Solving PDE's using Physics Informed Neural Networks
Project 3 – FYS-STK4155

Anders Thorstad Bø

*Department of Mathematics (Fluid Mechanics), University of Oslo, Oslo, Norway**

In this project, I have implemented programs that attempts to find solutions to partial differential equations using Physics Informed Neural Networks (PINNs). The main goal is to see how the PINN-approach holds up against standard numerical methods for solving PDE's, such as the finite difference method (FDM), which I use as a benchmark method. The main PDE's are the unsteady diffusion and wave equations in both one and two spatial dimensions. I have implemented two PINNs, one using TensorFlow, and one using Autograd to perform the network building and training. The analysis will include methods for model selection, like standard and random grid searches, and gradient descent methods that use adaptive learning rates, such as ADAM. The analysis will also involve the impact from parameters such as network depth, learning rates, regularization, among others, as well as some of the method choices where I look at different activation functions. Some highlights from the analysis is that the networks solve the one-dimensional equations well, but struggle when the problem is extended to two dimensions. In the 1D-cases, the PINNs obtained global errors comparable to the FDM-solver, namely $E = 0.01105$, against the FDM's $E = 0.00247$.

I. INTRODUCTION

Partial differential equations (PDEs) are powerful tool for the study of natural phenomena and physical systems, as they seek to describe the underlying behavior of a system using a specific relationships between an unknown function of two or more independent variables, and its derivatives (Tveito and Winther, 2005).

Standard methods for finding solutions to PDEs such as the *Finite Difference Method* (FDM) and *Finite Element Method* (FEM) are powerful and well-proven, but the methods also become more and more involved as the complexity of the system grows. They also require an implementation that is more or less tailored to a specific problem (Langtangen and Linge, 2017).

Several methods that utilize the strengths of neural networks (NNs) have been proposed in the recent years as models to solve PDEs. One of the main strengths is that a NN usually is a more general purpose than the traditional approaches. This is especially noticeable for non-linear systems, and for system where shocks and strong convection is present, where FDM and FEM are known to struggle (Cuomo et al., 2022).

In this project, I have focused on an approach commonly called *Physics-Informed Neural Networks* (PINNs), which can be seen as a specialization of a standard, fully-connected *Feed-Forward Neural Network* (FFNN). The PINNs uses information from the PDEs to create a generative model that is able to predict the behavior of the physical system (Cuomo et al., 2022).

My aim is to show how two different approaches of implementing a PINN perform against a FDM implementation that solves the wave equation and the diffusion equation in up to two spatial dimensions. The PINN-implementations are based on chapter 15 in (Hjorth-Jensen, 2023), and the companion programs to the paper (Raissi et al., 2019). Both uses an *automatic differentiation* approach for computing gradients, and the latter also uses the extensive **TensorFlow**-library, mostly through the *Keras-API* (Abadi et al., 2016; Maclaurin et al., 2015).

For the PINNs, I will also show how hyperparameter and network structure choices impact the solution, as well as choices related to gradient descent optimizers like RMSprop and ADAM, and different activation functions. For verification of both the PINN- and FDM- approach, I am using known analytical solutions to the two PDEs to compute the global error.

The report will go through the relevant theoretical concepts and methods in Sec.II, first describing the general concept of PDEs, and the building blocks of the neural network, with specific focus on how to make it "physics-informed". Following this, I give a quick breakdown of the FDM used, and finish off with a description of the model selection and assessment process.

Sec.III describes the program-, and class-structure of the implementation, as well as a quick breakdown of

* andetb@uio.no; https://github.com/andersthorstadboe/project-3-fys-stk4155-pinn

the basic training and solution algorithms for the two solution methods. I also give a short introduction to how to access and use the programs written as part of this project.

In the final parts, Sec.IV and Sec.V, I will show a selection of results from the model selection process, and the simulations, and also present a comparison between the methods. Some notable highlights from the analysis is that the PINN-predictions of the 1D diffusion equation give a global ℓ^2-errors, $E = \{0.02622,\ 0.01105\}$, while the FDM for the same problem yields an error of $E = 0.00247$

In the appendices, I present some additional results from the model selection process and analysis, as well as provide some more detail activation functions, the PINN-modeling choices, and discretization using finite differences.

II. THEORY AND METHODS

To start off, this report will address some important theoretical aspects. First, a quick description of what PDEs are, and the two specific ones used here. Then this section presents the concepts relevant for the two PINN-approaches. It will also give a quick presentation of the FD-method, and lastly address the theory and metrics behind the model selection and assessment.

A. General PDE's

A partial differential equation can be defined as an equation containing an unknown function, $u = u(z) = u(t, \boldsymbol{x})$, of two or more variables, where $\boldsymbol{x} = [x_1, \ldots, x_N]$, defined on a domain $\Omega \subset \mathbb{R}^N$, and containing a collection of the functions partial derivatives (Evans, 2010).

A general representation of this is

$$\begin{aligned} \mathcal{F}(u(z); \gamma) &= f(z) \quad \text{for } z \in \Omega \\ \mathcal{B}(u(z)) &= g(z) \quad \text{for } z \in \partial\Omega \end{aligned} \tag{A.1}$$

where $\partial\Omega$ denotes the boundary of Ω, and γ a set of parameters related to the physics the PDE aims to describe. $\mathcal{F}$ is here a non-linear differential operator describing the specific structure and relationship between the function and its partial derivatives. $\mathcal{B}$ is an operator describing the initial and boundary conditions, with $f(z)$ and $g(z)$ denoting the source term or function, and the boundary function, respectively (Cuomo et al., 2022).

This project concerns two specific PDE's, namely the wave equation and the diffusion equation. For vector $\boldsymbol{x} = [x, y]$, the wave equation can be written as

$$\frac{\partial^2 u}{\partial t^2} - c^2 \nabla^2 u = f(t, \boldsymbol{x}) \tag{A.2}$$

where c denotes a constant wave speed. The initial conditions can be defined generally as

$$\begin{aligned} u(0, \boldsymbol{x}) &= g(0, \boldsymbol{x}) = c_1(\boldsymbol{x}) \\ \frac{\partial u}{\partial t}|_{t=0} &= \frac{\partial g}{\partial t}|_{t=0} = c_2(\boldsymbol{x}) \end{aligned} \tag{A.3}$$

for some function $g(t, \boldsymbol{x})$ defining the initial conditions of the specific version of the PDE.

The boundary condition for this equation will be Dirichlet boundary condition, where the function values is prescribed on the boundary, $\partial\Omega$. In general this can be defined as

$$u(t, \boldsymbol{x}) = h(t, \boldsymbol{x}) \text{ for } \boldsymbol{x} \in \partial\Omega \tag{A.4}$$

As for the initial conditions, $h(t, \boldsymbol{x})$ must be defined to provide valid boundary values for the specific system the PDE models (Langtangen and Linge, 2017).

The diffusion equation can be written as

$$\frac{\partial u}{\partial t} - \nabla \cdot (D \nabla u) = f(t, \boldsymbol{x}) \tag{A.5}$$

with D as the so-called diffusion coefficient which can be taken either as a constant field, or as a variable. As this only has a first order derivative wrt. to time, the initial condition will be

$$u(0, \boldsymbol{x}) = u_e(0, \boldsymbol{x}) = c_1(\boldsymbol{x})$$

The boundary conditions are the same as in Eq.(A.4). For the purposes of this study, D will be considered constant, and $u_e(t, \boldsymbol{x})$ will denote some valid solution to the PDEs, used to assess the model performance. (Langtangen and Linge, 2017).

For the wave equation, on the domain $[0, L_x] \times [0, L_y]$, this solution will be

$$u_e(t, x, y) = a_0 \sin\left(\frac{k_x x}{L_x}\right) \sin\left(\frac{k_y y}{L_y}\right) \cos(\omega t) \tag{A.6}$$

with a_0 as a constant amplitude, $k_\alpha = m_\alpha \pi,\ \alpha = \{x, y\}$ as the wave number, and $\omega = \boldsymbol{k}c = (k_x \boldsymbol{i} + k_y \boldsymbol{j})$ as the wave dispersion coefficient. For the diffusion equation, the solution will be

$$\begin{aligned} u_e(t, x, y) &= a_0 \sin\left(\frac{\pi x}{L_x}\right) \sin\left(\frac{\pi y}{L_y}\right)\ \mathrm{e}^{-\rho t} \\ \rho &= D\pi^2 \left(\frac{1}{L_x^2} + \frac{1}{L_y^2}\right) \end{aligned} \tag{A.7}$$

with ρ denoting the decay rate of the solution, and arises from the eigenvalues of the solution (Chasnov, 2024).

These two solutions will be the foundation for scoring both the prediction from the neural network and the finite difference simulation.

B. Physics-Informed Neural Networks

The type of *Physics-Informed Neural Network* (PINN) made for this study uses the the standard structure of a FFNN as a basis to approximate the solution of a PDE by minimizing a cost function. The specialization to make the network physics-informed is to incorporate the equation's initial and boundary conditions into the terms of this cost function, as well as using the so-called residual form of PDE at specified points in the domain (Cuomo et al., 2022).

The approach results in an unsupervised learning strategy without any labeled data, and contrary to other numerical methods, the network is a mesh-free technique that directly solves the system by converting it to a cost optimization task (Cuomo et al., 2022).

How a standard FFNN works was covered extensively in my report in (Bø, 2024), so the description here will focus on the specifics of the PINN. However, some important aspects from that report will be repeated for completeness.

1. General structure

As the PINN builds on a FFNN, the general structure can be described as a collection of layers, where each layer is defined by its weights, $\boldsymbol{W}$, and biases, $\boldsymbol{b}$. A layer l then connects to the layers upstream and downstream by a mathematical function $\sigma(z)$, usually called an activation function. A fully connected FFNN is such a network where all neurons in a layer is connected to all other neurons in the down- and upstream layer (Raschka et al., 2022).

