution can be up to two orders of magnitude more accurate (using $N = 512$) than using the unweighted loss. The supra-optimal weight seems to be much more stable, if the number of collocation points is chosen correctly (Table A.4 and A.5). Again, $N = 128$ performs best. Note that for $\varepsilon = 0.1$, this method violates the assumption that $\lambda \in (0, 1)$. Still, it is remarkable how unstable this method becomes for this ε, because $\lambda > 1$ is equivalent to some $\lambda > 1/2$ and scaling the

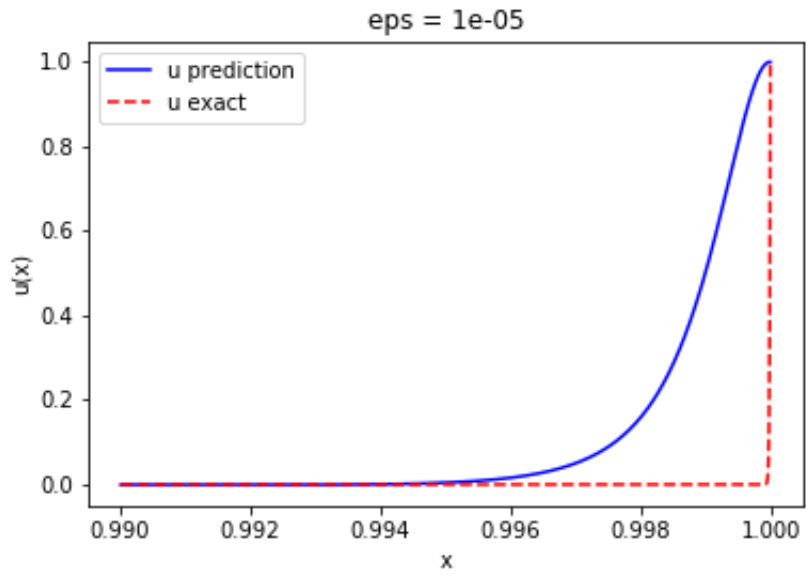

Figure 5.2: Zoomed in on Fig. 5.1

loss function (the ratio between interior and boundary loss matters).

Using L_G with $\lambda = \hat{\lambda}/50$ is even a more fragile method than using $\lambda = \hat{\lambda}$ (Table A.6 and A.7). It seems that the weight $\hat{\lambda}$ was already too small and $\hat{\lambda}/50$ makes it worse. Approximating the derivative du/dx using L_G remains difficult for all loss weights tested.

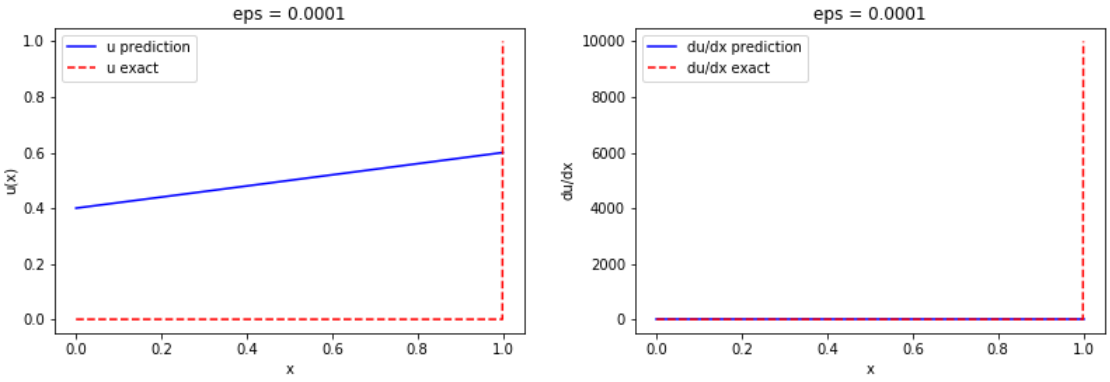

Figure 5.3: The unweighted loss L_G for $\varepsilon = 10^{-4}$ and $N = 1024$

5.2 Deep least squares performance

Next, we will investigate the performance of L_F for various loss weights. For the unweighted loss, the performance is comparable to to the unweighted loss

