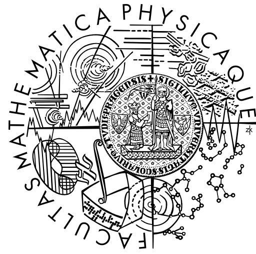

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Matěj Vais

Deep learning for the solution of differential equations

Department of Numerical Mathematics

Supervisor of the bachelor thesis: Scott Congreve, Ph.D.
Study programme: Mathematical Modelling
Study branch: MMOP

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my supervisor, Dr. Scott Congreve, for his patient guidance.

Title: Deep learning for the solution of differential equations

Author: Matěj Vais

Department: Department of Numerical Mathematics

Supervisor: Scott Congreve, Ph.D., Department of Numerical Mathematics

Abstract: Neural networks are becoming an ever more prominent method in the field of differential equations. Their use is embodied in the concept of physics-informed neural network (PINN), which combines a traditional deep neural network with the underlying laws of physics described by PDEs. We investigate the abilities of this relatively novel approach on thee diverse examples in order to give a good overview of its advantages and issues. Every problem is also solved via the finite element method, which serves as a reference. In addition to that, we propose the usage of pre-training, which is already present in other scientific areas. If we initialize the process of solving of one equation with a solution to a similar problem, in some settings, we are able to significantly reduce computation time, which is a major drawback of PINN.

Keywords: machine learning, deep learning, differential equations, finite element method, physics-informed neural network

Contents

Introduction **2**

1 Methods and Concepts **3**
1.1 Feed-Forward Neural Network . . . 3
1.2 Physics-Informed Neural Network . . . 4
1.3 PINN Optimizers – Gradient Descent . . . 6
1.4 Automatic Differentiation . . . 7
1.5 Finite Element Method . . . 7
1.5.1 Weak Formulation . . . 8
1.5.2 Choice of Mesh and Discretization . . . 9
1.5.3 Choice of Spaces and Summary . . . 10

2 Examples **12**
2.1 Simple Example . . . 12
2.1.1 Physics-Informed Neural Network . . . 12
2.1.2 Finite Element Method . . . 13
2.2 Nonlinear Problem . . . 13
2.2.1 Physics-Informed Neural Network . . . 14
2.2.2 Finite Element Method and Newton Iteration . . . 15
2.2.3 Pre-training . . . 18
2.3 Singularly Perturbed Problem . . . 18
2.3.1 Physics-Informed Neural Network . . . 18
2.3.2 Finite Element Method . . . 19

3 Numerical Results **21**
3.1 Simple Example . . . 21
3.2 Nonlinear Problem . . . 22
3.3 Singularly Perturbed Problem . . . 24
3.4 Comparison Between PINN and FEM . . . 26

Conclusion **27**

Bibliography **28**

List of Figures **29**

List of Tables **30**

List of Abbreviations **31**

A Attachments **32**
A.1 Detailed Results from Pre-training . . . 32

Introduction

Machine and deep learning has been on a constant rise in recent years. We have seen the the expansion of generative as well as discriminative models in all major scientific areas. Although text or image processing is arguably the most widespread application of neural networks, differential equations are becoming an increasingly popular field, where the deep learning approach shows rapid progress. The foundations were laid in 1998 (Lagaris et al. [1998]), when the concept of the so call physics-informed neural network, also known as PINN, was established. The traditional networks at that time could learn only from data points given to them. PINN on the other hand, also incorporated the underlying differential equations into the mix.

Unlike FEM or FDM, the physics-informed neural network does not require the creation of mesh inside the equation's domain nor discretization. PINN is based on minimization of a special function quantifying the approximating accuracy of the network in certain points chosen from the domain and the boundary. The network, specifically constructed for each equation, plays the role of the equation's solution. It is parameterized by a set of values that are optimized using gradient descent algorithms. All these concepts are conveniently bound together in the Python library DeepXDE (Lu et al. [2021]), which essentially made this thesis possible.

The following chapters will be focused on exploring the possibilities of this relatively novel approach. We will deal with three separate differential equations to get a better understanding of what are the drawbacks as well as the benefits of this method. At first, we will introduce a simple Poisson's equation to make the reader familiar with the setting, which will then be modified into its nonlinear analogue. The last task will be a singularly perturbed problem with a thin boundary layer exhibiting a steep gradient near the boundary. The singular perturbation will require from us use divide the equation's domain into two subdomains and solve an approximating equation on each part. As a reference approach to all the problems, we choose the finite element method.

In addition to the three problem, we will also try to improve the second one with an idea already in use in other parts fields. The focus will be on the so called pre-training, which consists of initializing the solution process of the original equation by a solution to a different one. The auxiliary equation is chosen, such that it is very similar to the original. It will be clear that this process can, in some cases, alleviate the long computation time, which plagues the the new method.

In chapter 1, we will familiarize the reader with the aforementioned concepts that are essential to understanding PINN. Chapter 2 will give a detailed description of all the examples, after which we include their results in chapter 3. At the end of the thesis, concluding remarks regarding the results will be made.

1. Methods and Concepts

In this chapter, we will give a brief introduction to some of the methods of machine and deep learning that are utilized when solving differential equations. We will explore the idea of feed-forward neural network (FNN) that can be developed into so called physics-informed neural network (PINN). This network employs many attractive concepts such as automatic differentiation (AD) or gradient descent, which will be detailed in the following subsections. At the end of this chapter, we will also give a short overview of the finite element method (FEM).

1.1 Feed-Forward Neural Network

Feed-forward neural network (FNN, also known as multilayer perceptron) is a type of neural network that can very easily approximate complex nonlinear data. It consists of an input layer, one or more hidden layers and an output layer.

From a strictly mathematical point of view, it is a parametric function $\mathcal{N}^L$ that is highly nonlinear with respect to its inputs. It maps a real d_{in}-dimensional input vector to a d_{out}-dimensional output vector.

$$\mathcal{N}^L(\boldsymbol{x}, \boldsymbol{\theta}) : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$$

Here, $\boldsymbol{\theta} = \{\boldsymbol{W}^1, \dots \boldsymbol{W}^L, \boldsymbol{b}^1, \dots, \boldsymbol{b}^L\}$ denotes the weights and biases parameterizing the neural network and L the number of hidden layers plus the output layer. Each layer l of size N_l, apart from the input, consist of four parts.

First, the *weight matrix* $\boldsymbol{W}^l \in \mathbb{R}^{N_l \times N_{l-1}}$ multiplied by the *output* of the previous layer $\mathcal{N}^{l-1} \in \mathbb{R}^{N_{l-1}}$. Each row in the weight matrix $\boldsymbol{W}^l$ corresponds to one neuron in the l-th layer. Next, the *bias vector* $\boldsymbol{b}^l \in \mathbb{R}^{N_l}$ is added to the previous result, which acts as a shift. Eventually, the linear part is wrapped into the so called *activation function* a^l, that introduces the nonlinearity into the scheme. This process can be iteratively applied multiple times to produce a deep neural network.

$$\begin{aligned}
\text{input layer:} \quad & \mathcal{N}^0(\boldsymbol{x}) = \boldsymbol{x} \in \mathbb{R}^{d_{in}} \\
\text{hidden layers:} \quad & \mathcal{N}^l(\boldsymbol{x}) = a^l(\boldsymbol{W}^l \mathcal{N}^{l-1}(\boldsymbol{x}) + \boldsymbol{b}^l) \in \mathbb{R}^{N_l}, \qquad l \in \{1, \dots, L-1\} \\
\text{output layer:} \quad & \mathcal{N}^L(\boldsymbol{x}) = a^L(\boldsymbol{W}^L \mathcal{N}^{L-1}(\boldsymbol{x}) + \boldsymbol{b}^l) \in \mathbb{R}^{d_{out}}
\end{aligned}$$

The aforementioned function composition is efficiently visualized in the scheme 1.1, where each neuron from one layer contributes to the next one via the weights, biases and subsequent activation to produce a complicated result.

It is a known fact that a neural network with only one hidden layer can approximate any continuous function on a closed interval. Although this is a very convenient result, the size of the layer would have to be enormous for complex problems. Instead, it is much better to define multiple hidden layers of various sizes that are chosen in accordance with the problem's complexity. A simple task might only need one layer, whereas a complicated one may require ten. Unfortunately, there are no known ways to compute the size, which has to be determined by trial and error or by looking at similar problem.

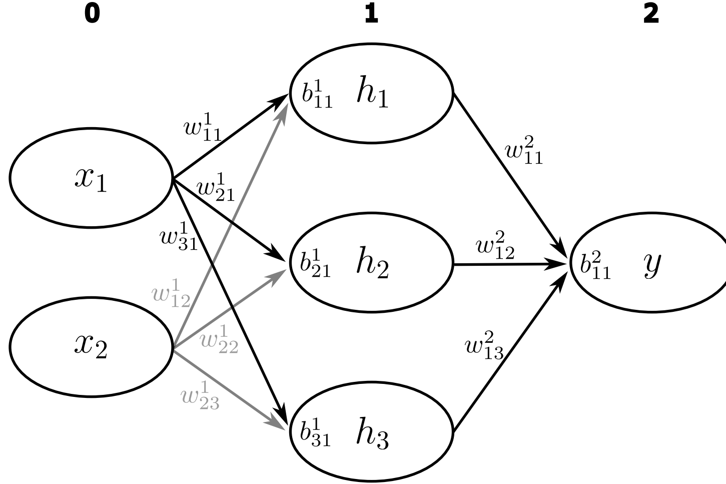

Figure 1.1: A scheme of a FNN consisting of a two-dimensional input layer, one three-dimensional hidden layer and a one-dimensional output layer. The weight matrix $\boldsymbol{W}^1 = \{w^1_{ij}\} \in \mathbb{R}^{3\times 2}$ has one row for each element of the hidden layer. Thee dimensional bias vector $\boldsymbol{b}^1$ alos has one element for each neuron/element in the hidden layer. Similarly $\boldsymbol{W}^1 \in \mathbb{R}^{3\times 1}$ and $\boldsymbol{b}^1 \in \mathbb{R}$.