Mathematically, the connections between the layers can be described by the layer output of layer l, $\boldsymbol{a}^l$, as

$$\boldsymbol{a}^l = \sigma^l\left((\boldsymbol{W}^l)^T\boldsymbol{a}^{l-1} + \boldsymbol{b}^l\right) \tag{B.1}$$

where the dependent variable, $\boldsymbol{z}^l$ in the activation function, denoted

$$\boldsymbol{z}^l = (\boldsymbol{W}^l)^T\boldsymbol{a}^{l-1} + \boldsymbol{b}^l \tag{B.2}$$

is known as the layer activation. The input layer, at $l = 1$, then takes the information from the input and feeds it forward to the output layer, through the hidden layers of the network. In general, if the output layer is denoted by L, a deep FFNN will have $(L-2)$ hidden layers (Raschka et al., 2022).

For the PINN, the input requires some explanation. The unknown function, u, as the solution to the PDE, depends on the variables $(t, \boldsymbol{x})$. The task of the network is then to predict $u(t, \boldsymbol{x})$ at the *collocation points*. A collocation point is a point in the domain defined by $(t^r, \boldsymbol{x}^r)$, where $\boldsymbol{x}^r = [x_1, x_2, \ldots, x_{N_r-1}]^r$, for N_r as the number of collocation points in the interior of the domain. As the method is mesh-free, these collocation points can be chosen randomly inside the domain. Initial and boundary points are also placed on $\partial\Omega$ in a similar fashion. They can be defined as the points where

$$\begin{aligned} \text{Initial: } (t^0, \boldsymbol{x}^0) &\quad \text{for } \boldsymbol{x}^0 \in \partial\Omega \\ \text{Boundary: } (t^b, \boldsymbol{x}^b) &\quad \text{for } \boldsymbol{x}^b \in \partial\Omega \end{aligned} \tag{B.3}$$

As with the collocation points, the number of points will be denoted by N_b and N_0, for the number of boundary and initial points respectively. They are used to fix the initial and boundary conditions of the PDE, and is a integral part of what makes the network *physics-informed*. (Raissi et al., 2019).

For this project, the collocation are initialized using a random, uniform distribution with lower and upper bound constrained by the domain. The $\boldsymbol{x}_i$'s in the initial points uses the same distribution, and the t^i's are kept fixed at t^0. To distribute the boundary points, the time-coordinates is also picked using a uniform distribution. To ensure that the boundary points actually are located on the domain boundary, the $\boldsymbol{x}_b$'s are here picked using

$$x_j^b = x_{j,0}^b + (x_{j,N}^b - x_{j,0}^b)\ P_b(p) \tag{B.4}$$

where $P_b(p)$ is a Bernoulli distribution defined as

$$P_b(X = x) = \begin{cases} p & \text{if } x = 1 \\ 1 - p & \text{if } x = 0 \end{cases} \tag{B.5}$$

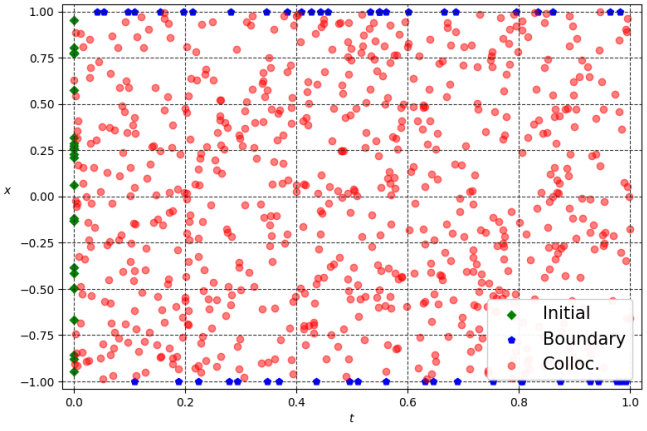

FIG. II.1: Showing an example of how the collocation, initial and boundary points are defined during the network initialization.

so that it gives a discrete probability distribution with outcomes of either $x^b_{j,0}$ or $x^b_{j,N}$ (MathWorld, 2024), (Raissi et al., 2019). Fig.II.1 shows an example of how the different points may be placed in a domain.

2. Trail and residual functions

The next part of the PINN is building up a function based on the PDE that will be used in the optimization process. The two different implementations use slightly different approaches to make this model function. Both versions uses a combination of the actual function value at the initial and boundary points and the prediction from the network.

For the first version, based on (Hjorth-Jensen, 2023; Lagaris et al., 1998), uses the notion of a trail function that is defined as

$$u_t(t, \boldsymbol{x}) = A(t, \boldsymbol{x}) + B(t, \boldsymbol{x}, \mathcal{N}(t, \boldsymbol{x}, P)) \tag{B.6}$$

For this to work, $A(t, \boldsymbol{x})$ must be defined to satisfy the system's boundary and initial conditions, and B must be constructed so it does not contribute to the BC's, such that the network output is set to zero when the function is evaluated at $\partial\Omega$. P is the *parameter matrix*, which contains the weights and biases of the network, i.e. the network structure (Lagaris et al., 1998).

As an example, the trail function for the one-dimensional version of the diffusion equation given in Eq.(A.5), becomes, with

$$A(t, x) = (1 - t_*)u_0(x) \tag{B.7}$$

where $u_0(x)$ is the initial conditions for the PDE, and $t_* = t/T$, T as the total time of the simulation, and

$$B(t, x, \mathcal{N}) = t_* \left(\frac{x}{L} - \frac{x^2}{L^2} \right) \mathcal{N}(t, x, P) \tag{B.8}$$

The final trail function will then be

$$u_t(t, x) = (1-t_*)u_0(x) + t \left(\frac{x}{L} - \frac{x^2}{L^2} \right) \mathcal{N}(t, x, P) \tag{B.9}$$

How to use this in the optimization is covered in the next section, and more details on this can be found in App.C.

The second approach uses the so-called residual form of the PDE in the optimization process. This form is simply the left-hand side of the PDE, which in general will be

$$\hat{R}(z^r) = \mathcal{F}(u(z^r); \gamma) \tag{B.10}$$

where $z^r = (t^r, \boldsymbol{x}^r)$ is the collocation points. The residual function , $\hat{R}$ therefore inherits, or shares, the network parameters, but the residual form also feeds in the underlying behavior of the system as defined by the PDE. This gives the network some "insight" into the physics of the system (Raissi et al., 2019).

As an example, for the one-dimensional diffusion equation is this becomes

$$\hat{R}(t, x) = \frac{\partial^2 \hat{u}}{\partial t^2} - D\frac{\partial^2 \hat{u}}{\partial x^2} - f(t, x) \tag{B.11}$$

This will, together with the boundary and initial functions, defined by equations on the same form as Eq.(A.3),(A.4), be the building blocks for a cost function, presented in the next section. For more details on the specific trail functions and residual function used in the analysis, see App.C.

3. Optimization

As for a standard FFNN, the PINN needs a cost function for the backpropagation and optimization steps of the network training. To fulfill the requirement of the PDE, the network must seek the solution that gives

$$\mathcal{F}(u(z); \gamma) - f(z) = 0 \tag{B.12}$$

so the cost function to be minimized can, generally, be defined as the mean square error (MSE) of this difference, i.e.

$$C(z) = \frac{1}{N}(\mathcal{F}(u(z); \gamma) - f(z))^2 \tag{B.13}$$

where N will be the total number of points in the discretized domain z, so for $z = (t, x)$, then $N = Nt{*}Nx$.

Again, the differences in the two approaches yields somewhat different cost functions. The approach from (Lagaris et al., 1998) expresses the gradient of the cost function using the explicit derivatives of the trail function is constructs. For time-dependent PDE's, these include both derivative in time and space. Therefore, it is often necessary to compute the Jacobian and Hessian matrix for the first and second order derivative of $C(z)$, and then extract the necessary derivatives of the trail function to feed into the cost function. For a function, $\boldsymbol{F}(z) = (F_i(z))_{i=0}^n$, $z = (t, \boldsymbol{x})$, the Jacobian, $\boldsymbol{J}(\boldsymbol{F}(z))$ is

$$\boldsymbol{J}(\boldsymbol{F}(z)) = \begin{bmatrix} \frac{\partial F_1(z)}{\partial t} & \frac{\partial F_1(z)}{\partial x_1} & \cdots & \frac{\partial F_1(z)}{\partial x_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial F_n(z)}{\partial t} & \frac{\partial F_n(z)}{\partial x_1} & \cdots & \frac{\partial F_n(z)}{\partial x_N} \end{bmatrix} \tag{B.14}$$

and the Hessian matrix, $\boldsymbol{H}(\boldsymbol{F}(z))$,

$$\boldsymbol{H}(\boldsymbol{F}(z)) = \begin{bmatrix} \frac{\partial^2 \boldsymbol{F}(z)}{\partial t^2} & \frac{\partial^2 \boldsymbol{F}(z)}{\partial t \partial x_1} & \cdots & \frac{\partial^2 \boldsymbol{F}(z)}{\partial t \partial x_N} \\ \frac{\partial^2 \boldsymbol{F}(z)}{dx_1 \partial t} & \frac{\partial^2 \boldsymbol{F}(z)}{\partial x_1^2} & \cdots & \frac{\partial^2 \boldsymbol{F}(z)}{\partial x_1 \partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \boldsymbol{F}(z)}{\partial x_N \partial t} & \frac{\partial^2 \boldsymbol{F}(z)}{\partial x_N \partial x_1} & \cdots & \frac{\partial^2 \boldsymbol{F}(z)}{\partial x_N^2} \end{bmatrix} \tag{B.15}$$

for $\boldsymbol{x} = [x_1, \ldots, x_N]$ (Lindstrøm, 2017).