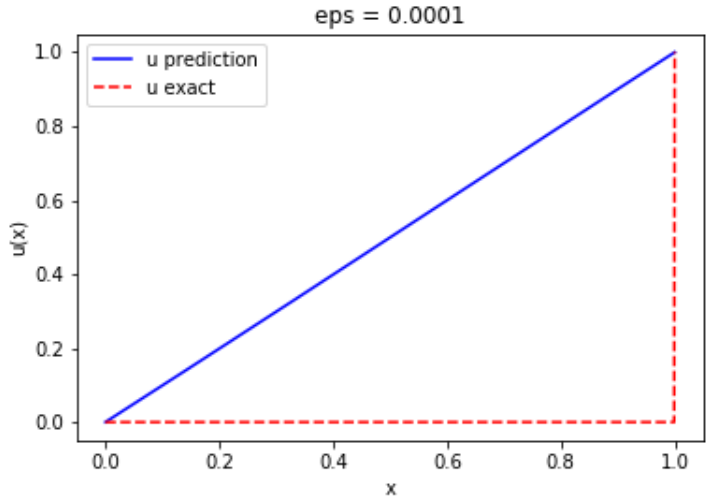

Figure 5.4: The loss L_G with $\lambda = \hat{\lambda}$, $\varepsilon = 10^{-4}$ and N=1024

function L_G (Table A.8). For $\varepsilon = 10^{-5}$, this time the best result is obtained for 64 collocation points, where the graph is similar to Fig. 5.1 but with slightly thicker boundary layer. For other values of ε, the lower three values for N are good choices.

The choice $\lambda = \hat{\lambda}$ again results in poor accuracy for $\varepsilon \leq 10^{-3}$ (Table A.10). The infra-optimal weight gives very similar results (Table A.13). Comparing $\lambda = 50\hat{\lambda}$ to $\lambda = 1/2$, we see that the accuracy slightly deteriorates (Table 5.2). For 64 collocation points and $\varepsilon = 10^{-5}$, u has a relative L_2 error of 6.04 which is illustrated in Fig. 5.5. We see that there is some instability around both boundary points.

	Supra-optimal weight, L_F error u				
ε	64	128	256	512	1024
1e-5	6.04	18.2	18.2	18.2	18.2
1e-4	18	4.97	17.9	18.2	18.1
1e-3	1.1	9.03	0.421	0.694	15.5
1e-2	7.86e-03	1.85e-03	3.45e-04	1.11e-03	1.33e-03
1e-1	5.92e+151	inf	8.67e+151	1.04e+145	8.44e+151

Table 5.2: Relative L_2 error for u using loss L_F with $\lambda = 50\hat{\lambda}$ for various numbers of collocation points.

5.3 Modifying the neural network architecture

Neural networks are in essence black box algorithms. To make the inner workings more interpretable, we can modify the shape as follows. We can use five hidden