Activations a^l are nonlinear function modifying the output and can be, in theory, chosen differently for each layer. However, we usually choose one activation for all the hidden layers and another for the output layer. The most frequent options are ReLU 1.1, hyperbolic tangent 1.2, the sigmoid function 1.3 and, of course, the identity.

$$\mathrm{ReLU}(x) = \max\{0, x\} \tag{1.1}$$

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{1.2}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.3}$$

$$\mathrm{softmax}(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_{1\leq j\leq \dim(\boldsymbol{x})} e^{\boldsymbol{x}_j}} \tag{1.4}$$

For inputs of higher dimensions, i.e. $\dim(\boldsymbol{x}) \geq 2$, the activations are applied component-wise, one exception being the sigmoid which is generalized into the softmax function 1.4 ([Goodfellow et al., 2016, page 181]).

1.2 Physics-Informed Neural Network

In the section 1.1 above, we have seen a general setting of FNN but haven't been concerned with the optimality of the network. To produce accurate predictions, i.e. outputs $\boldsymbol{y} \in \mathbb{R}^{d_{out}}$ for a given input $\boldsymbol{x} \in \mathbb{R}^{d_{in}}$, the network must undergo so called *training*, where it gradually learns to predict on the *training set.* Its approximation capacity is then tested on the *test set.* To quantify the discrepancy between the desired output (also known as *target*), we employ the *loss function*, which is minimized during training.

To continue further, we must introduce the setting. Throughout this thesis, we will be concerned with differential equations in a very general form. Setting $d = d_{in}$, let us define $\boldsymbol{x} = (x_1, \cdots, x_d) \in \mathbb{R}^d$ as a point in the domain of the equation

$$f\left(\boldsymbol{x}; u, \frac{\partial u}{\partial x_1}; \ldots; \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1^2}; \ldots; \frac{\partial^2 u}{\partial x_d^2}; \ldots\right) = 0, \quad \boldsymbol{x} \in \Omega, \tag{1.5}$$

with an appropriate boundary condition

$$\mathcal{B}\left(\boldsymbol{x}, u, \frac{\partial u}{\partial x_1}; \ldots; \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1^2}; \ldots; \frac{\partial^2 u}{\partial x_d^2}; \ldots\right) = 0, \quad \boldsymbol{x} \in \partial\Omega. \tag{1.6}$$

Two finite sets of training points are then chosen from the domain Ω and $\partial\Omega$ respectively. The first set $\mathcal{T}_f \subset \Omega$ allows the neural network to approximate the solution of 1.5 in the domain, whereas the second set $\mathcal{T}_b \subset \partial\Omega$ allows it to capture the boundary condition 1.6. Their size is usually tens or at most hundreds of points for a one-dimensional problem, squared that for a two-dimensional ($\sim 10^4$) and cubed for a three-dimensional ($\sim 10^6$). The points are mostly sampled randomly, but an equidistant division also works in most cases. Moreover, this can be aided by adaptive sampling that happens during the training (Lu et al. [2021]). Similarly, two sets of testing points are chosen, on which the approximating capability of $\mathcal{N}^L$ is tested.

As originally proposed in Lagaris et al. [1998] and further developed in Lu et al. [2021], we define a FNN $\hat{u}(\boldsymbol{x}; \boldsymbol{\theta})$ parameterized by its weights and biases that will represent the solution to the equation 1.5 with the boundary condition 1.6. The loss function $\mathcal{L}$ of this network takes the form of a weighted sum

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b), \tag{1.7}$$

where w_f and w_b are the weights and $\mathcal{L}_f$ and $\mathcal{L}_b$ are sub-parts of the loss function corresponding to the both training sets. They appear as sum of L^2-norms of the equation operator, and the boundary condition respectively, evaluated at each point of the set.

$$\mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{\boldsymbol{x} \in \mathcal{T}_f} \left\| f\left(\boldsymbol{x}; \hat{u}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1^2}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_d^2}; \ldots\right) \right\|_2^2$$

$$\mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{\boldsymbol{x} \in \mathcal{T}_b} \left\| \mathcal{B}\left(\boldsymbol{x}; \hat{u}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1^2}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_d^2}; \ldots\right) \right\|_2^2$$

Small values of $\mathcal{L}$ on both the training set and the test indicates an accurate solution. Sometimes, a high value on the test occurs, which together a with small value on the training set signals *over-training*, meaning that the neural network approximates the training set too well and shows considerably worse performance elsewhere. This displays the necessity to exploit the test set, which is not employed during the loss function minimization, and is used solely as a check.

We call this modified FNN *physics-informed neural network*, because, apart from the training set with the target values, it also incorporates the governing

equations into the mix. Without them, the network would work as simple function approximator learning only from data and not utilizing the underlying laws of physics. For that, the mean square error MSE can be employed instead of 1.7.

$$\mathrm{MSE}(\boldsymbol{\theta}, \mathcal{T}) = \frac{1}{2|\mathcal{T}|} \sum_{\boldsymbol{x} \in \mathcal{T}} \left(\hat{u}(\boldsymbol{x}; \boldsymbol{\theta}) - t_{\boldsymbol{x}}\right)^2 ,$$

where $\mathcal{N}^L(\boldsymbol{x}, \boldsymbol{\theta})$ is a prediction for a given data point and $t_{\boldsymbol{x}}$ its target value.

It is worth noting that the treatment of the equation doesn't depend much on its difficulty and is the same for both linear and nonlinear case. Also, multiple dimensions don't scale the problem as much (Lu et al. [2021]) as with the finite element method, the only increase being the in the number of training points, which will be visible in the following chapters.

1.3 PINN Optimizers – Gradient Descent

During training, we search the space of parameters for an optimal vector $\boldsymbol{\theta} \in \mathbb{R}^n$ which minimizes the loss function $\mathcal{L}$. Here, n is the dimension of the parameter space (one dimension for each scalar in each weight matrices and bias vectors). This search is usually done using gradient descent methods. We start with some initial estimation of $\boldsymbol{\theta}$ and employing the so called *learning rate* α update the original estimate as

$$\boldsymbol{\theta}_{i+1} \leftarrow \boldsymbol{\theta}_i - \alpha \nabla \mathcal{L}(\boldsymbol{\theta}; \mathcal{T})$$

to obtain the next approximation. This simple process is then repeated until we reach the minimum with desired accuracy.

The **Adam** optimizer (Kingma and Ba [2017]) refines this simple approach in multiple ways. It uses a vector of learning rates rather than a constant, one element for each parameter. This way, it can adaptively adjust the step length in all directions. Nonetheless, the constant α must be provided as an upper bound for the step. Secondly, it includes a history of previous updated into the evaluation of the current step. The weight of the history of updates can be tweaked by two constants β_1 and β_2. However, in most cases, they require very little tuning. The recommended values are $\beta_1 = 0.9, \beta_2 = 0.999$. Finally, among some other minor benefit, the algorithm is invariant to rescaling of $\nabla \boldsymbol{\theta}$.

The **L-BFGS** (Byrd et al. [1995]) is a quasi-Newton method that, as opposed to Adam, also employs the second derivative. We first compute the direction of the steepest descent, in which we substitute the minimized function $\mathcal{L}$ by a quadratic model. The Cauchy point of this approximation is then computed as the next approximate value of the $\mathcal{L}$'s minimizer. This approach is based on the BFGS algorithm, but has only linear memory requirements $\mathcal{O}(n)$ for the inverse of the Hesse matrix $\boldsymbol{B}$. The inverse is stored in a decomposed form represented by a set of m vectors that are given by the update history of $\boldsymbol{\theta}$ and $\nabla \mathcal{L}$. The reduced demand for memory broadens its field of applications to neural networks with large n, because the original algorithm requires us to store a dense estimate of $\boldsymbol{B}$, which is $\mathcal{O}(n^2)$.

Both optimizers offer advantages. Adam is typically more efficient than L-BFGS in terms of computation time. The latter approach however provides more accurate result in fewer iterations as can be seen in chapter 2.