To illustrate this, take the one-dimensional diffusion equation. As the network is using the trail function, $u_t(t, x)$ to find the solution to the PDE, this will give a cost function

$$C(z) = \frac{1}{N_T} \left(\frac{\partial u_t}{\partial t} - D \frac{\partial^2 u_t}{\partial^2 x} - f(t, x) \right)^2 \tag{B.16}$$

for $u_t = u_t(t, x)$ and $N_T = N_t N_x$. Practically in the implementation, and to make this more general, the specific derivatives will be extracted from $\boldsymbol{J}$ and $\boldsymbol{H}$, such that the cost function to be minimized for the diffusion equation will be

$$C(z) = \frac{1}{N_T} \left(\boldsymbol{J}(u_t)_{(00)} - D\boldsymbol{H}(u_t)_{(11)} - f(t, x) \right)^2 \tag{B.17}$$

where subscript (00) indicates the first instance in the Jacobian matrix, and similarly for the Hessian with subscript (11). The gradient, $\nabla C(z)$, will be computed using automatic differentiation through the *Autograd*-package in Python (Maclaurin et al., 2015). More on that in Sec.III.A.

For the second approach, it is useful see the cost function as a sum of the contributions from the network approximation, $\hat{u}(t, \boldsymbol{x})$, for the collocation, boundary and initial points. Using the MSE, the individual contributions are then

$$C_r(t^r, \boldsymbol{x}^r) = \frac{1}{N_r} \sum_{i=0}^{N_r - 1} |\hat{R}(t_i^r, \boldsymbol{x}_i^r)|^2 \tag{B.18}$$

as the residual cost function,

$$C_b(t^b, \boldsymbol{x}^b) = \frac{1}{N_b} \sum_{i=0}^{N_b - 1} |\hat{u}(t_i^b, \boldsymbol{x}_i^b) - u_e(t_i^b, \boldsymbol{x}_i^b)|^2 \tag{B.19}$$

as the boundary cost function, and

$$C_0(t^0, \boldsymbol{x}^0) = \frac{1}{N_0} \sum_{i=0}^{N_0 - 1} |\hat{u}(t_i^0, \boldsymbol{x}_i^0) - u_e(t_i^0, \boldsymbol{x}_i^0)|^2 \tag{B.20}$$

as the initial cost function. The total error is then simply the sum

$$C(t, \boldsymbol{x}) = \sum_\alpha C_\alpha \quad \text{for } \alpha \in \{r, b, 0\} \tag{B.21}$$

The gradient of this function is computed as

$$\nabla C = \sum_\alpha \nabla C_\alpha \quad \text{for } \alpha \in \{r, b, 0\} \tag{B.22}$$

To compute the different gradients, the implementation uses the automatic differentiation available in TensorFlow (Abadi et al., 2016). Sec.III.A describes this in more detail.

Looking at the details of the two approaches, they essentially perform the same type of assembly and computations for setting up the input to the cost function. The differences lies on how the physics are incorporated into the optimization.

The first approach samples the collection of collocation points uniformly inside the domain, such that the boundary and initial points are integrated into these points. The second approach samples of the points using different random sampling methods. As NN are probabilistic by nature, this added stochasticity may aid the network in its predictions (Raissi et al., 2019).

4. Gradient descent

The gradient descent method used in this project is similar to the method described in Sec.II.A1 in (Bø, 2024). The following is a quick summary based on that report.

For the cost function $C(z)$, the gradient descent method seeks to find some optimal parameter $\boldsymbol{\theta}_*$ that minimizes the cost function $C(\boldsymbol{\theta})$. If the gradient if $C(\boldsymbol{\theta})$ is

$$\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \sum_{n=0}^{N-1} \nabla_{\boldsymbol{\theta}} C_n(\boldsymbol{\theta}) \tag{B.23}$$

for C_n as the cost for data point n, then the $(i+1)^{th}$ update to $\boldsymbol{\theta}$ is

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta_i \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_i) \tag{B.24}$$

for some learning rate, η_i. Iterating like this, should, as $i \to T$ for some large number T, make $||\boldsymbol{\theta}_{i+1} - \boldsymbol{\theta}_i|| \to 0$ (Hjorth-Jensen, 2023).

The implementation in this project uses the two gradients defined above. For the implementation based on (Lagaris et al., 1998), the parameters is stored in the parameter matrix P, s.t. the gradient descent update becomes

$$P_{e+1} = P_e - \eta_e \nabla_{P_e} C(z, P_e) \tag{B.25}$$

for some training step e. The implementation based on (Raissi et al., 2019) follows the basic update presented in

Eq.(B.24), with the gradient computed for the total cost function given in Eq.(B.22).

The general gradient descent method shown above illustrates the basic idea of the optimization and training of the network. The project will be using gradient descent methods that uses *adaptive learning rates*. These modify or replace η based on the system behavior during the training in an effort to improve the convergence rate of the gradient descent. Specifically, the project uses the methods **Adagrad, RMSprop** and **ADAM** in both the implementations. For a more in-depth description of these methods see App.C in (Bø, 2024).

5. Activation functions

Each layer in the network are linked through activation functions that introduces the necessary non-linearity to the system. They scale the input to the function, and gives a scalar output. This output is the next layer's input. The reason why it is important to have a non-linear connection between the layers in the network, is because the other parts of the network is a series of linear operations. Without some non-linearity, a deep network would be equivalent to a one-layer network, as all the linear operations of all hidden layers could be represented by one single layer (Bishop, 2006).

There are two effects to look put for when choosing activation functions for the hidden layer. The first is the so-called *vanishing gradients*. This is a phenomena that that may arise if $\sigma(z)$ have an asymptotic shape towards extreme values of z. Then the output from the gradients will diminish more and more. In turn this leads to little to no change in the update of the different layer's weights and biases during backpropagation, effectively killing the network training (Raschka et al., 2022).

The other effect is an opposite one, so-called *exploding gradients*. This may arise as the activation function amplifies the weights and biases more and more during training, making the backpropagation unstable. The worst case scenario is that the layer parameters will be forced to infinity. The training will thus diverge more and more from the target solution, never reaching the minimum the algorithm is seeking (Goodfellow et al., 2016).

Both these are common issues when implementing deep NN, as the depth tends to amplify the effect even more. This is because the backpropagation algorithm takes the information from the gradient of the cost at the network output and feeds it backwards towards the input. Additional steps in the backpropagation will then carry this issue through each operation. A remedy for the vanishing gradient problem can be changing the activation functions in the hidden layers. By avoiding function with asymptotic behavior at extreme values, this will ensure that the information from the output reaches the input (Goodfellow et al., 2016).

A popular way to deal with the exploding gradients is to introduce regularization in the gradient descent update. Penalizing large values in the parameters may dampen the exponential growth between the steps. Another approach is to set a upper limit on the gradient. This is known as *gradient clipping*, and will explicitly limit how large the gradients are allowed to become before using them in the gradient descent update (Goodfellow et al., 2016).

This project will use a number of different activation functions, with a preference on continuous activation functions, such as the *tanh*- and *GeLU*-functions. The expected behavior and shape of the PDE's will be informing the best possible choice of layer activation. A selection of the relevant functions are presented in App.B.

C. Finite-Difference method

Mainly, the *Finite Difference Method* (FDM) is a method that uses discretization of partial differential equations by approximating derivatives with finite differences based on Taylor's theorem. This section is based on the textbook (Langtangen and Linge, 2017) and the lecture notes from (Mortensen, 2024).

The method requires discrete domains, where the function value of a continuous function is defined at specific *mesh points*. For the time domain, $t \in [0, T]$, this is usually defined as

$$t_n = n\Delta t, \quad \text{for } n = 0, 1, \ldots, N_t, \ \Delta t = \frac{T}{N_t} \tag{C.1}$$

and the spatial domain, $(x, y) \in [0, L_x] \times [0, L_y]$,

$$\begin{aligned} x_i &= i\Delta x, \quad \text{for } i = 0, 1, \ldots, N_x, \ \Delta x = \frac{L_x}{N_x} \\ y_j &= j\Delta y, \quad \text{for } j = 0, 1, \ldots, N_y, \ \Delta y = \frac{L_y}{N_y} \end{aligned} \tag{C.2}$$

Here, $\Delta\alpha$, $\alpha \in \{t, x, y\}$ is the time-step, and spatial steps in x and y, respectively.

The method seeks to approximate the exact solution to the PDE, $u = u(t, x, y)$, by solving the discrete version of the PDE on the numerical mesh. This solution is called a mesh function, $u_{ij}^n = \mathbf{u}^n$. The goal of the method is then to approximation u with $\boldsymbol{u}^n$ s.t.

$$u_{ij}^n \approx u(t_n, x_i, y_j) \tag{C.3}$$

The approximation error, $|\boldsymbol{u}^n - u|$ is expected to tend towards zero for $\Delta\alpha \to 0$.

Using Taylor's theorem, an expansion around the function value of $u(x)$ at $x_{i\pm1}$,

$$u(x_{i\pm1}) = u(x_i \pm \Delta x) = u(x_i) \pm \Delta x u'(x_i) + \cdots \\ \cdots + \frac{\Delta x^2}{2} u''(x_i) + \mathcal{O}(\Delta x^3) \tag{C.4}$$

can give expressions for the 1st derivative as

$$\frac{du}{dx} = \frac{u(x_{i+1}) - u(x_i)}{\Delta x} + \mathcal{O}(\Delta x) \\ \frac{du}{dx} = \frac{u(x_i) - u(x_{i-1})}{\Delta x} + \mathcal{O}(\Delta x) \tag{C.5}$$

and using both version of the expansion, it is also possible to find the 2nd-order derivative

$$\frac{d^2u}{dx^2} = \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{\Delta x^2} + \mathcal{O}(\Delta x^2) \tag{C.6}$$

The same equations can be used for partial derivatives wrt. t and y as well. Related expressions for the derivatives that can be found by including more terms in the expansion, as well as taking the expansion around $x_{i\pm2}$ and also further "away" from i.