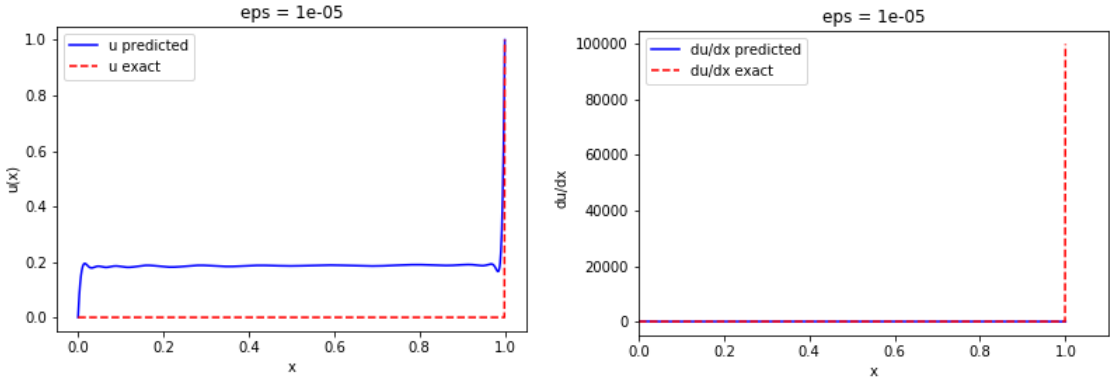

Figure 5.5: Output using L_F, $\lambda = 50\hat{\lambda}$, $N = 64$ and $\varepsilon = 10^{-5}$

layers, with the first layer having sufficiently many neurons. Then, we can take the next layer having less neurons while the third layer has a very limited number of neurons. The fourth and fifth layer increase in size the same way it was decreased. If the third layer has M neurons and we can obtain good results, then this means that the solution space can somehow be parametrized using M parameters. This approach also gives insight into the possibilities of decreasing computation time by decreasing the size of the neural network. Using the non-branched neural network, we have obtained the following result.

M	1	2	3	4	5	6	7	8
error u	6.96	4.32	2.9e-04	4.2e-05	1.1e-04	6.5e-05	3.7e-05	2.4e-04
error σ	0.99	0.60	2.9e-05	1.8e-05	1.6e-05	2.7e-05	2.3e-05	2.4e-05

Table 5.3: Using the unweighted loss L_G with 5 hidden layers having number of neurons per layer (20,10,M,10,20). Here $\varepsilon = 10^{-2}$ and $N = 128$.

These results suggest we need at least three neurons in the middle layer, which in turn indicates that the solution space is three dimensional. For the branched neural network we have done a similar experiment, where the number of neurons in the middle layer is equal to M in each branch (Table 5.4).

M	1	2	3	4	5	6	7
error u	0.03	0.11	0.05	0.01	0.03	0.03	0.10
error σ	0.03	0.20	0.04	0.02	0.04	0.06	0.19

Table 5.4: Using the unweighted loss L_F with 5 hidden layers having number of neurons per layer (15,7,M,7,15) in both branches. Here $\varepsilon = 10^{-2}$ and $N = 64$.

Note that $M = 1$ seems enough, although the solutions are not as accurate as the previous experiment.

5.4 Training with ε as input

Additionally, we've implemented a neural network for the FOSLS functional having inputs x as well as ε. In this way, a solution can be quickly computed for any value of ε (within a certain range) by a pre-trained network. As seen in Table 5.5, this resulted in relatively inaccurate approximations. This makes sense because training a network for a very specific task should logically provide better results than training a network for a range of tasks. Here the interval for ε is $[10^{-3}, 10^{-5}]$. Using this method, usually the boundary layer is too wide and this width stays approximately constant for different values of ε.

# col. points	16	32	64	128
error u, $\varepsilon = 10^{-3}$	0.93	29.2	29.0	1.03
error σ, $\varepsilon = 10^{-3}$	3.54	2.92	3.61	0.86
error u, $\varepsilon = 10^{-5}$	1.34	31.4	31.2	1.36
error σ, $\varepsilon = 10^{-5}$	3.16	2.31	3.80	1.07

Table 5.5: Using a branched neural network with inputs x and ε for various number of collocation points within the ε-domain. The number of points for x is always 64.

CHAPTER 6

Conclusion and outlook

In this paper we have compared four methods involving PINNs to compute solutions to the 1D advection-diffusion equation. These methods involve: the ordinary PINN functional L_G, weighted and unweighted and the FOSLS functional L_F, weighted and unweighted.

We have seen that the loss weight $\hat{\lambda}$ that is optimal according to [8] did in fact not produce accurate results when we've applied this to L_G, which was surprising because it was derived for this loss function. It also did not perform well when it is applied to L_F, which is less surprising. We did find that, using L_G, a slightly amplified loss weight ($50\hat{\lambda}$) produced much better results than $\hat{\lambda}$ for high diffusion, but for low diffusion, $\hat{\lambda}$ with L_G performed the best of all methods tested. Using L_G with no weights as well as $50\hat{\lambda}$, we obtained stable solutions for very large diffusion ($\varepsilon = 10^{-5}$), although the boundary layer of the computed solution was still too thick. The same holds for the method with L_F with no weights ($\lambda = 1/2$). Any other weights applied to L_F further destabilized the method.