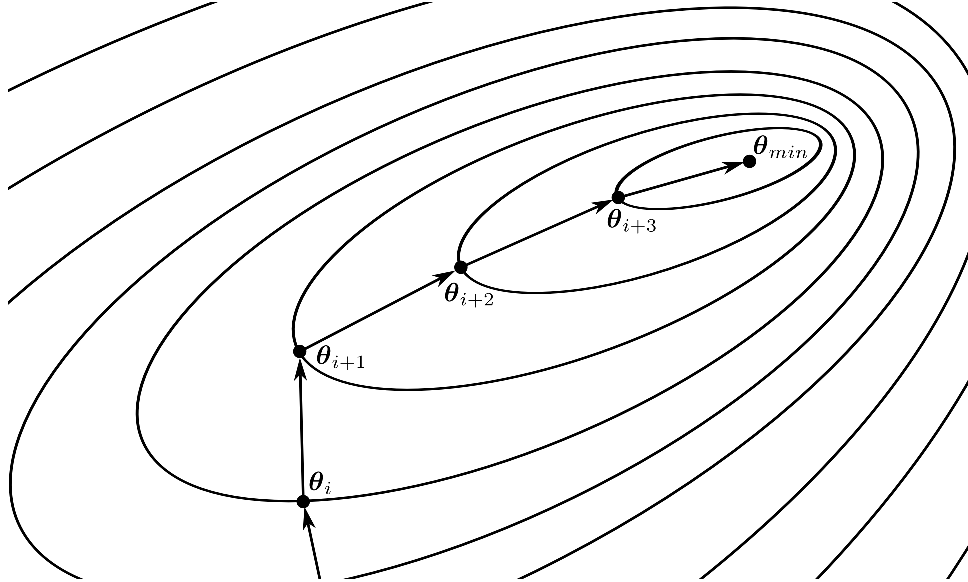

Figure 1.2: Evolution of parameters $\boldsymbol{\theta}$ in the space $\mathbb{R}^n$. We move perpendicular to the contour lines of the loss function $\mathcal{L}$ until we reach a minimum within the required tolerance.

1.4 Automatic Differentiation

In the previous section, we have seen the need for an efficient computation of network's derivatives. Traditional numerical differentiation is not very well suited for such a task, since it is computationally expensive and often introduces large errors. Neither symbolic differentiation, which results in very large expressions, is optimal. On the other hand, automatic differentiation (AD), which is already embedded in the TensorFlow package (backend of the DeepXDE library), offers an applicable way of computing gradients for deep neural networks that have a large input $\boldsymbol{x} \in \mathbb{R}^{d_{in}}$ and a small output $\boldsymbol{y} \in \mathbb{R}^{d_{out}}$.

AD relies mainly on the multivariable chain rule and other basic properties of derivatives to construct an computational graph, which subdivides the evaluation into primitive operations. One pass in the direction opposite to the construction grants us the whole Jacobian matrix, i.e. derivatives of the output with respect to all the inputs (Lu et al. [2021]). This approach in know as backpropagation or reverse mode AD and gives us the function value and its gradient in constant time. An example showing the computation for FNN with one hidden layer is given in Lagaris et al. [1998]. Forward mode also exists, its use is, however, mostly limited to generative applications that have a low-dimensional input and a high-dimensional output.

1.5 Finite Element Method

Finite element method (FEM) is a numerical method traditionally used for solving ordinary and partial differential equation in one, two or three spacial dimensions. First, the domain of the equation is divided into subdomains of polygonal shape, the finite elements. Then, the solution is found as a polynomial interpolation separately on each element, which produces the discretization. This allows us to

solve equations where the analytical solution is inaccessible or impractical to use.

This approach, which will serve as a benchmark method for PINN, will be illustrated on one dimensional Poisson's equation with zero Dirichlet boundary conditions. We chose this equation due to its simplicity and close resemblance to other problems detailed in the Examples chapter. However, a straightforward generalization to more dimensions is possible, even including zero Neumann condition. The method also offers ways to tackle non-zero values on the boundary, but this will not be pursued in this thesis.

Let us define

$$-u''(x) = f(x) \quad \forall x \in \Omega, \tag{1.8}$$

$$u(x) = 0 \quad \forall x \in \partial\Omega, \tag{1.9}$$

where $\Omega \subset \mathbb{R}$, $u \in C^2(\Omega) \cap C(\overline{\Omega})$ and $f \in C(\overline{\Omega})$. The method consists of two main steps that also apply to any other equation other than Poisson's: the weak formulation and a subsequent discretization, which provides a piecewise polynomial solution.

1.5.1 Weak Formulation

Starting with the *weak formulation*, we multiply the the equation 1.8 by a test function $v \in V$ and integrate over the whole domain Ω. We obtain

$$-\int_\Omega u''(x)v(x)dx = \int_\Omega f(x)v(x)dx.$$

Next, we apply integration by parts to reduce the order of the derivative of u:

$$-\int_{\partial\Omega} u'(x)v(x)dx + \int_\Omega u'(x)v'(x)dx = \int_\Omega f(x)v(x)dx. \tag{1.10}$$

So far, we have not defined the space V to which the test functions v belong. Doing it now will help us eliminate the integral over $\partial\Omega$, because V will be designed as a space of functions that are zero on $\partial\Omega$.

$$V := H_0^1 = \left\{v : v \in L^2(\Omega), \quad v' \in L^2(\Omega), \quad v(x) = 0 \quad \forall x \in \partial\Omega\right\}.$$

The equation 1.10 therefore reduces to

$$\int_\Omega u'(x)v'(x)dx = \int_\Omega f(x)v(x)dx,$$

which is often described as an equality between a symmetric bilinear form a dependent on u and v and a functional F dependent on v

$$a(u, v) = \langle F, v\rangle, \quad u, v \in V. \tag{1.11}$$

1.11 is the final weak formulation that helped us reduce the assumptions imposed on u.

1.5.2 Choice of Mesh and Discretization

Since the space V is infinite-dimensional, we choose its subspace V_h to produce a *discretization*. For that, we need to divide the interval Ω into subintervals. The most straightforward approach is an equidistant division into N subintervals of length h with dividing points $\{x_i\}_{i=0}^N$, where x_0 and x_N are the left and right boundary points. In more dimensions, this step would be equivalent to creating a polygonal mesh inside Ω. Let us define $V_h \subset V$ as

$$V_h := \left\{ v_h \in H_0^1, \quad v_h|_{[x_i, x_{i+1}]} \in \mathcal{P}_1([x_i, x_{i+1}]), \quad i = 0, \dots, N-1 \right\}.$$

The space V_h is a space of piecewise linear functions, in which we will look for an approximate solution u_h to the weak formulation 1.11. It is possible to choose polynomials of higher order, but we will not pursue this option. Let us state the equation approximating 1.11 as

$$a(u_h, v_h) = \langle F, v_h \rangle, \quad u_h, v_h \in V_h.$$

To continue further, we must select a basis of V_h. Making the following choice that will greatly simplify the successive procedure.

$$V_h = \operatorname{span}\{\phi_j\}_{j=1}^N$$

$$\phi_j(x) = \begin{cases} \frac{x - x_{j-1}}{h} & \text{if } x \in [x_{j-1}, x_j], \\ \frac{x_{j+1} - x}{h} & \text{if } x \in (x_j, x_{j+1}], \\ 0 & \text{otherwise,} \end{cases}$$

$$\phi_j'(x) = \begin{cases} \frac{1}{h} & \text{if } x \in [x_{j-1}, x_j], \\ -\frac{1}{h} & \text{if } x \in (x_j, x_{j+1}], \\ 0 & \text{otherwise.} \end{cases}$$

Here, ϕ_j' expresses the derivative of ϕ_j. The function ϕ_j are often call *hat function* due to their characteristic shape. Apart from ϕ_0 and ϕ_N, they are zero everywhere except for two subintervals, where they produce a characteristic peak in the middle. The two outer hat functions are defined only on one interval with their peak on the boundary.

In this basis, the approximate solution can be expressed as a linear combination

$$u_h(x) = \sum_{j=1}^{N-1} u_h(x_j)\phi_j(x).$$

Functions ϕ_0 and ϕ_N are excluded due to zero boundary conditions. We note that the coefficient of the linear combination are exactly the function values at the points x_j. Substituting for u_h in 1.11 and exploiting the linearity of the form a, we obtain

$$\begin{aligned} a(u_h, v_h) &= \langle F, v_h \rangle \\ a\left(\sum_{i=1}^{N-1} u_h(x_i)\phi_i(x), v_h \right) &= \langle F, v_h \rangle \\ \sum_{i=1}^{N-1} u_h(x_i) a(\phi_i(x), v_h) &= \langle F, v_h \rangle \end{aligned}$$

Choosing $v_h := \phi_j$ and labeling $a_{ij} := a(\phi_i, \phi_j) = \int_\Omega \phi_i'(x)\phi_j'(x)dx$ and $f_j := \langle F, \phi_j \rangle = \int_\Omega f(x)\phi_j(x)dx$ yields

$$\sum_{i=1}^{N-1} u_h(x_i) a(\phi_i(x), \phi_j) = \langle F, \phi_j \rangle$$

$$\sum_{i=1}^{N-1} u_h(x_i) \int_\Omega \phi_i'(x)\phi_j'(x)dx = \int_\Omega f(x)\phi_j(x)dx,$$

$$\sum_{i=1}^{N-1} u_h(x_i) a_{ij} = f_j, \quad j = 1, \dots, N-1,$$

which can be represented in a matrix form as

$$\boldsymbol{A}\boldsymbol{u}_h = \boldsymbol{f}. \tag{1.12}$$

Here, $\boldsymbol{A} = \{a_{ij}\}_{i,j=1}^{N-1} \in \mathbb{R}^{(N-1)\times(N-1)}$ and $\boldsymbol{f} = \{f_j\}_{j=1}^{N-1} \in \mathbb{R}^{(N-1)}$. The boundary values $u(x_0)$ and $u(x_N)$ are, of course, equal to zero as a result of the imposed boundary conditions 1.9. The matrix 1.12 equation can be further simplified, since many of the elements of the matrix $\boldsymbol{A}$ are zero. Thanks to the compact support functions ϕ_i have, a straightforward computation leads us to

$$a_{ij} = \begin{cases} \frac{2}{h^2} & \text{if } i = j, \\ -\frac{1}{h^2} & \text{if } i = j+1 \text{ or } j = i+1, \\ 0 & \text{otherwise.} \end{cases}$$

Note that a concrete example of the evaluation of a_{ij} can be seen in subsection 2.2.2.