The discretization of a PDE following the FD-method is a straightforward substitution of the relevant numerical differentiation expressions given in Eq.(C.5), (C.6) into the PDE. How to reorder depends on the type of system, but as an example, lets consider the one-dimensional wave equation,

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = f(t, x) \tag{C.7}$$

for c as the wave speed, and $f(t, x)$ as a general source function. Substituting the expression from Eq.(C.6) for both partial derivatives gives, taking $u(t_n, x_i) \approx u_i^n$

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} \tag{C.8}$$

and

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta t^2} \tag{C.9}$$

Now, substituting this into the wave equation, and solving for the future time-step $n+1$ gives

$$u_i^{n+1} = 2u_i^n - u_i^{n-1} + C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + f(t_n, x_i) \tag{C.10}$$

with the vector $\boldsymbol{u}^{n+1}$ as the unknown, all other variables are assumed to be known, $C = (c\Delta x/\Delta t)$, the so-called *Courant number*, and $f(t_n, x_i)$ as the source function evaluated at the mesh points.

C will in this context serve as an important constraint to ensure numerical stability for the simulation. For the one-dimension wave equation, the requirement can be found as $C \leqslant 1$. It is common to fix the spatial discretization, and solve for the required time step to ensure stability, which is

$$\Delta t \leqslant \frac{\Delta x}{c} \tag{C.11}$$

1. Differentiation matrix

One last aspect of the implementation of this method for solving the PDE's used in this study is the *differentiation matrix*, which allows to use matrix-vector multiplication to lower the computational cost of looping through the spatial mesh. Defining the sparse matrix, $D_x^{(2)}$, based on the repeating pattern of the result from Eq.(C.9), give the second order differentiation matrix as

$$D_x^{(2)} = \frac{1}{\Delta x^2} \begin{bmatrix} 2 & -5 & 4 & -1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & \cdots & 0 & 1 & -2 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 & -2 & 1 \\ 0 & \cdots & 0 & -1 & 4 & -5 & 2 \end{bmatrix} \tag{C.12}$$

The first and last row look different, due to $\boldsymbol{u}_{-1}^n$ and $\boldsymbol{u}_{N_x+1}^n$ landing outside the boundary. Those values are found using a different discretization there allows for filling out the matrix with approximations of the same accuracy (Mortensen, 2024).

When used for specific specific PDE's, it is possible to include some types of boundary conditions. As an example, for zero Dirichlet BC's, the first and last row can be set to zero during initialization, which will handle the BC's for the entire simulation. A similar matrix exists for the first order derivative as well (Mortensen, 2024).

D. Model selection

The goal of the methods described in the previous sections is to obtain the best possible prediction of a systems behavior based on input parameters. The algorithms this project uses to create these predictions contain a multitude of parameters with various impacts on the model performance.

The notion of **model-selection** refers to using different methods to figure out the best possible parameters for a given dataset. Because of the goal and nature of the network model, the selection analysis is limited to so-called *grid search* method for finding

optimal hyperparameters. The specifics of the grid search method can be found in my report from project 2 (Bø, 2024). The following description will therefore only give a quick overview.

The general method can be described as a straightforward process of checking different combinations of network hyperparameters, and then evaluating their performance. For the PINN-implementations here, relevant parameters are learning rate, η, gradient regularization, λ, training epochs, e, network depth, d, and hidden layer size, M. The method is then to choose a set of different values for these parameters, and then run a training sequence with each possible combination of the different elements in these sets. Measure of the model performance using the different combinations can then be stored and evaluated (Raschka et al., 2022).

In practice, the method compares two hyperparameters while other model parameters are kept fixed. Picking the elements may be done in two ways, either sequentially "walking" through the elements from start to end, or picking combinations from a larger distribution of elements. The latter method is called a *randomized grid search*, and may increase the probability of picking a more optimal combination of parameters, as it does not limiting the search to sets of predefined elements (Raschka et al., 2022).

Results from the grid search analysis will be given in Sec.IV, and in App.A.

E. Model assessment

The term **model assessment** refers to how well model performs in the task it is built for, using quantitative metrics (Hastie et al., 2009). In the context of approximating solutions to PDEs, it is common to look at different measures that gives an indication to what degree the final prediction deviates from an analytical or valid test solution (Langtangen and Linge, 2017).

A common approach in the FDM-context is to use an error mesh function

$$\mathbf{e}^n = \boldsymbol{u}^n - \boldsymbol{u}_e^n \tag{E.1}$$

where $\boldsymbol{u}_e^n = u_e(t_n, \boldsymbol{x}_j)$ is the test solution, and $\boldsymbol{u}^n$ is the mesh function. The global error can then be quantified using the discrete ℓ^2-norm of $\mathbf{e}^n$, as

$$E = ||\mathbf{e}^n||_{\ell^2} = \left(\Delta t \sum_{n=0}^{N_t} (\mathbf{e}^n)^2\right)^{1/2} \tag{E.2}$$

This global error will then give the measure on the overall deviation from an exact solution using an equivalent norm to the continuous L^2-norm (Mortensen, 2024).

Similarly, the maximum global error can be computed using the ℓ_∞-norm, defined as

$$||\mathbf{e}^n||_{\ell^\infty} = \max_n |\mathbf{e}^n| \quad \text{for } n = 0, 1, \ldots, N_t \tag{E.3}$$

Another common assessment tool for numerical solutions to PDEs is the convergence rate. This looks at how the global error evolves with numerical mesh refinement. As convergence implies that

$$\mathbf{e}^n \to 0 \quad \text{as} \quad \Delta t \to 0$$

and similarly for $\Delta x, \Delta y$, it is possible to compute a convergence rate, r, by assuming the total error for discretization i can be computed as $E_i = C(\Delta t_i)^r$, s.t.

$$\frac{E_{i-1}}{E_i} = \frac{(\Delta t_{i-1})^r}{(\Delta t_i)^r} \tag{E.4}$$

Taking the logarithm of this yields

$$r = \frac{\log\left(\frac{E_{i-1}}{E_i}\right)}{\log\left(\frac{\Delta t_{i-1}}{\Delta t_i}\right)} \tag{E.5}$$

The ideal evolution of this convergence rate will have a slope of Δt^r, and it is common to visualize this in a log-log-plot, where this will appear linear (Langtangen and Linge, 2017).

These notions can be extended to the assessment of the PINN-approximation as well. The intention of the model solution is still the same as for other numerical approaches, i.e., the model should converge to the correct solution. As the training of NNs are statistical learning processes, we are approximating the solution by minimizing the cost functions defined in Eq.(B.13), (B.21). This changes the reasoning behind how to measure the error and convergence, but conclusions from (Kutyniok, 2022) and (Ryck et al., 2023), and the review (Cuomo et al., 2022) provides justifications for continuing with similar measures as for FDM, although the error norm is bounded by a slightly different upper bound, namely

$$||\boldsymbol{u}_e - \hat{\boldsymbol{u}}||_{L^2} \leqslant \left(C\hat{R}(\hat{\boldsymbol{u}}) + \mathcal{O}\left(N^{-1/d}\right)\right)^{\frac{1}{2}} \tag{E.6}$$

with N as the number of collocation points, and d as the depth of the network[1] (Cuomo et al., 2022).

Adapting that bound to this project, it will be assumed that the ℓ^2-norm is a valid measure of global error, and convergence of the network to the analytical solution will

[1] The details are outside the scope of this study and will not be presented here

be assumed to follow

$$||\boldsymbol{u}_e - \hat{\boldsymbol{u}}_{(\kappa)}||_{\ell^2} \to 0 \quad \text{as} \quad \kappa \to \infty$$

for κ as the number of coefficients in the network. In this context, κ will be dependent as the number of training points and the network width (Cuomo et al., 2022).

Finally, for the purposes of visualizing the details of the deviations, the absolute error between the approximations and the analytical solutions can be computed as

$$\mathbf{e}_{\text{abs}}(t, \boldsymbol{x}) = |\boldsymbol{u}_e - \hat{\boldsymbol{u}}|$$

This provides a function to plot in a contour plot. This kind of plot is a useful tool to visually see where in the domain the approximation is struggling to give an accurate replication of the target function.

III. CODES AND CODE DESCRIPTION

A. Program and class structure

This project organizes its programs in **Jupyter-notebooks** that runs the Python programs, and also allows for text, formulas and comments to be added to the code using a markup language called *Markdown* (Community, 2021).

The general notebook structure is:

1. Imports from supporting files and external packages
2. Setting notebook defaults like figure parameters and random seed default
3. Definition of problem, setting up the domain, and constants for the PDE to solve
4. Initialization of the main class, like the neural network, with all necessary instance variables and functions
5. Setup of supporting classes and methods
6. Main calculations, parameter study, like a grid search, and figure plots.
7. Comparison with exact solution and other solution methods

Three different neural network classes was created for this project, one following the approach described in chapter 15 in (Hjorth-Jensen, 2023) in one spatial dimension, lets denote that the *Plain*-PINN, and two following the approach described in (Raissi et al., 2019), one for one spatial dimension, and one for two spatial dimensions, denoted the *TensorFlow*-PINN, or TF-PINN.

The class initialization, domain setup and layer creation are kept as similar as the approaches allow. The network training methods are more specialized, as the two approaches differ in the setup, mostly due to the structure and methods available in TensorFlow. The two-dimensional class inherits all relevant methods from the one-dimensional class.

Both implementations uses automatic differentiation to compute the necessary derivatives during the network training. The Autograd-approach follows the algorithm presented in Alg.1.

Algorithm 1 General algorithm for training using the Autograd-approach from (Hjorth-Jensen, 2023)

```
Create layer structure
for all training epochs e do
  for all (t_n, x_i) in n = 0, 1, ..., N − 1 do
    for all (t_n, x_i) in i = 0, 1, ..., N − 1 do
      Feed-forward pass for (t_n, x_i)
      Compute J(u_t(t_n, x_i)), H(u_t(t_n, x_i))
      Compute error and cost for current (t_n, x_i)
    end for
  end for
  Return accumulated cost
  for all layers in P do
    Compute P_e = P_{e−1} + ∇C(z, P_e)
  end for
end for
Compute final cost and network output with update P
```

This approach is, unfortunately, very slow, especially if N, as the number of points in the discrete domain, increases. The reason is the double for-loop over all points t_n, x_i. Extending to two spatial dimension was therefore abandoned for this project, because it made the runtime unpractical.