A lower weight than $\hat{\lambda}$ proved to be not useful at all, for both L_G and L_F. Furthermore, it seemed disadvantageous to include too many collocation points, due to convergence problems of the optimizer. Intuitively, more collocation points should provide more accurate results. This is probably true if the algorithm converges to a stable solution, but what we obtained is that often the output is a straight line. Thus, more research could be done on how the algorithm can be made to converge with a large number of collocation points. This may be possible with better loss weights than the ones suggested in [8]. Both L_G and L_F have performed equally well overall. Moreover, both functionals had problems approximating the derivative of the solution. The computed derivative was often the zero function, even though the solution was non-constant. Only when the solution was very accurate, was the derivative nonzero. This immediately made the derivative also accurate. To solve this problem in the future, the FOSLS functional is useful because we can specifically give a weight to the relation between the solution and its derivative.

6.1 Further Research

In this paper a very limited setting is used to test various PINN methods. Thus our implementation of the neural network is not reusable for different problems. For example, the Python code must be substantially modified to implement the convection diffusion equation in higher dimensions. It is interesting to see if we can obtain a similar performance in this new setting. In two dimensions, one would need to implement a grid of collocation points in the (x, y) plane instead of an interval, as well as many more points on the boundary of the domain instead of just two.

Secondly, when using a neural network with inputs x and ε for the 1D problem, implementing the FOSLS functional, the following extension can be made. We have seen that the optimal loss weight derived in [8] is not conducive to accurate solutions, but changing the loss weight does play a role in the accuracy. Perhaps a neural network can compute the value for λ that is optimal for the given problem. To this end, we cannot use the same network to do this, as the loss function itself would be changing every iteration of training and this becomes too complicated for the optimizer. Instead, an additional neural network could be employed to first compute λ. Then this value is passed to a second network that computes u and σ and uses the weighted loss function. This loss is then used to optimize the parameters of this second network. Finally, the parameters of the first network are readjusted using the accuracy of u and σ.

APPENDIX A

Detailed Results

ε	Unweighted, L_G, error σ				
	64	128	256	512	1024
1e-5	1	1	1	1	1
1e-4	1	1.04	1	1	1
1e-3	1.7	1.32	1	1	1
1e-2	0.996	9.07e-05	2.75e-06	5.12e-06	2.16e-06
1e-1	5.32e-05	1.98e-05	5.13e-05	5.77e-05	1.14e-04

Table A.1: Relative L_2 error for σ using loss L_G with $\lambda = 1/2$ for various numbers of collocation points.

ε	Optimal weight, L_G, error u				
	64	128	256	512	1024
1e-5	18.2	18.2	18.2	18.2	18.2
1e-4	1.92	18.2	18.2	18.2	18.2
1e-3	1.69e+01	1.69e+01	1.69e+01	1.69e+01	1.69e+01
1e-2	2.70e-03	7.02e+00	5.98e-04	7.97e-04	8.30e-05
1e-1	3.07e-06	6.85e-06	2.04e-06	6.94e-07	1.19e-05

Table A.2: Relative L_2 error for u using loss L_G with $\lambda = \hat{\lambda}$ for various numbers of collocation points.

ε	Optimal weight, L_G, error σ				
	64	128	256	512	1024
1e-5	1	1	1	1	1
1e-4	1	1	1	1	1
1e-3	1	1	1	1	1
1e-2	1.48e-03	9.16e-01	2.21e-04	2.29e-04	1.99e-05
1e-1	5.13e-06	8.90e-06	4.12e-06	1.25e-06	1.08e-05

Table A.3: Relative L_2 error for σ using loss L_G with $\lambda = \hat{\lambda}$ for various numbers of collocation points.