Finally, we present the matrix equation 1.12 explicitly as

$$\frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} u_h(x_1) \\ \\ \vdots \\ \\ u_h(x_{N-1}) \end{bmatrix} = \begin{bmatrix} f_1 \\ \\ \vdots \\ \\ f_{N-1} \end{bmatrix}, \tag{1.13}$$

which completes the whole process. We have successfully transformed the original differential equation into a set of $N-1$ linear algebraic equation, which can be efficiently solve, as it has a tridiagonal matrix.

1.5.3 Choice of Spaces and Summary

To summarize, a linear partial differential equation is solved using the FEM generally in the following way:

- integration by parts and subsequent weak formulation,
- choice of mesh subdividing the domain Ω,
- definition of the solution space V_h and choice of base (discretization),

- formulation of the algebraic system.

The choice of space V and its finite-dimensional counterpart V_h was done, such that we can guarantee H_0^1-boundedness and H_0^1-ellipticity of the left-hand side of the equation 1.11. This is valid thanks to the Hölder's and Poincaré's inequality and leads us to the existence and uniqueness of a solution for most functions f by applying the well known Lax-Milgram lemma.

2. Examples

In this section, we will illustrate PINN on three example. Starting with a simple Poisson's equation described in subsection 1.5, we will continue with a nonlinear modification of the same problem and see what changes are necessary in order to obtain an accurate solution. The section will be concluded by a linear problem with a singular perturbation that shows an interesting behaviour near the boundary.

All of the thee examples will also be solved via the finite element method. This and an analytical solutions will allow us to assess the deep learning approach.

2.1 Simple Example

This section will show try to show the application of the PINN on a simple equation to make the reader more familiar with the concept. For this purpose, we will use the one-dimensional Poisson's equation 1.8 with $f = 1$ and $\Omega = (0, 1)$. Let us define

$$-u''(x) = 1, \quad \forall x \in (0,1), \tag{2.1}$$

$$u(0) = u(1) = 0, \tag{2.2}$$

with an analytical solution $u(x) = -\frac{1}{2}(x^2 - x)$.

2.1.1 Physics-Informed Neural Network

We start with subtracting the right-hand side of 2.1, to produce an equation with zero right-hand size usable for training, which corresponds to the general differential equation 1.5. This is represented via the function `pde`. Next, we define an analytical solution `func` and an auxiliary function `boundary` involved in determining, whether a point is on the boundary or not.

```
import deepxde as dde
import numpy as np

def pde(x, y):
    dy_xx = dde.grad.hessian(y, x)
    return (- dy_xx - 1)

def boundary(x, on_boundary):
    return on_boundary

def sol(x):
    return (-0.5 * x**2 + 0.5 * x)
```

At this point, we are able to define the geometry of the problem including the boundary conditions 2.2, which corresponds to general boundary condition 1.6. Inside the interval, we chose 16 random points and 2 on the boundary,

i.e. $|\mathcal{T}_f| = 16$ and $|\mathcal{T}_b| = 2$. All the points, which were sampled from a uniform distribution, are then used in the evaluation of the losses $\mathcal{L}_f$ and $\mathcal{L}_b$. The geometry `geom` and boundary conditions `bc` are ultimately combined in a `data` object.

```
geom = dde.geometry.Interval(0, 1)
bc = dde.icbc.DirichletBC(geom, sol, boundary)
data = dde.data.PDE(
    geom, pde, bc, num_domain=16, num_boundary=2,
    solution=sol, num_test=100, train_distribution="uniform")
```

We further define the neural network object `net` corresponding to the neural network $\hat{u}$ with three hidden layers presented in section 1.1. Each one of the hidden layers consists of 50 neurons.

```
layer_size = [1] + [50] * 3 + [1]
activation = "tanh"
initializer = "Glorot uniform"
net = dde.nn.FNN(layer_size, activation, initializer)
```

Finally, we are able to combine the `data` and the `net` and create a `model` object which undergoes training. For this particular example, we chose the Adam optimizer, 10000 epochs and a learning rate of 0.001. The approximate values of the solution $\hat{u}(x)$ are obtained from the function `predict`. To compare of the result with the prescribed analytical solution, we chose a discrete ℓ^2 relative error, which is a standard metric used in the library.

```
model = dde.Model(data, net)
model.compile("adam", lr=0.001, metrics=["l2 relative error"])
losshistory, train_state = model.train(iterations=10000)
x = geom.uniform_points(1000, True)
u = model.predict(x)
```

2.1.2 Finite Element Method

To produce a finite element approximation of the solution $u(x)$, we will utilize the already derived matrix equation 1.13.

2.2 Nonlinear Problem

The problem we will tackle in this section is an one dimensional ordinary differential equation with a zero Dirichlet boundary condition very similar to the previous Poisson's equation.

$$-\frac{d}{dx}\left(\mu\left(\left|\frac{du}{dx}\right|\right)\frac{du}{dx}\right) = f \quad \text{in } (0,1), \tag{2.3}$$

$$u(0) = u(1) = 0 \tag{2.4}$$

$$\mu(x) = 1 + \frac{1}{1+x^2}. \tag{2.5}$$

Since the function μ is between 1 and 2, it introduces only a weak nonlinearity into the equation. The function f is chosen such that the solution u satisfies

$$u(x) = \sin(\pi x),$$

namely

$$f(x) = \frac{\pi^2 \sin(\pi x)\left(\pi^4 \cos^4(\pi x) + \pi^2 \cos^2(\pi x) + 2\right)}{\left(\pi^2 \cos^2(\pi x) + 1\right)^2}. \tag{2.6}$$

2.2.1 Physics-Informed Neural Network

Handling of this nonlinear problem via a neural network is very similar to the previous example. In fact, it doesn't require any major modification, even thought the problem is not linear. As we will see in the following subsection, this is an advantage, because the finite element method cannot be applied directly and needs significant adjustments.

The derivative present in the argument of the function μ in 2.3 is introduce as an supplementary function `dy`, which is then used very naturally in the definition of the function `nonlinear` corresponding to the nonlinear equation.

```
def dy(x, y):
    return dde.grad.jacobian(y, x)

def nonlinear(x, y):
    dy_x = dy(x, y)
    operator = dde.grad.jacobian(
        ( 1 + 1 / ( 1 + tf.abs( dy_x )**2 ) ) * dy_x, x)
    f = np.pi**2 * tf.sin(np.pi*x)
        * (2 + (np.pi* tf.cos(np.pi*x))**2
            + (np.pi * tf.cos(np.pi*x))**4)
        / (1 + (np.pi* tf.cos(np.pi*x))**2)**2
    return (- operator - f)
```

The second major difference is in the use of L-BFGS optimizer together with 64-bit floats. This change allows for higher accuracy of the approximation.

```
dde.config.real.set_float64()

model.compile(optimizer="L-BFGS", metrics = ["l2 relative error"])
```

2.2.2 Finite Element Method and Newton Iteration

The traditional numerical approach to this task is to employ the finite element method. In general, we will follow the steps described in section 1.5, i.e. integration by parts, weak formulation and discretization. However, the integral equation will first have to be linearized and then solved via the Newton's method.

After employing the procedure described in subsection 1.5.1, we obtain the weak formulation

$$\int_0^1 \mu\left(\left|\frac{du}{dx}\right|\right)\frac{du}{dx}\frac{dv}{dx}dx = \int_0^1 fvdx, \quad u, v \in V,$$

of the equation 2.3, where V is defined again as

$$V := H_0^1 = \left\{v : v \in L^2((0,1)), \quad v' \in L^2((0,1)), \quad v(0) = v(1) = 0\right\}.$$

Then, by choosing an equidistant division $\{x_i\}_{i=0}^N$ of the interval $[0,1]$, we define the approximating problem

$$\int_0^1 \mu\left(\left|\frac{du_h}{dx}\right|\right)\frac{du_h}{dx}\frac{dv_h}{dx}dx = \int_0^1 fv_h dx, \quad u_h, v_h \in V_h.$$

The space V_h is a space of piecewise linear functions is defined analogously to the space V as

$$V_h := \left\{v_h \in H_0^1, \quad v_h|_{[x_i,x_{i+1}]} \in \mathcal{P}_1([x_i, x_{i+1}]), \quad i = 0, \dots, N-1\right\}.$$

This weak formulation unfortunately cannot be solved directly, linearization is therefore needed. Substituting $u = u_h$ and $v = v_h$, we define the form a as

$$a(u,v) = \int_0^1 \mu\left(\left|\frac{du}{dx}\right|\right)\frac{du}{dx}\frac{dv}{dx}dx \tag{2.7}$$

for $u, v \in V_h$. Next, we introduce a parameter $\phi \in V_h$ and compute the Gateaux derivative of a with respect to the that parameter in the direction u as

$$\begin{aligned}
a'(\phi; u, v) &= \lim_{t\to 0}\frac{1}{t}\left\{a(\phi + tu; \phi + tu, v) - a(\phi; \phi, v)\right\} \\
&= \lim_{t\to 0}\frac{1}{t}\int_0^1 \mu\left(\left|\frac{d\phi}{dx} + t\frac{du}{dx}\right|\right)\left(\frac{d\phi}{dx} + t\frac{du}{dx}\right)\frac{dv}{dx}dx \\
&\quad - \lim_{t\to 0}\frac{1}{t}\int_0^1 t\mu\left(\left|\frac{d\phi}{dx}\right|\right)\frac{d\phi}{dx}\frac{dv}{dx}dx \\
&= \lim_{t\to 0}\frac{1}{t}\int_0^1 \left\{\mu\left(\left|\frac{d\phi}{dx} + t\frac{du}{dx}\right|\right) - \mu\left(\left|\frac{d\phi}{dx}\right|\right)\right\}\frac{d\phi}{dx}\frac{dv}{dx}dx \\
&\quad + \lim_{t\to 0}\frac{1}{t}\int_0^1 t\mu\left(\left|\frac{d\phi}{dx} + t\frac{du}{dx}\right|\right)\frac{du}{dx}\frac{dv}{dx}dx \qquad (2.8) \\
&= \int_0^1 \mu'\left(\left|\frac{d\phi}{dx}\right|\right)\left|\frac{d\phi}{dx}\right|^{-1}\frac{du}{dx}\left(\frac{d\phi}{dx}\right)^2\frac{dv}{dx}dx \\
&\quad + \int_0^1 \mu\left(\left|\frac{d\phi}{dx}\right|\right)\frac{du}{dx}\frac{dv}{dx}dx. \qquad (2.9)
\end{aligned}$$