The training algorithm for the TensorFlow-approach is very similar to the one showed above, but it uses the specific methods and classes available through the package. Most notably is the elimination of the looping over points in the domain, reducing the runtime drastically.

The hidden layers are set up using the **Dense**-class, making the layers fully-connected. The **GradientTape**-class is used extensively to record and store information when computing the terms for the residual function, and other gradients in the optimization. For the gradient descent, the implementation uses the available classes **Adagrad, RMSprop** and **ADAM** from TensorFlow.

Both approaches includes evaluation- and plotting-methods, computing the error between the network output and the analytical solution to the PDE, the global error, and the additional metrics defined in Sec.II.E. The plotting gives, in addition to plots of the actual shape of the network solution, figures displaying the absolute

difference contours and the solution to visually assess the network prediction.

All final versions of the program files containing these classes and methods, as well as a selection of notebooks, can be found in the public GitHub-repository created for this project[2]. Here, the reader will also find a description to how to access and run the different notebooks that performs the same kind of analysis presented in this report.

B. Analysis algorithms

The following algorithms presents the general grid search procedure used in the model selection, and the marching algorithm used by the FD-solvers. Note that the grid search algorithm describes a standard grid search, which requires a double loop over predefined parameter-lists.

The random grid search uses only one loop, randomly picking the values for **param1** and **param2** in the beginning of the loop. The parameter-list in the random search will then be a predefined range of values to pick the random value between. All random distribution used are uniform.

Algorithm 2 General grid search algorithm for hyperparameters η, λ or epochs, e, using the TF-PINN

```
Setting up the physics problem and domain arrays
Create list/distribution of values to draw param1, param2 from
Create storage array(s) for computed measure(s)
for all values in param1-list do
    for all values in param2-list do
        Initialize network with grid search parameters and other fixed parameters
        Network training
        Compute measure(s) and store in array(s)
    end for
end for
Create heatmap plot with the measure-scores for each combination of param's
```

The marching algorithm in Alg.3 describes the specific algorithm for the two-dimensional wave equation. The solvers for the PDEs uses similar versions, where the differences are due to the specific stencil from the discretization.

[2]Repository can be found here: `https://github.com/andersthorstadboe/project-3-fys-stk4155-pinn-for-pdes`

Algorithm 3 Marching algorithm used for solving the two-dimensional wave equation using the FDM

```
Define initial conditions, u_0(x,y), v_0(x,y)
Initialize solution storage arrays U^{n+1}, U^n, U^{n-1}
Setting up and modifying the D^2_α-matrices
Compute initial conditions and applying them to U^0, U^1
for n = 1, 2, ..., N_t do
    Compute U^{n+1}
    Apply boundary conditions to U^{n+1}
    Recast U^{n-1} ← U^n, U^n ← U^{n+1}
    Store current solution U^{n-1} for analysis
end for
```

IV. RESULTS

This section will present a selection of results with the aim of showing the model performance, and comparing it to the more traditional finite-difference computation of the same PDEs. These solutions acts as a benchmark for the solutions from the networks.

The goal is also to show how the different choices of hyperparameters and network structure impacts the PINNs predictions. It will start with a short overview of the simulation parameters, and continue with simulation results for the PDEs in both one and two dimensions. A selection of results will be included in this section. Additional figures from the model selection and simulations are also included in App.A.

A. Simulation parameters

This first part details the solutions to the PDEs in one spatial dimension using the initial and boundary conditions detailed in Sec.II.A, and the trail and residual function discussed in Sec.II.B.2. Tab.IV.1 shows the physical parameters used in the simulations. The same parameters will be used in the two-dimensional case as well. Tab.IV.2 shows the input variables and parameters used in the runs for the two different implementations. The same domain was used for both PDEs.

B. One-dimensional PDEs

This section will focus on the results for the diffusion equation, and a comparison between the two different PINN-implementations.

Attempting to improve the performance of the networks, both regular and random grid searches were performed. Fig.IV.1 shows a standard grid search comparing the learning rate, η to the regularization, λ, and Fig.IV.2 shows the results from comparing the learning rate, η, and the number of epochs, e in a random

TABLE IV.1: Physical parameters used in the simulations

Variable	Value	Comment
D	0.1 - 1.0	Diff. coeff.
c	0.1 - 1.5	Wave speed
A	0.1 - 2.0	Amplitude
m	1 - 3	Wave length
k	$m\pi$	Wave number
ω	kc	Wave dispersion coeff.
ρ	$D\pi^2(L_x^{-2} + L_y^{-2})$	Decay rate

grid search. The optimal combination for this specific case is marked with a star.

Similar studies were done comparing the performance between the parameters in Tab.IV.2, which resulted in picking the optimal parameters shown in Tab.IV.3.

Using those parameters in the simulations, the following shows a search for what activation function to use. Tab.IV.4 shows the resulting mean global error, E, computed for different runs using the different activation functions. It shows that the ReLU-function are not suited for the task, while most of the others perform well, especially for the TensorFlow-implementation. The continuous functions, like *Sigmoid* and *tanh* seems to work best for the Plain-PINN, and the TF-implementations seems to be impacted less by the choice of the continuous functions, but the SiLU-function clearly performs best.

The following solutions to the 1D diffusion equation uses the *tanh* as activation function. The reason is to have a setup that makes the Plain-PINN more comparable to the other solvers. The analytical solution, the network solutions, and the FD-solution to the diffusion equation in one dimension can be seen in Fig.IV.3, IV.4, IV.5, IV.6, using lower values for the

TABLE IV.2: Input variables and parameters for the two implementations of PINNs

Variable	Plain PINN	TF-PINN
Time-span	$(0, T]$	$(0, T]$
Domain (x, y)	$[0, 1] \times [0, 1]$	$[0, 1] \times [0, 1]$
Grid (Nx, Ny)	(10, 10)	(100, 100)
Learn. rate	$\eta \in [10^{-3}, 10^{-1}]$	$\eta \in [10^{-3}, 10^{-1}]$
Reg.parameter	$\lambda \in [10^{-9}, 10^{-7}]$	$\lambda \in [10^{-9}, 10^{-7}]$
Layer width	$M \in [2, 50]$	$M \in [2, 50]$
Network depth	$d \in [1, 6]$	$d \in [1, 6]$
Epochs	$e \in [100, 500]$	$e \in [100, 4000]$

TABLE IV.3: An overview of the optimal values for the hyperparameters for the two implementations of PINNs.

Variable	Plain PINN	TF-PINN
Learn. rate	$\eta = 2 \cdot 10^{-2}$	$\eta = 5 \cdot 10^{-2}$
Reg.parameter	$\lambda = 1 \cdot 10^{-9}$	$\lambda = 4 \cdot 10^{-8}$
Layer width	$M = \{2, 4\}$	$M = 2$
Network depth	$d = \{\}$	$d = \{\}$
Epochs	$e = 200$	$e = 2500$

TABLE IV.4: The global error, E, for the solution to the 1D diffusion equation for the PINNs using different activation functions. The FD scored an $E = 0.00247$.

Activation func.	Plain-PINN	TF-PINN
Sigmoid	0.03996	0.01847
tanh	0.02622	0.02065
ReLU	0.03819	0.23740
ELU	0.35155	0.01419
SiLU	0.51996	0.00223
GELU	0.10289	0.01071

physical parameters D and A, as this gave more stable solutions.

The PINNs gave output here showing a good correspondence with the exact solution. The mean absolute difference and global error is shown in Tab.IV.5. It shows This shows that the FD-solver is performing best, with TF-PINN as a close second.

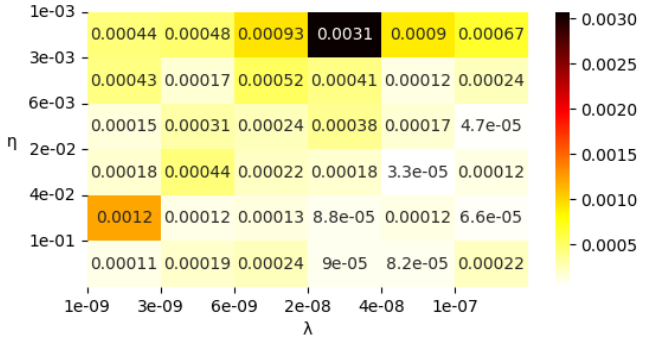

FIG. IV.1: Showing cost results for a standard grid search for η, λ using the range indicated in Tab.IV.2. It indicates that the optimal range lies around $\eta = 2 \cdot 10^{-2}$, $\lambda = 4 \cdot 10^{-8}$

TABLE IV.5: Showing the mean absolute difference and global error for the solution to the 1D-diffusion equation, comparing the performance of the PINNs and FD-solver.

Solver	**Mean abs.diff**	**Global error** E
Plain-PINN	0.02123	0.02622
TF-PINN	0.00993	0.01105
FD	0.00201	0.00247

TABLE IV.6: Optimal hyperparameters found from the grid search analysis running the TF-PINN against the 2D wave function.

Variable	**TF-PINN**
Learn. rate	$\eta = 4 \cdot 10^{-2}$
Reg.parameter	$\lambda = 2 \cdot 10^{-8}$
Layer width	$M = \{16, 20\}$
Network depth	$d = \{3, 4\}$
Epochs	$e = 2545$

C. Two-dimensional PDEs

The results for the two-dimensional PDEs will focus on the wave equation simulations. It should be noted that as before, the physical parameters were chosen in the lower part of the range, as this provided more stability in the analysis for the two-dimensional cases as well. The grid search analysis resulted in the optimal parameters shown in Tab.IV.6.