ε	Supra-optimal weight, L_G, error u				
	64	128	256	512	1024
1e-5	2.04	0.958	18.2	18.2	18.2
1e-4	1.95	18.2	0.645	18.2	18.2
1e-3	2.93e+01	2.93e+01	1.65e+01	1.65e+01	1.65e+01
1e-2	7.10e+00	2.57e-05	2.64e-05	3.40e-05	1.42e-05
1e-1	2.30e+152	2.13e+152	2.93e+148	9.17e+146	3.04e+144

Table A.4: Relative L_2 error for u using loss L_G with $\lambda = 50\hat{\lambda}$ for various numbers of collocation points.

ε	Supra-optimal weight, L_G, error σ				
	64	128	256	512	1024
1e-5	1	1	1	1	1
1e-4	1	1	1.02	1	1
1e-3	1.08	1.25	0.999	0.999	0.999
1e-2	0.797	4.03e-05	1.09e-05	1.37e-05	6.17e-06
1e-1	1	1	1	1	1

Table A.5: Relative L_2 error for σ using loss L_G with $\lambda = 50\hat{\lambda}$ for various numbers of collocation points.

	Infra-optimal weight, L_G, error u				
ε	64	128	256	512	1024
1e-5	18.7	20.1	19.7	20	20.7
1e-4	18.2	18.2	18.2	18.2	18.2
1e-3	16.9	16.9	16.9	16.9	16.9
1e-2	13.4	7.51	7.45	5.92	1.96e-02
1e-1	2.13e-05	2.50e-05	1.13e-04	1.53e-05	1.83e-05

Table A.6: Relative L_2 error for u using loss L_G with $\lambda = \hat{\lambda}/50$ for various numbers of collocation points.

	Infra-optimal weight, L_G, error σ				
ε	64	128	256	512	1024
1e-5	1	1	1	1	1
1e-4	1	1	1	1	1
1e-3	0.999	0.999	0.999	0.999	0.999
1e-2	3.05	0.986	0.977	0.77	4.18e-03
1e-1	2.21e-05	3.92e-05	1.23e-04	2.76e-05	2.36e-05

Table A.7: Relative L_2 error for σ using loss L_G with $\lambda = \hat{\lambda}/50$ for various numbers of collocation points.

	Unweighted, L_F, error u				
ε	64	128	256	512	1024
1e-5	1.26	16.1	16.1	16.1	16.1
1e-4	2.57	16	0.829	16.1	16.1
1e-3	1.26	0.379	13.5	15	14.7
1e-2	1.98e-03	6.60e-03	6.92e-04	2.05e-03	6.76e-03
1e-1	1.39e-05	1.14e-05	2.41e-05	1.04e-05	4.95e-06

Table A.8: Relative L_2 error for u using loss L_F with $\lambda = 1/2$ for various numbers of collocation points.

	Unweighted, L_F, error σ				
ε	64	128	256	512	1024
1e-5	1	1	1	1	1
1e-4	0.997	0.987	1.08	0.997	0.998
1e-3	1.31	1	0.863	0.974	0.959
1e-2	2.66e-02	7.73e-03	7.01e-04	1.78e-03	5.62e-03
1e-1	1.39e-05	9.28e-06	2.19e-05	1.07e-05	5.65e-06

Table A.9: Relative L_2 error for σ using loss L_F with $\lambda = 1/2$ for various numbers of collocation points.

ε	Optimal weight, L_F, error u				
	64	128	256	512	1024
1e-5	18.2	18.2	18.2	18.2	18.2
1e-4	18.2	18.1	18.2	18.2	18.2
1e-3	17.4	16	16.4	10	16.6
1e-2	1.01e-02	1.20e-02	1.49e-03	2.87e-03	1.11e-02
1e-1	8.01e-05	5.59e-05	1.10e-04	2.69e-05	4.08e-05

Table A.10: Relative L_2 error for u using loss L_F with $\lambda = \hat{\lambda}$ for various numbers of collocation points.