To get from 2.8 to 2.9, we had to use the following trick:

$$\begin{aligned}\lim_{t\to 0}\left\{\mu\left(\left|\frac{d\phi}{dx}+t\frac{du}{dx}\right|\right)-\mu\left(\left|\frac{d\phi}{dx}\right|\right)\right\} &= \frac{d}{dt}\left\{\mu\left(\left|\frac{d\phi}{dx}+t\frac{du}{dx}\right|\right)\right\}\Bigg|_{t=0}\\ &= \mu'\left(\left|\frac{d\phi}{dx}+t\frac{du}{dx}\right|\right)\cdot\frac{d}{dt}\left(\left|\frac{d\phi}{dx}+t\frac{du}{dx}\right|\right)\Bigg|_{t=0}\\ &= \mu'\left(\left|\frac{d\phi}{dx}+t\frac{du}{dx}\right|\right)\cdot\frac{\frac{d\phi}{dx}+t\frac{du}{dx}}{\left|\frac{d\phi}{dx}+t\frac{du}{dx}\right|}\\ &\quad\cdot\frac{d}{dt}\left(\frac{d\phi}{dx}+t\frac{du}{dx}\right)\Bigg|_{t=0}\\ &= \mu'\left(\left|\frac{d\phi}{dx}+t\frac{du}{dx}\right|\right)\cdot\frac{\frac{d\phi}{dx}+t\frac{du}{dx}}{\left|\frac{d\phi}{dx}+t\frac{du}{dx}\right|}\cdot\frac{du}{dx}\Bigg|_{t=0}\\ &= \mu'\left(\left|\frac{d\phi}{dx}\right|\right)\cdot\frac{\frac{d\phi}{dx}}{\left|\frac{d\phi}{dx}\right|}\cdot\frac{du}{dx}\Bigg|_{t=0}.\end{aligned}$$

Now that we have the formulas 2.7 and 2.9 for a and a' at hand, we can formulate the Newton iteration ([Süli and Mayers, 2003, section 4.3]) as

$$a'(u_{n-1};u_n,v_n)=a'(u_{n-1};u_{n-1},v_n)-\left(a(u_{n-1};u_{n-1},v_n)-\int_0^1 fv_n dx\right),\tag{2.10}$$

where n denotes the index of the iteration. With an initial guess u_0 consisting of a zero vector, we usually arrive at an accurate solution in a few iterations.

In the second argument of a in 2.10, we substitute the linear combination of the basis functions ϕ_j

$$\begin{aligned}u_n(x)&=\sum_{j=1}^{N-1}u_n(jh)\phi_j(x)\\ &=\sum_{j=1}^{N-1}u_n^{(j)}\phi_j(x)\end{aligned}$$

and obtain

$$\begin{aligned}a'\left(u_{n-1};\sum_{j=1}^{N-1}u_n^{(j)}\phi_j,v_n\right)&=a'\left(u_{n-1};\sum_{j=1}^{N-1}u_{n-1}^{(j)}\phi_j,v_n\right)\\ &\quad-a\left(u_{n-1};\sum_{j=1}^{N-1}u_{n-1}^{(j)}\phi_j,v_n\right)+\int_0^1 fv_n dx\end{aligned}\tag{2.11}$$

for $j=0,\ldots,N-1$. Indices $j=0$ and $j=N$ are not included in the sum because of the zero boundary condition 2.4.

Since the form a is linear in the second argument, the equation 2.11 can be thought of as a linear system that can be represented by the following matrix equation:

$$\boldsymbol{A}\boldsymbol{u}_n=\boldsymbol{A}\boldsymbol{u}_{n-1}-\left(\boldsymbol{B}\boldsymbol{u}_{n-1}-\boldsymbol{f}\right),\tag{2.12}$$

where $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{(N-1)\times(N-1)}$ and

$$\begin{aligned}
a_{ij} &= a'(u_{n-1}, \phi_j, \phi_i) \\
&= \int_0^1 \mu'\left(\left|\frac{du_{n-1}}{dx}\right|\right)\left|\frac{du_{n-1}}{dx}\right|^{-1}\left(\frac{du_{n-1}}{dx}\right)^2 \frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx \\
&\quad + \int_0^1 \mu\left(\left|\frac{du_{n-1}}{dx}\right|\right)\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx \\
b_{ij} &= \int_0^1 \mu\left(\left|\frac{du_{n-1}}{dx}\right|\right)\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx \\
f_i &= \int_0^1 f\phi_i dx
\end{aligned}$$

are the the elements of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ and $i, j = 1, \ldots, N-1$. This can be again simplified due to the fact that the function ϕ_i have a compact support. For the diagonal elements, we attain

$$\begin{aligned}
a_{ii} &= \int_{x_{i-1}}^{x_i}\left\{\mu'\left(\left|\frac{du_{n-1}}{dx}\right|\right)\left|\frac{du_{n-1}}{dx}\right|^{-1}\left(\frac{du_{n-1}}{dx}\right)^2 + \mu\left(|u_{n-1}|\right)\right\}\left(\frac{d\phi_i}{dx}\right)^2 dx \\
&\quad + \int_{x_i}^{x_{i+1}}\left\{\mu'\left(\left|\frac{du_{n-1}}{dx}\right|\right)\left|\frac{du_{n-1}}{dx}\right|^{-1}\left(\frac{du_{n-1}}{dx}\right)^2 + \mu\left(|u_{n-1}|\right)\right\}\left(\frac{d\phi_i}{dx}\right)^2 dx \\
&= \frac{1}{h^2}\int_{x_{i-1}}^{x_i}\left\{\mu'\left(\left|\frac{du_{n-1}}{dx}\right|\right)\left|\frac{du_{n-1}}{dx}\right|^{-1}\left(\frac{du_{n-1}}{dx}\right)^2 + \mu\left(|u_{n-1}|\right)\right\}dx \\
&\quad + \frac{1}{h^2}\int_{x_i}^{x_{i+1}}\left\{\mu'\left(\left|\frac{du_{n-1}}{dx}\right|\right)\left|\frac{du_{n-1}}{dx}\right|^{-1}\left(\frac{du_{n-1}}{dx}\right)^2 + \mu\left(|u_{n-1}|\right)\right\}dx.
\end{aligned}$$

If $i = j+1$, the elements are on the first subdiagonal and

$$a_{ij} = -\frac{1}{h^2}\int_{x_{i-1}}^{x_i}\left\{\mu'\left(\left|\frac{du_{n-1}}{dx}\right|\right)\left|\frac{du_{n-1}}{dx}\right|^{-1}\left(\frac{du_{n-1}}{dx}\right)^2 + \mu\left(|u_{n-1}|\right)\right\}dx.$$

If $i = j-1$, the elements are on the first superdiagonal and

$$a_{ij} = -\frac{1}{h^2}\int_{x_i}^{x_{i+1}}\left\{\mu'\left(\left|\frac{du_{n-1}}{dx}\right|\right)\left|\frac{du_{n-1}}{dx}\right|^{-1}\left(\frac{du_{n-1}}{dx}\right)^2 + \mu\left(|u_{n-1}|\right)\right\}dx.$$

Otherwise $a_{ij} = 0$.

Similarly for $\boldsymbol{B}$, we have

$$\begin{aligned}
b_{ii} &= \frac{1}{h^2}\int_{x_{i-1}}^{x_i}\mu'\left(\left|\frac{du_{n-1}}{dx}\right|\right)dx + h^2\int_{x_i}^{x_{i+1}}\mu'\left(\left|\frac{du_{n-1}}{dx}\right|\right)dx, \\
b_{ij} &= \begin{cases} -\frac{1}{h^2}\int_{x_{i-1}}^{x_i}\mu'(|\frac{du_{n-1}}{dx}|)dx & \text{if } i = j+1, \\ -\frac{1}{h^2}\int_{x_i}^{x_{i+1}}\mu'(|\frac{du_{n-1}}{dx}|)dx & \text{if } i = j-1, \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}$$

We are now in a situation analogous to the matrix equation 1.13, the only difference being that the 2.12 has to be solved multiple times for us to converge to an precise result.

2.2.3 Pre-training

In the succeeding chapter, we will see that, when we employ the deep learning approach with the L-BFGS optimizer, the computation process becomes very long. To at least partially mitigate that, we will pre-train the neural network with a solution to a different but similar equation. In simple words, we first solve the similar equation, and then save the FNN $\hat{u}$ approximating its solution. This we load before computing the solution of 2.13 and start the training from there.