This shows that the parameters found are fairly similar to the ones from the one-dimensional simulations. Using these parameters, and the activation functions

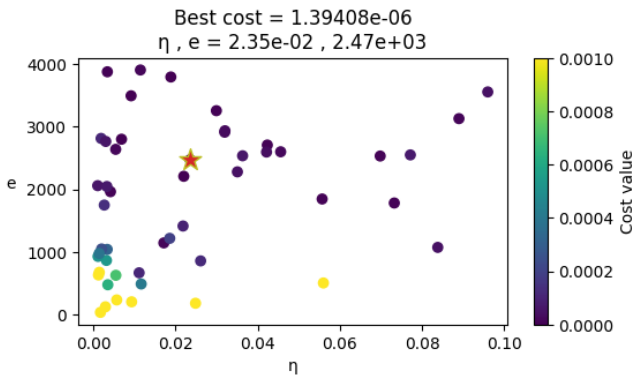

FIG. IV.2: Showing cost results for a randomized grid search for η, e using the range indicated in Tab.IV.2. It indicates that the optimal combination lies at $\eta = 3.5 \cdot 10^{-2}$, $e = 2470$, indicated by the star

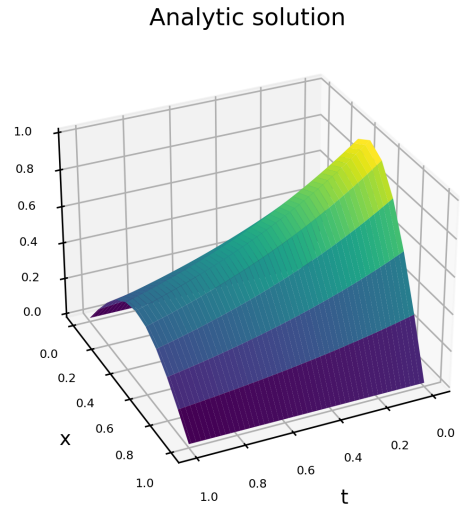

FIG. IV.3: The analytical solution for the diffusion equation

that seemed to be favored by the TF-PINN in the one dimensional cases, similar runs were done for the 2D diffusion and wave equations. Tab.IV.7 shows a comparison between three activation functions and the resulting errors when simulating the 2D wave equation.

From this, it seems that the network is still favoring the GELU-activation function in the hidden layers. Moving on with that function, and the parameters found in the grid search, the analytical and network solution to the wave equation at end of the simulation time can be seen in Fig.IV.8, IV.7 and the FD-solution in Fig.IV.10.

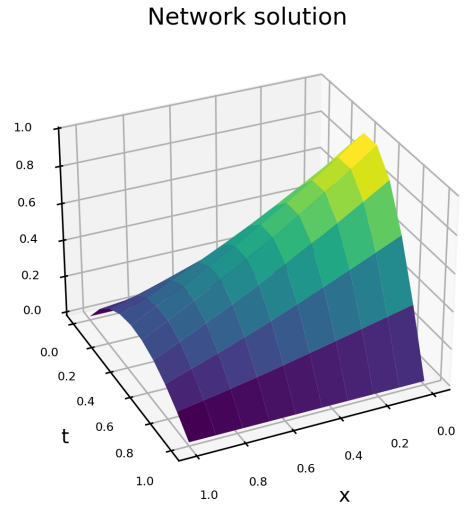

FIG. IV.4: The Plain-PINN prediction to the 1D diffusion equation. This shows the entire prediction $\hat{u}(t, x)$, i.e. the entire time duration of the simulation. The global error for this specific run was $E = 0.02622$.

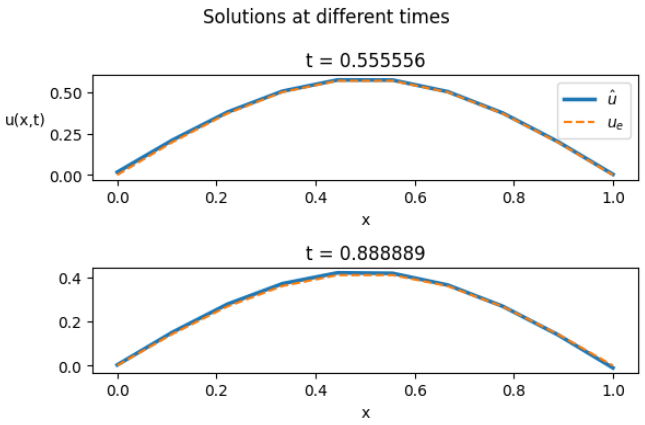

FIG. IV.5: The TF-PINN prediction to the diffusion equation, showing two slices at specific times. The overall global error was $E = 0.01105$

The network is clearly struggling more in the two-dimensional case to find a satisfactory solution. It is also suspected that the initial and boundary conditions are not defined correctly in the implementation, making the target the network seeks different from the FD-solution.

The final best performance of the network against the 2D wave equation is presented in Tab.IV.8, comparing it to the FD-solution. Together with the figures showing the results from these simulations, it is clear that something is wrong with the network implementation. The same issues was also found when attempting to find a solution to the diffusion equation in 2D.

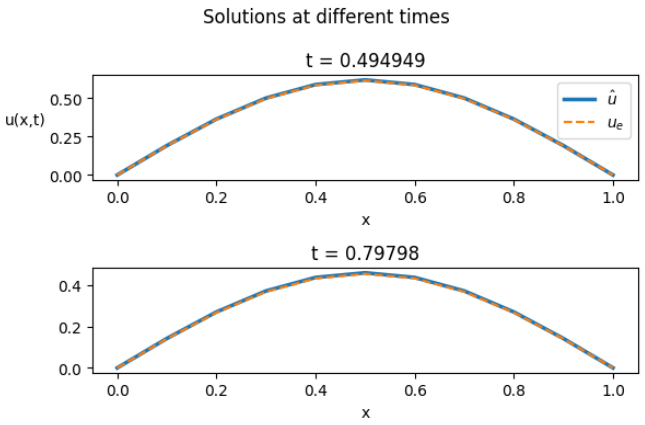

FIG. IV.6: The FD-solution to the diffusion equation, same as in Fig.IV.5. The difference in time comes from the difference in sampling. The overall global error was $E = 0.00247$

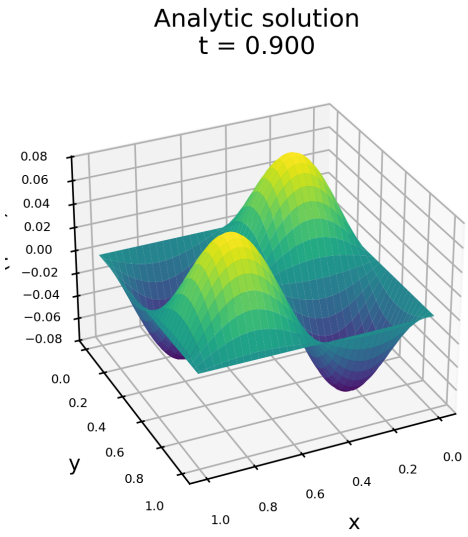

FIG. IV.7: The analytical solution to the 2D wave equation used in this project, with the wave number, $(m_x, m_y) = [2, 2]$.

TABLE IV.8: Showing the mean absolute difference and global error for the solution to the wave equation in two spatial dimensions, comparing the performance of the the TF-PINN and FD-solver.

Solver	**Mean abs.diff**	**Global error E**
TF-PINN	0.04139	0.35431
FD	0.00035	0.00045

TABLE IV.7: Showing a comparison between three different activation functions using the TF-PINN to solve the 2D wave equation. The FD-solver manage an global error of $E = 0.00036$ for this setup.

Solver	**Mean abs.diff**	**Global error E**
tanh	0.04139	0.51245
SiLU	0.04462	0.54307
GELU	0.02852	0.35431

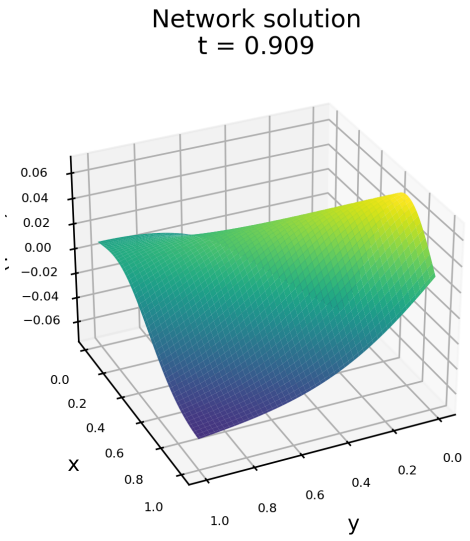

FIG. IV.8: The TF-PINN prediction for the wave equation in two spatial dimensions. This shows the prediction $\hat{u}(t, x, y)$ at $t \approx 0.9 \cdot T$. The error for this particular run was $E = 0.44075$

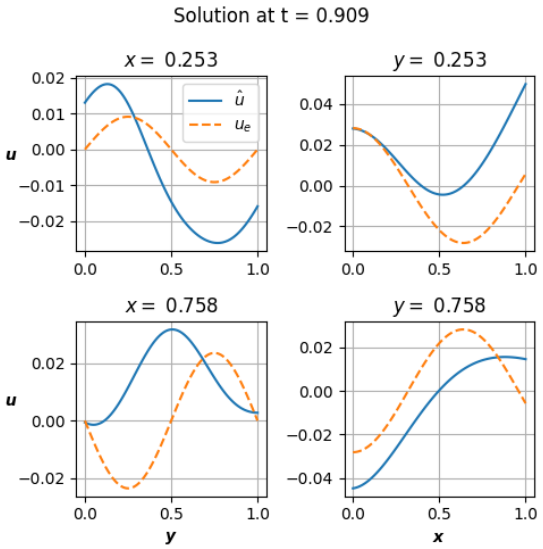

FIG. IV.9: The TF-PINN prediction for the wave equation at $t \approx 0.9 \cdot T$ for different values of x and y.