ε	Optimal weight, L_F, error σ				
	64	128	256	512	1024
1e-5	1	1	1	1	1
1e-4	0.996	0.994	0.996	0.996	0.996
1e-3	1.12	0.955	0.956	0.662	0.977
1e-2	1.30e-02	1.38e-02	1.46e-03	1.61e-03	1.58e-03
1e-1	6.11e-05	4.15e-05	8.88e-05	2.95e-05	4.32e-05

Table A.11: Relative L_2 error for σ using loss L_F with $\lambda = \hat{\lambda}$ for various numbers of collocation points.

ε	Supra-optimal weight, L_F, error σ				
	64	128	256	512	1024
1e-5	1	0.999	0.999	0.999	0.999
1e-4	0.991	0.787	0.983	0.992	0.994
1e-3	0.718	0.813	0.414	0.203	0.918
1e-2	1.35e-02	2.00e-03	3.82e-04	5.99e-04	7.94e-04
1e-1	1.42e+151	inf	1.08e+151	3.24e+143	1.31e+151

Table A.12: Relative L_2 error for σ using loss L_F with $\lambda = 50\hat{\lambda}$ for various numbers of collocation points.

	Infra-optimal weight, L_F, error u				
ε	64	128	256	512	1024
1e-5	19.4	18.5	19.2	19.4	18.8
1e-4	18.2	18.2	18.2	18.2	18.2
1e-3	16.7	16.8	16.6	16.8	16.5
1e-2	1.89e-02	9.80e-02	7.09e-03	3.13e-02	1.78e-02
1e-1	2.75e-04	4.46e-04	3.93e-04	1.52e-04	1.14e-04

Table A.13: Relative L_2 error for u using loss L_F with $\lambda = \hat{\lambda}/50$ for various numbers of collocation points.

	Infra-optimal weight, L_F, error σ				
ε	64	128	256	512	1024
1e-5	1	1	1	1	1
1e-4	0.998	0.998	0.998	0.998	0.998
1e-3	0.997	0.989	0.979	0.99	0.966
1e-2	2.42e-02	5.73e-02	6.27e-03	2.32e-02	1.30e-02
1e-1	2.84e-04	2.76e-04	3.07e-04	1.45e-04	1.73e-04

Table A.14: Relative L_2 error for σ using loss L_F with $\lambda = \hat{\lambda}/50$ for various numbers of collocation points.

Bibliography

[1] Cai, Z. et al. 'First-Order System Least Squares for Second-Order Partial Differential Equations: Part I'. In: *SIAM Journal on Numerical Analysis* vol. 31, no. 6 (1994), pp. 1785–1799.

[2] Cai, Z. et al. 'Deep least-squares methods: An unsupervised learning-based numerical method for solving elliptic PDEs'. In: *Journal of Computational Physics* vol. 420 (2020), p. 109707.

[3] Choromanska, A. et al. 'The Loss Surfaces of Multilayer Networks'. In: *arXiv e-prints*, arXiv:1412.0233 (Nov. 2014), arXiv:1412.0233. arXiv: 1412.0233 [cs.LG].

[4] Lu, L. et al. 'DeepXDE: A deep learning library for solving differential equations'. In: *arXiv preprint arXiv:1907.04502* (2019).

[5] Nielsen, M. A. *Neural Networks and Deep Learning*. Determination Press, 2015.

[6] Raissi, M., Perdikaris, P. and Karniadakis, G. E. 'Numerical Gaussian processes for time-dependent and non-linear partial differential equations'. In: *SIAM Journal of Scientific Computing* vol. 40 (2018), A172–A198.

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

[8] van der Meer, R., Oosterlee, C. and Borovykh, A. 'Optimally weighted loss functions for solving PDEs with Neural Networks'. In: *arXiv e-prints*, arXiv:2002.06269 (Feb. 2020), arXiv:2002.06269. arXiv: 2002.06269 [math.NA].

[9] Vuik, C. et al. *Numerical methods for ordinary differential equations*. Delft Academic Press, 2018.