Let us define a problem analogous to the nonlinear equation 2.3.

$$-a\frac{d^2u}{dx^2} = f \quad \text{in} \quad (0,1), \tag{2.13}$$
$$u(0) = u(1) = 0,$$

where f is chose again as 2.6 and a is a positive constant. This is the equation that will be utilized in the pre-training for the original nonlinear equation 2.3.

By "similar" solutions, we mean that that the solution are similar in shape, i.e. are well comparable. This characterization will suffice, for we will not perform any analysis and only demonstrate the concept by means of a numerical experiment.

2.3 Singularly Perturbed Problem

The last example we will be interested in is a one-dimensional linear ODE perturbed by a small constant at its highest derivative.

$$-\epsilon\frac{d^2u}{dx^2} + b\frac{du}{dx} = 1 \quad \text{in } (0,1), \tag{2.14}$$
$$u(0) = u(1) = 0, \tag{2.15}$$

where $b \gg \epsilon$. Throughout the thesis, we will work with $b = 1$ and $\epsilon \sim 10^{-2}$. Near $x = 1$, this choice results in behaviour much different from the rest of the interval (see figure 2.1 on the next page). We can observe this from the analytical solution

$$u(x) = \frac{xe^{b/\epsilon} - e^{bx/\epsilon} - x + 1}{b\left(e^{b/\epsilon} - 1\right)}.$$

2.3.1 Physics-Informed Neural Network

Sadly, the physics informed neural network cannot cope with equation 2.14 in the setting explained on the previous two examples. This is a known phenomenon at length explained in Arzani et al. [2023]. The authors propose the so called boundary-layer PINN (BL-PINN), which enables us to manage cases, where the regular PINN cannot find a meaningful solution.

The central thought behind BL-PINN is to split the domain Ω into two parts: the outer part where the solution behaves nicely and the inner part near the

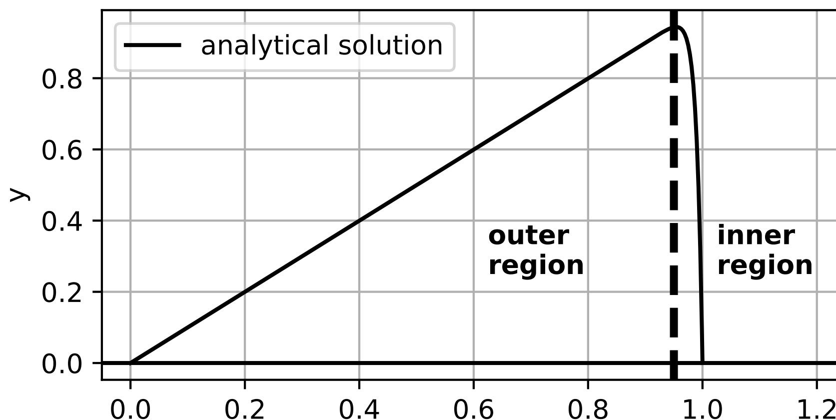

Figure 2.1: An approximate division of the domain $\Omega = [0, 1]$ into the outer an inner region. If we substitute $\epsilon = 0.01$ and $b = 1$, we find that the derivative $u'(x)$ is equal to -99 at $x = 1$.

boundary in which large gradients occurs. On both subdomains, the equation 2.14 is approximated with an asymptotic expansion. Additionally, the very thin inner part is "stretched" with a coordinate transformation which enables a precise solution. Ultimately, we solve the equation via two coupled PINNs, one for each part of the domain.

For the purposes of this example, we will devise a simplified approach similar to BL-PINN. We first solve the outer approximation, the inner approximation is then solved separately (without stretching the interval) by guessing the constant C, which determines the derivative a $x = 1$. Both solutions are then "cut" and "tied together" at their intersection which produces the final solution.

Outer approximation

$$b\frac{du_{out}}{dx} = 1$$
$$u_{out}(0) = 1$$

Inner approximation

$$-\epsilon\frac{d^2u_{in}}{dx^2} = 1$$
$$u_{in}(1) = 0$$
$$u'_{in}(1) = C$$

Another viable procedure, which will not be pursued in the thesis, is to first solve the outer approximation and choose a point $A \in (0, 1)$ inside the domain. The value of the outer solution at that point $u(A)$ would then serve as a left boundary condition for the inner part, i.e. $u_{in}(A) = u_{out}(A)$.

2.3.2 Finite Element Method

As always, we obtain a weak formulation of 2.14 by multiplication by a test function v, integration over the whole domain and integration by parts.

$$\epsilon \int_0^1 u'(x)v'(x)dx + b \int_0^1 u'(x)v(x)dx = \int_0^1 f(x)v(x)dx,$$

where $u, v \in H_0^1$.

Discretizing on an equidistant mesh $\{x_i\}_{i=0}^N$ and exploiting the bilinearity of the left-hand side in u and v, we retrieve

$$\epsilon \sum_{j=1}^{N-1} u(x_j) \int_0^1 \phi_j' \phi_i' + b \sum_{j=1}^{N-1} u(x_j) \int_0^1 \phi_j' \phi_i = \int_0^1 f \phi_i, \quad i = 1, \ldots, N-1,$$

where $u(x)$ was expressed as a linear combination of basis functions $\{\phi_i\}_{i=1}^{N-1}$.

Utilizing the compact support of ϕ_i, $i = 1, \ldots, N-1$, only a few integrals in the first and second sum are non-zero. This leads to a matrix equation similar to 1.13 with one additional term corresponding to bu'.

$$\left(\frac{b}{2} \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ -1 & 0 & 1 & & \vdots \\ 0 & -1 & 0 & \ddots & 0 \\ \vdots & & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & -1 & 0 \end{bmatrix} - \frac{\epsilon}{h^2} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \right) \begin{bmatrix} u^1 \\ \\ \vdots \\ \\ u^{N-1} \end{bmatrix} = \begin{bmatrix} f^1 \\ \\ \vdots \\ \\ f^{N-1} \end{bmatrix}.$$

3. Numerical Results

This chapter provides an outcome of the computations described above as well a brief discussion.

3.1 Simple Example

The first thing we observe on the machine learning approach, when running the DeepXDE code multiple times, is the randomness of the result. This stems mainly from the random choice of training points $\mathcal{T}$ and can be combated by fixing a random seed at the beginning of the program. The authors of the DeepXDE library recommend to run the code multiple times and choose the solution with the smallest training loss as solution, because, currently, there in no way of predicting the approximation error of PINN. It is an open research problem (Lu et al. [2021]).

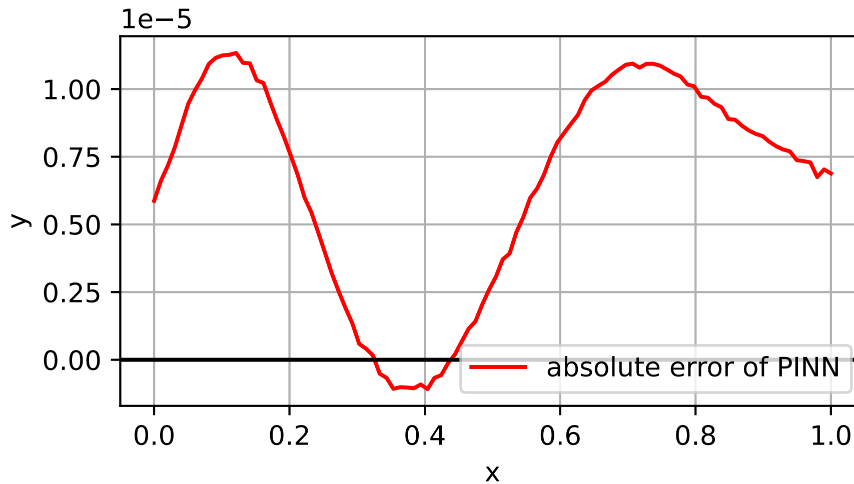

Figure 3.1: Absolute error of a solution to 2.1 provided by PINN after 10000 training epochs using the random seed 6. All other hyperparameters are taken from section 2.1.1.

We may expose the issue on the following example. Setting random seed equal to 6, the PINN provides a solution with an error displayed in figure 3.1. If we then decide to switch to double-precision, the order of error rises from 10^{-5} to 10^{-2}. However, increasing the number of epochs from 10000 to 20000 fixes the issue by achieving the order of 10^{-6}. This occurs due to the evolution of the loss which, from the global point of view, decreases during training, but this descent is not monotonous.

The second issue we may notice from the figure 3.1 is the fact that the boundary conditions are not satisfied exactly. By design, we enforce "soft" boundary conditions via the boundary part of the loss $\mathcal{L}_b$. If this approach is not suitable, we can replace the approximating neural network $\mathcal{N}^L(x)$ by $x(x-1)\tilde{\mathcal{N}}^L(x)$, which satisfies the boundary conditions automatically (Lagaris et al. [1998]).

On the other hand, finite elements do not face any of the problems mentioned above. As can be seen in figure 3.2, only 51 dividing point are enough for a very

accurate solution. As opposed to PINN, where it is necessary to fix a random seed, solving a system of linear algebraic equations is a deterministic process.