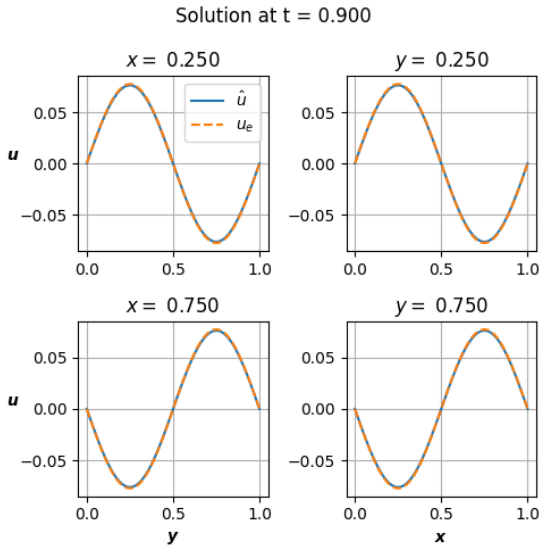

FIG. IV.10: The FD-solution for the for the wave equation at $t = 0.9 \cdot T$ for different values of x and y, with $E = 0.00036$

V. DISCUSSION

This discussion will go through the major findings from the results, and give notes on the general model performance, hyperparameters, activation functions and other network configuration choices. There will also be a final section in the discussion commenting on the applicability of the methods in this project in solving PDEs.

A. Model performance

From the model assessment in the previous section, the PINNs solving the unsteady, one-dimensional equations perform comparably with the benchmarking method, and they were able to give satisfactory solutions. Is must be reiterated that it required the physical parameters to be lowered to below 1.0. One reason for why that helped, may be that it makes the range between the maximum and minimum values of the target function smaller. This, in turn, creates a simpler landscape for the network to reproduce.

A smaller range between the max and min-values of the target function also makes some of the metrics used in the analysis more forgiving, as they are measures of the distance from the target. It does not seem that the global error and mean absolute difference metrics suffered too much from this effect. It was observed that the impact on the performance of the FD-solvers were not changed too much by changing the physical parameters, so this builds confidence in the metrics.

The solutions produced for the two-dimensional cases did not fare as well as the 1D ones. The network struggled to find good solutions to the equations, and did not perform well compared to the benchmark. One of the main reasons for this may lay in the implementation, where it seems to be issues with how the boundary and initial values are enforced during the training. This in essential to how the network searches for the solution, and errors here will impact the performance. Boundary conditions and initial conditions is equally important for other numerical methods, like the FDM, for solving PDEs, so this just highlights that getting that wrong will give wrong solutions.

B. Model parameters

Neural networks introduce a lot of hyperparameters into the implementations, and studying how they impact the performance was found to be just as important for the PINNs as for the FFNN in project 2. The nature of the PINN makes tools like the cross-validation and bootstrapping used in the other project irrelevant. But the grid searches, both standard and random, proved helpful in determining the optimal hyperparameters.

What was interesting, was that the values did not change very much between the two different implementations, and between the two different PDE-cases. From the figures shown in the results, it seems clear that it is the learning rate that has the most effect on performance, with the regularization parameter as a close second. The simulations required a fairly low λ-value, which means that there is a need to penalize the weights and biases in the hidden layers to a certain degree. This is usually a indicator that the process, if left untended, give parameters in the layers that dominate a lot.

The network depth, i.e. the number of hidden layers, was one hyperparameter that need to be different between the two PINNs. The Plain-PINN performed better with a deeper network, with the final optimal value at $d = 5$. A reason for the difference may be because the Plain-implementation iterated over every collocation point, performing a full feed-forward and backpropagation pass per point. As the backpropagation passes the information from the output cost back towards the input through the gradient, the incremental approach might benefit from more backpropagation steps.

C. Layer activation functions

Using TensorFlow for one of the network-implementations opened up the opportunity to be introduced to some new activation functions. The ReLU-function seems to be popular in a lot of applications. Using some of the continuous functions inspired by the ReLU, like the SiLU and GELU, gave good results in this study. The discontinuous functions, like the ReLU and ELU performed and was not used in the final simulations.

It should be noted that the implementations of these two done by the author did not perform well when used together with the Plain-PINN. Using them through the TF-implementation made them work a lot better, so it might be something in how they were programmed that made them less usable. The Plain-implementation computes the derivative of the activation functions using *Autograd*, which may also impacts the gradient update in unintended ways.

D. Method suitability

From the discussion above, and the results presented, it becomes clear that using neural networks for solving partial differential equations is a viable approach. The PINNs implemented here reproduces the solutions to the

equations well, especially in the one dimensional case, where both approaches manage to produce comparable results to finite difference solvers.

One of the major advantages of the NN-approach is that the program and classes become very general purpose, with little need for modification of the structure of the network when trying to solve new problems, or extend to more variables. The only need for change is to the input to the methods, like a new analytical function, boundary and initial conditions, and the residual function to using in the modeling.

Another advantage is that the domain becomes mesh-free, allowing for freely picking mesh-points in the domain. The traditional numerical approaches, like FDM and FEM, is often very sensitive to how the mesh is chosen, specially for the distances between spatial grid points and time-step length. Choosing wrong here when simulating unsteady problems usually leads to unstable solvers (Mortensen, 2024).

With the mesh-free network domains randomly sampling the collocation points used in the simulation, it leads to a more general application across several different domains, which is a major advantage to other methods. This also allows for using the trained model to extrapolating the solution outside the original domain. This generative ability may be very useful, and does not exist when using other methods.

The randomness of the NN-approach may also be considered a disadvantage compared to other numerical methods. The reason is that the results may be less reproducible, especially during the implementation and troubleshooting of the programs. This was at least the experience during this project, where it was sometimes hard to understand if the issues were linked to the implementation or due to the randomness in the network. It may be that the random state of the program was not set in the correct way, such that it changed from test to test.

This next disadvantage is somewhat linked to the point in the paragraph above. The TF-PINN uses TensorFlow exclusively for the network part of the classes. It has a lot of functionality, and is a well-tested software, but it introduces a layer of abstractness that is not present when using for example FDM to solve PDEs. This is more or less true for a lot of applications that uses machine learning, as it creates various degrees of black boxes. Being able to know the inner workings of the program solving the equation for you is a major advantage, but this is made harder when using neural networks.

Comparing the performance of the networks to the benchmark method, it should be noted that it is hard to beat the performance of a FD-solver on the problems used in this project. The TF-PINN came close in the one-dimensional cases, but when comparing the runtime, the FDM were much faster than any of the networks. For linear PDEs with constant coefficients, it is no problem for a FD-solver to quickly converge to the analytical solution with fairly large time-steps, especially with one spatial dimension(Langtangen and Linge, 2017).

That said, the PINNs may come out much better if used to model non-linear PDEs, where FD-solvers need a lot of modifications to adapt the time-steps and perform intermediate convergence iterations to be stable. Here, the PINN-approach's generality and independence from a mesh, is a huge advantage.

VI. CONCLUSIONS

In summary, lets revisit some of the main parts and finding of the report to give some closing remarks with a critical view on the project.

The main goal of the project was the implementations of general purpose physics-informed neural networks (PINNs) that was able to find solutions to unsteady, linear partial differential equations in one and two spatial dimensions. The project also uses four different finite difference-solvers to benchmark the numerical solutions, as well as comparing the results to specific analytical solutions to the equations.

The main trend is that the two different implementations performs well for the one-dimensional wave and diffusion equation, with comparable results to FD-solutions. The global errors for the diffusion equation was $E = \{0.02622,\ 0.01105,\ 0.00247\}$ for the Plain-PINN, TF-PINN and FD-solver, respectively. The physical parameters, like the diffusion coefficient, wave speed, and amplitudes, had to be kept low, e.g. between $0.1 - 0.5$, to achieve this performance.

The Plain-PINN also seem to prefer the continuous activation functions, like *tanh* or *sigmoid*, to the *ReLU*-family of functions. The network implemented using TensorFlow had a bigger selection of activation functions to pick from, and here the SiLU or GELU-functions seemed to give the best performance.

The networks struggled more on the two-dimensional cases, and due to runtime, the TF-PINN was the only one used here. It still manages to reproduce the solution to the wave equation to a certain degree, but not nearly as well as the FD-solver. Comparing the two, the TF-PINN and FD-solver managed global errors $E = \{0.35431,\ \}$, respectively. It is suspected that there are issues with implementation of the boundary and initial conditions that the network require to function properly.

A. Perspectives and Improvements

Looking ahead, there are a couple of improvements and extensions that can be looked into. Firstly, it would be worthwhile to compare the performance of the TF-PINN to other numerical methods when trying to model non-linear differential equations. Such a study may highlight the advantages of a PINN-approach more than this project managed.

An issue that the project faced was how the analytical solution, boundary conditions and initial conditions was implemented and used in the network. Reworking the implementation into something that is cleaner may highlight the issues, and create more comparable results, especially if the equation depends on more than two variables.

Lastly, adjusting the layer initialization and collocation sampling may be something to look further into. TensorFlow has a lot of options when creating the network structure, and exploring this should be a fairly straightforward task. The same holds for picking the collocation points, which was done uniformly for all points. Changing this approach for the initial and boundary points may allow the network to better satisfy the conditions then it did here.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Mathew Gardner, Geoffrey Irving, Michael Isard, Yangqing Jia, Rahul Kaul, Quoc V. Le, Md. S. A. M. N., Josh Susskind, Xiaoqiang Zhang, and Yuanzhong Xu. Tensorflow: Large-scale machine learning on heterogeneous systems. *arXiv preprint arXiv:1603.04467*, 2016. URL `https://arxiv.org/abs/1603.04467`.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, New York, 1st edition, 2006. ISBN 978-0-387-31073-2.

Anders Thorstad Bø. Using feed-forward neural networks for linear regression and classification - project 2 - fys-stk4155. Written as part of course FYS-STK4155, delivered 04.11.2024, 2024. URL `https://github.com/andersthorstadboe/project-2-fys-stk4155-ffneural-network`.