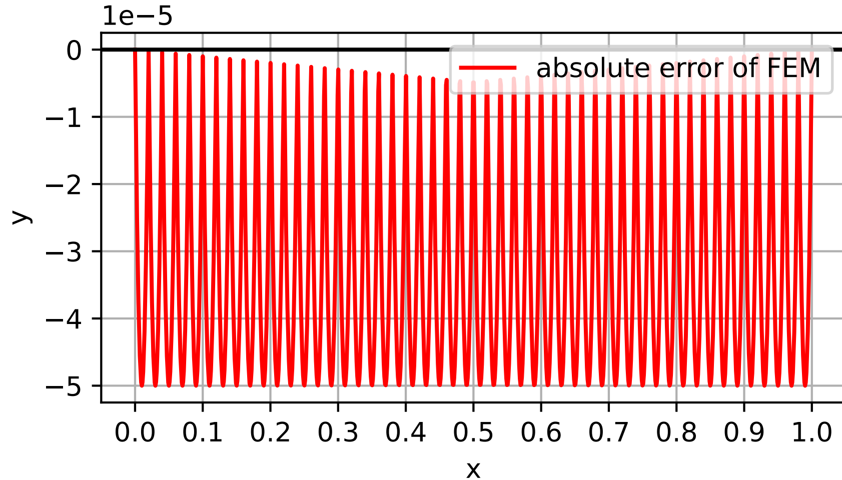

Figure 3.2: Absolute error of a solution obtained from the finite element code described in 1.5. We chose 51 equally space points dividing the interval $[0, 1]$ into 50 subintervals.

For future comparison with the nonlinear example, we state that the evaluation time is bellow 0.1 seconds for all $N \in \{16, 31, 51, 101, 201\}$. Detailed result are available in table 3.1 below.

N	ℓ^2 rel. error	time (s)
16	3.7×10^{-3}	< 0.1
31	9.3×10^{-4}	< 0.1
51	3.3×10^{-4}	< 0.1
101	8.4×10^{-5}	< 0.1
201	2.1×10^{-5}	< 0.1

Table 3.1: Evolution of ℓ^2 relative error with increasing N in the simple example.

3.2 Nonlinear Problem

The nonlinear example shows the very same behaviour as the previous linear example. As we have already discussed, there is no significant difference between the two examples from the perspective of PINN.

Let us therefore move towards pre-training described in the subsection 2.2.3. As was mentioned above, the Poisson's equation 2.13 is used to generate solutions for different values of parameter a, which then serve as a starting point for solving 2.3. For each value of a, only 3000 epoch of training take place, because a rough approximation suffices as a starting point for the nonlinear equation. This approach will be compared to training starting from default initializers available in the library.

optimizer	Adam
layer size	$[1] + [20] \times 3 + [1]$
initializer	Glorot normal
learning rate	0.001
epochs	3000
training points	$16 + 2$
a	$\in [1, 2]$

Table 3.2: Hyperparameters used during pre-training.

When solving the equation of interest, we utilize both Adam and L-BFGS optimizer. As a stopping criterion for training we chose the ℓ^2 relative error. Unfortunately, DeepXDE doesn't provide us with an easy way to stop the training, as soon as we reach the desired precision, neither a straightforward modification is possible. The only two available criterions are the training loss and the test loss, none of which can be reliably converted to ℓ^2 relative error, at least to our knowledge. For this reason, we chose to train for a fixed number of epochs repetitively, until a solution error below the desired tolerance was achieved. In the case of L-BFGS, we chose 10^{-4}, with Adam, we opted for 10^{-3}, because higher accuracy could not be reliably attained for all initializers and random seeds.

optimizer	no pre-training mean	pre-training mean	decrease/ increase
L-BFGS	632.7	559.1	−13 %
Adam	3594.3	5550.0	+54 %

Table 3.3: An average number of epoch needed for convergence in pre-training.

In the table 3.3, we can see that pre-training actually helps to decrease number of iterations needed for convergence, when we employ L-BFGS. Employing pre-training, number of epochs is mostly independent of a. The same cannot be said about Adam, where the mean number of iterations is actually higher with pre-training. However for $a = 1.35$ and $a = 1.4$ the number drops to a few hundreds, which is much lower than the no pre-training mean (see the attachments A).

Moving on to the finite elements, just 16 points were enough to produce an ℓ^2 relative error of 3.3×10^{-4}. Further increase in the number of points N proved to reduce the error, as can be see in the table 3.4. However, the computational costs scale superlinearly with increasing N, it is therefore very inefficient beyond a certain value.

Looking back at the previous example, we see that the computation time increased immensely, when we switched from a linear Poisson's equation to its nonlinear analogue. Some of it can be attributed to the simple implementation of the latter case, but the repeated solving of a linear algebraic system is fundamentally more expensive then doing it only once.

Contrarily, the PINN does not face these issues. There is no fundamental difference between solving a nonlinear differential equation and a nonlinear one. If we unify the hyperparameters of examples 2.1 and 2.2 and use the Adam

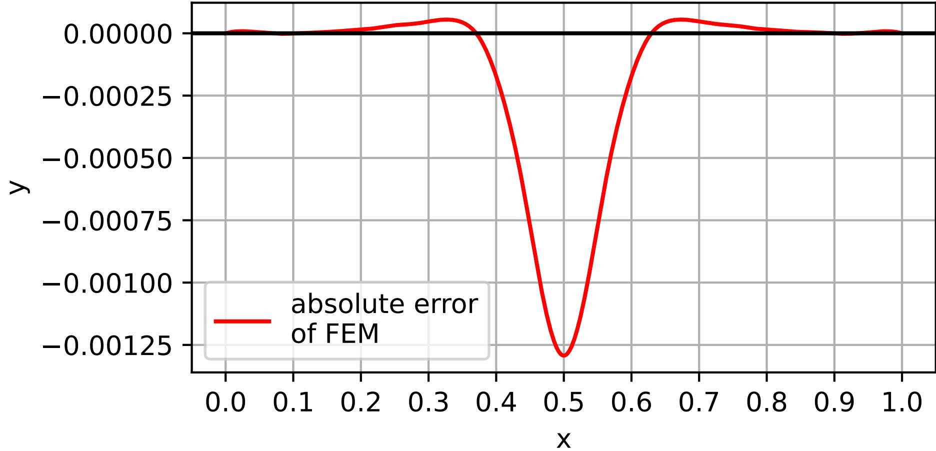

Figure 3.3: Absolute error of a solution to the nonlinear equation 2.3 provided by FEM ($N = 51$). The result was interpolated by a cubic spline.

N	ℓ^2 rel. error	time (s)
16	3.3×10^{-4}	0.6
31	8.2×10^{-5}	2.4
51	3.0×10^{-5}	6.6
101	7.4×10^{-6}	27.0
201	1.8×10^{-6}	114

Table 3.4: Evolution of ℓ^2 relative error and computation time with increasing N in the nonlinear example. Newton's method converged in 5 iterations for all values of N.

optimizer, we discover that, the first example takes about 14 seconds to compute and the second one approximately 18 seconds.

3.3 Singularly Perturbed Problem

As we have mentioned in the previous chapter, the machine learning approach is not as smooth as in the previous examples. For higher values of ϵ, PINN can approximate the analytical solution quite well. It starts to deviate heavily around $\epsilon = 0.05$ and is unusable for values below. The primitive version of BL-PINN described above, allows us to extend the range for ϵ approximately to 0.005. For smaller values, we unfortunately run again into numerical issues. This prevents us from seeing the full potential of the approximations, because u_{out} and u_{in} have the best fit, when ϵ is very small.

Having witnessed that PINN can solve singularly perturbed problems, we move on to the finite element method. Figure 3.5 suggests that not even here is the situation without troubles. For a small number of dividing points N, we observe very large oscillations near the right boundary. As opposed to the machine learning approach, the issue can be mitigated by increasing N, however, smaller ϵ again reintroduce the oscillations. A proper choice of ϵ and N is related

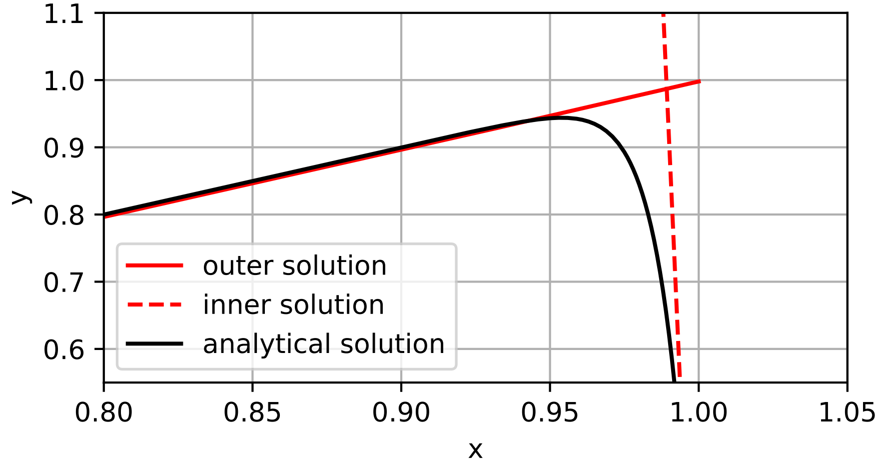

Figure 3.4: Analytical solution of 2.14 together with the outer and inner approximations obtained from the simplified BL-PINN. Exact analytical solutions: $u_{out}(x) = x$, $u_{in}(x) = -50x^2 + x + 49$ for $C = -99$.

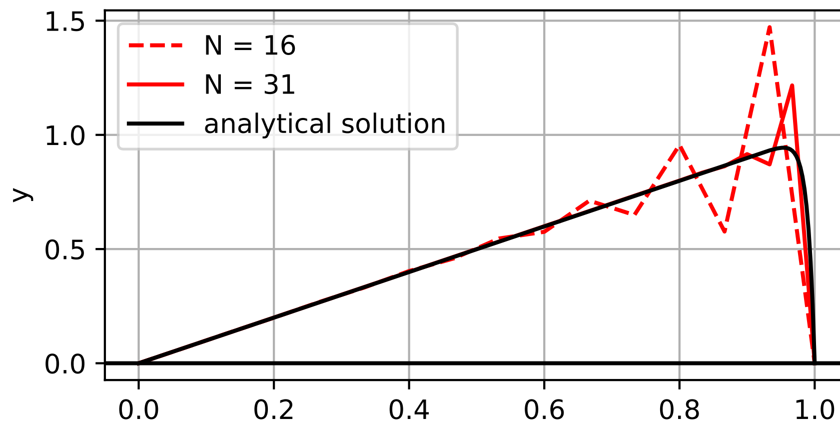

Figure 3.5: Analytical solution of 2.14 compared to numerical solutions obtained from FEM with different numbers of dividing points.

to a so called Péclet's number Pe, which is defined as a ration between coefficients b and ϵ ($\text{Pe} = \frac{b}{\epsilon}$).

Choosing $N = 1001$, the ℓ^2 relative error is of the order 10^{-5}, which we consider to be an accurate result. Thousands of points needed for a precise solution may seem like a lot, however, considering that the usual computation time for PINN is in tens of seconds, it is still a faster and perhaps an even more reliable option.

3.4 Comparison Between PINN and FEM

A the end of this chapter, we include a listing the major advantages and disadvantages of PINN, many of which we have encountered in the text above. The specified properties are meant to be in direct contrast to the reference finite element method.

Advantages

- easy integration of different kinds of boundary conditions
- linear and nonlinear problems are handled in the same way
- the code closely resembles the mathematical formulation (DeepXDE)
- solving a particular equation does not require a tedious derivation of the code
- general flexibility in solving various types of PDEs

Disadvantages

- long computation time even for simple problems
- no known error bounds
- has troubles dealing with strong nonlinearities and discontinuities
- some problems need a significant modification (BL-PINN)
- multiple runs with different settings are often necessary to achieve and optimal result

Conclusion

Previous chapters have demonstrated to us the possibilities opened by neural networks applied to the field of differential equations. We have seen that in some respects, such as versatility, PINN outperforms the traditional numerical methods significantly. For example, in terms of computation time, however, FEM is still considerably better. Although PINNs are effective in a wide range of problems, they are by no means a silver bullet. They still struggle with highly nonliner behaviour and suffer from the same issues as optimization problems. Since they are often solving non-convex optimization problems, there is no guarantee of finding a global minimum.

A useful application of the machine learning approach might be parametric PDEs, where the values of the parameter are from a narrow interval. We can therefore initialize training for one value with another much like we have seen in the example 2.2. This brings, in many cases, a profound decrease in number of epochs and computation time. One might even consider a more sophisticated approach (Uriarte et al. [2022]) that is inspired by the finite element method and has already shown favourable results. Nonetheless, some may still prefer iterative methods, since they in principle offer the same benefit, thanks to the choice of initial guess.

Example 2.3 was a great illustration of the limitations faced by PINN. We have observed the difficulties tied with thin boundary layer problems. Unlike the previous nonlinear example, the approach needed substantial modification in order to return meaningful results. All in all, just like FEM required major adjustments in the nonlinear case, PINN needed similar treatment when solving the thin boundary layer problem.

Several other modifications to PINN have emerged in recent years such as XPINN or DPINN, each one suited for different problems. As a result, the deep learning approach to ODEs and PDEs is gradually becoming more and more applicable. This in also thanks to the rise of specialized processors (TPUs), which speed up computations.

In our opinion, pre-training is the topic that deserves a more thorough investigation. The nonlinear example showed us the possibility to substantially decrease computation time if the we choose the right set of hyperparameters initializing the training. For reasons unknown, some choices decreased the time by an order of magnitude, other options caused the exact opposite.

Bibliography

Amirhossein Arzani, Kevin W. Cassel, and Roshan M. D'Souza. Theory-guided physics-informed neural networks for boundary layer problems with singular perturbation. *Journal of Computational Physics*, 473:111768, 2023. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2022.111768. URL https://www.sciencedirect.com/science/article/pii/S0021999122008312.

Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995. doi: 10.1137/0916069. URL https://doi.org/10.1137/0916069.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998. doi: 10.1109/72.712178.

Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis, jan 2021. URL https://doi.org/10.1137%2F19m1274067.

Endre Süli and David F. Mayers. *An Introduction to Numerical Analysis*. Cambridge University Press, 2003. doi: 10.1017/CBO9780511801181.

Carlos Uriarte, David Pardo, and Ángel Javier Omella. A finite element based deep learning solver for parametric pdes. *Computer Methods in Applied Mechanics and Engineering*, 391:114562, 2022. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2021.114562. URL https://www.sciencedirect.com/science/article/pii/S0045782521007374.

List of Figures

1.1 A scheme of a FNN. 4
1.2 Evolution of parameters $\boldsymbol{\theta}$ in the space $\mathbb{R}^n$. 7

2.1 An approximate division of the domain Ω into the outer an inner region. 19

3.1 Absolute error of a solution to 2.1 provided by PINN after 10000 training epochs using the random seed 6. 21
3.2 Absolute error of a solution obtained from the finite element code described in 1.5. 22
3.3 Absolute error of a solution to the nonlinear equation 2.3 provided by FEM ($N = 51$). 24
3.4 Analytical solution of 2.14 together with the outer and inner approximations obtained the simplified BL-PINN. 25
3.5 Analytical solution of 2.14 compared to numerical solutions obtained from FEM. 25

List of Tables

3.1 Evolution of ℓ^2 relative error with increasing N in the simple example. 22
3.2 Hyperparameters used during pre-training. 23
3.3 An average number of epoch needed for convergence in pre-training. 23
3.4 Evolution of ℓ^2 relative error and computation time with increasing N in the nonlinear example. 24

A.1 Number of epochs needed for convergence below a desired tolerance. Solver: L-BFGS. 32
A.2 Number of epochs needed for convergence below a desired tolerance. Solver: Adam. 33

List of Abbreviations

AD automatic differentiation (algorithmic differentiation). 3, 7

BL-PINN boundary-layer physics-informed neural network. 18, 19, 24, 25, 26, 29

DPINN distributed physics-informed neural networks. 27

FDM finite difference method. 2

FEM finite element method. 2, 3, 7, 10, 24, 25, 27, 29

FNN feed-forward neural network (multilayer perceptron). 3, 4, 5, 7, 18, 29

L-BFGS limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm. 6, 14, 18, 23, 30, 32

ODE ordinary differential equation. 18, 27

PDE partial differential equation. 26, 27

PINN physics-informed neural network. 2, 3, 8, 12, 18, 19, 21, 22, 23, 24, 25, 26, 27, 29

TPU tensor processing unit. 27

XPINN extended physics-informed neural network. 27

A. Attachments

A.1 Detailed Results from Pre-training

Initializer/a	Random Seed					Mean Epochs
	0	1	2	3	4	
Glorot normal	920	694	463	1134	912	824.6
Glorot uniform	460	472	234	1162	686	602.8
He normal	681	918	230	231	1128	637.6
He uniform	467	222	706	464	668	505.4
LeCun normal	468	695	693	468	464	557.6
LeCun uniform	914	719	473	686	911	740.6
Orthogonal	464	482	460	477	917	560.0
1.0	451					451.0
1.05	440					440.0
1.1	445					445.0
1.15	667					667.0
1.2	443					443.0
1.25	665					665.0
1.3	438					438.0
1.35	675					675.0
1.4	447					447.0
1.45	691					691.0
1.5	895					895.0
1.6	675					675.0
1.7	676					676.0
1.8	440					440.0
1.9	453					453.0
2.0	444					444.0
no pre-training mean						**632.7**
pre-training mean						**559.1**

Table A.1: Number of epochs needed for convergence below a desired tolerance. Solver: **L-BFGS**, tolerance: $\mathbf{10^{-4}}$, step: **200** epochs. Note that in the pre-training part, the number of epochs doesn't depend on the random seed.

Initializer/a	Random Seed					Mean Epochs
	0	1	2	3	4	
Glorot normal	2200	2400	2400	1000	3000	2200.0
Glorot uniform	1800	1400	1600	2400	1800	1800.0
He normal	2200	3200	3600	800	2600	2480.0
He uniform	800	2200	8200	3200	13600	5600.0
LeCun normal	2600	6200	7400	4600	2200	4600.0
LeCun uniform	2200	1200	2200	2200	2600	2080.0
Orthogonal	7800	3600	3000	1800	15800	6400.0
1.0	8200					11400.0
1.05	8200					10600.0
1.1	11200					11200.0
1.15	1800					1800.0
1.2	1600					1600.0
1.25	4400					4400.0
1.3	4400					4400.0
1.35	600					600.0
1.4	800					800.0
1.45	6600					6600.0
1.5	2800					2800.0
1.6	3000					3000.0
1.7	10200					10200.0
1.8	7200					7200.0
1.9	4000					4000.0
2.0	8200					8200.0
no pre-training mean						**3594.3**
pre-training mean						**5550.0**

Table A.2: Number of epochs needed for convergence below a desired tolerance. Solver: **Adam**, tolerance: $\mathbf{10^{-3}}$, step: **200** epochs. Note that in the pre-training part, the number of epochs doesn't depend on the random seed.