James Chasnov. Solution of the diffusion equation, 2024. URL `https://math.libretexts.org/Bookshelves/Differential_Equations/Differential_Equations_(Chasnov)/09%3A_Partial_Differential_Equations/9.05%3A_Solution_of_the_Diffusion_Equation`. Accessed: 2024-11-11.

Executable Books Community. Jupyter book, feb 2021. URL `https://doi.org/10.5281/zenodo.4539666`.

Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics–informed neural networks: Where we are and what's next. *J. Sci. Comput.*, 92(3), 2022. ISSN 0885-7474. doi:10.1007/s10915-022-01939-z. URL `https://doi.org/10.1007/s10915-022-01939-z`.

Lawrence C. Evans. *Partial Differential Equations*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 2nd edition, 2010. ISBN 978-0-8218-4974-3.

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

T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, 2009. ISBN 9780387848846. URL `https://books.google.no/books?id=eBSgoAEACAAJ`.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023. URL `https://arxiv.org/abs/1606.08415`.

Morten Hjorth-Jensen. Introduction to machine learning. `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html`, 2023. Accessed: 2024-09-27.

Gitta Kutyniok. The mathematics of artificial intelligence. In *Proceedings of the International Congress of Mathematicians 2022*, pages 5118–5139. European Mathematical Society, 2022. doi:10.4171/ICM2022/141. URL `https://ems.press/books/standalone/279/5593`.

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.

Hans Petter Langtangen and Svein Linge. *Finite Difference Computing with PDEs: A Modern Software Approach*. Texts in Computational Science and Engineering. Springer Nature, Cham, 2017. ISBN 978-3-319-55456-3. doi: 10.1007/978-3-319-55456-3.

Tom L. Lindstrøm. *Spaces: An Introduction to Real Analysis*, volume 29 of *Pure and Applied Undergraduate Texts*. American Mathematical Society, 2017. ISBN 978-1-4704-1100-4.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, page 5, 2015.

Wolfram MathWorld. Bernoulli distribution. `https://mathworld.wolfram.com/BernoulliDistribution.html`, 2024. Accessed: 2024-11-30.

Mikael Mortensen. Lecture notes for matmek-4270. `https://matmek-4270.github.io/matmek4270-book/intro.html`, 2024. Accessed: 2024-11-08.

Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. In *International Conference on Learning Representations (ICLR)*, 2018. URL `https://arxiv.org/abs/1710.05941`.

Sebastian Raschka, Yuxi (Hayden) Liu, and Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-Learn*. Packt Publishing, 2022.

Tim De Ryck, Ameya D. Jagtap, and Siddhartha Mishra. Error estimates for physics-informed neural networks approximating the navier-stokes equations. *IMA Journal of Numerical Analysis*, 44(1):83–109, 2023. doi:

10.1093/imanum/drac060. URL https://academic.oup.com/imajna/article/44/1/83/6989836.

Aslak Tveito and Ragnar Winther. *Introduction to Partial Differential Equations: A Computational Approach*. Springer, 2nd edition, 2005. ISBN 978-3-540-24233-0. doi: 10.1007/b138962.

Appendix A: Grid search results

This section provides some additional results from the grid search analysis that was performed during model selection.

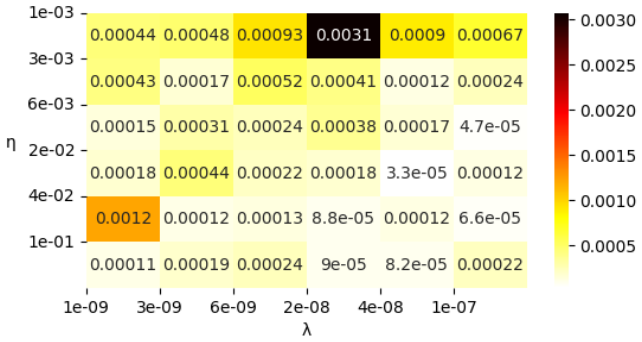

FIG. A.1: Caption

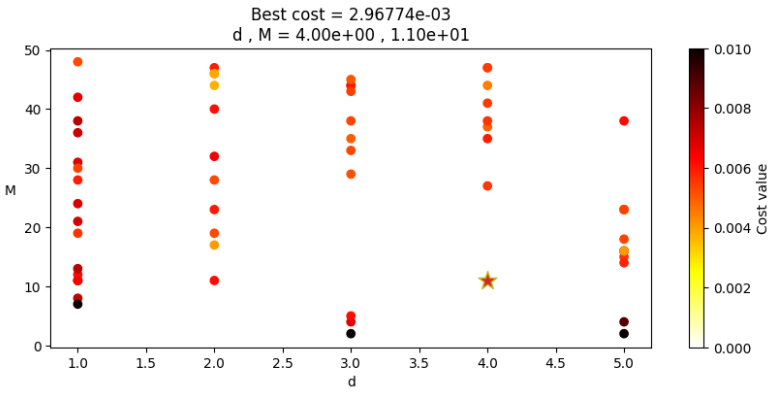

FIG. A.2: Caption

Appendix B: Details on activation functions

This section will give some additional details on the activation functions used in this project. All derivatives of these functions were computed using the *Autograd*-package, or through the TensorFlow-methods used.

The Sigmoid-function

$$\sigma_S = \frac{1}{1+\mathrm{e}^{-z}} \tag{0.1}$$

The Hyperbolic tangent-function

$$\tanh(z) = \frac{\mathrm{e}^{2z}-1}{\mathrm{e}^{2z}+1} \tag{0.2}$$

The ReLU-function

$$\sigma_{\mathrm{R}}(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leqslant 0 \end{cases} \tag{0.3}$$

The Exponential LU-function, $\alpha \geqslant 0$

$$\sigma_{\mathrm{E}}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha(\mathrm{e}^{z}-1) & \text{if } z \leqslant 0 \end{cases} \tag{0.4}$$

Sigmoid LU-function(Ramachandran et al., 2018)

$$\sigma_{\mathrm{Si}} = z \cdot \sigma_S(\beta z) = \frac{\beta z}{1+\mathrm{e}^{-\beta z}} \tag{0.5}$$

A usual choice for the parameter is $\beta = 1$.

The Gaussian Error LU-function(Hendrycks

and Gimpel, 2023)

$$\sigma_G(z) = z \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right]$$

$$\sigma_G(z) \approx \frac{z}{2}\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(z + 0.044715z^3)\right]\right) \quad (0.6)$$

$$\sigma_G(z) \approx \sigma_{\text{Si}}(\beta z) \text{ for } \beta = 1.702$$

which comes from computing the cumulative distribution of a Gaussian. The approximations is needed for computer implementation, and the SiLU-approximation usually gives a faster program. The two last ones are clearly related, as the GELU can be approximated with a specific choice of β. This also relates them to the Sigmoid-function, and they both tries to recreate the advantages of the LU-functions, using continuous functions (Hendrycks and Gimpel, 2023; Ramachandran et al., 2018)

Appendix C: Trail functions used in the modeling

This provides details on the two trail functions used in the Plain-PINN. That approach was limited to one spatial dimension due to challenges with the runtime of the implementation. The trail functions are used to assemble the cost function, and needs to satisfy the initial and boundary conditions of the PDEs they attempt to model.

For the variables $(t, x) = (0, T] \times [0, L_x]$, the diffusion equation is

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0 \quad (0.1)$$

with the analytical solution as

$$u_e(t, x) = a_0 \sin\left(\frac{\pi x}{L_x}\right) \text{e}^{-\rho t}, \quad \rho = D\frac{\pi^2}{L_x^2}$$

The initial condition are taken as

$$u(0, x) = u_0(x) = u_e(0, x) = a_0 \sin\left(\frac{\pi x}{L_x}\right)$$

and with boundary conditions

$$u(t, 0) = g_0(t) = u_e(t, 0) = 0$$

$$u(t, L_x) = g_1(t) = u_e(t, L_x) = 0$$

Assembling the trail function then gives, with $t_* = t/T$

$$A(t, x) = (1 - t_*)\left[u_0(x) - \left(\left(1 - \frac{x}{L}\right)u_0(0) + \frac{x}{L}u_0(L)\right)\right]$$

$$A(t, x) = (1 - t_*)u_0(x) \quad (0.2)$$

and

$$B(t, x, \mathcal{N}) = t_*\left(1 - \frac{x}{L}\right)\frac{x}{L}\,\mathcal{N}(t, x, P) \quad (0.3)$$

Then the trail function for the network to model the diffusion equation becomes

$$u_t(t, x, \mathcal{N}) = (1 - \frac{t}{T})u_0(x) + t\left(\frac{x}{L} - \frac{x^2}{L^2}\right)\mathcal{N}(t, x, P) \quad (0.4)$$

Taking the domain as $(t, x) = (0, T] \times [0, L_x]$, the wave equation becomes

$$\frac{\partial^2 u}{\partial t^2} - c^2\frac{\partial^2 u}{\partial x^2} = 0 \quad (0.5)$$

with the analytical solution as

$$u_e(t, x) = a_0 \sin\left(\frac{k_x x}{L_x}\right)\cos(\omega t) \quad (0.6)$$

with $k_x = m_x\pi$ as the wave number, and $\omega = k_x c$. The initial conditions are

$$u(0, x) = u_0(x) = u_e(0, x) = a_0 \sin\left(\frac{k_x x}{L_x}\right)$$
$$\frac{\partial u}{\partial t}|_{t=0} = \frac{\partial u_e}{\partial t}|_{t=0} = v_0(x) = \frac{a_0 k_x}{L_x}\cos\left(\frac{k_x x}{L_x}\right) \quad (0.7)$$

and boundary conditions as

$$u(t, 0) = g_0(t) = f(t, 0) = 0$$
$$u(t, L_x) = g_1(t) = f(t, L_x) = 0 \quad (0.8)$$

Then, assembling the trail function then gives

$$A(t, x) = (1 - t_*^2)\Big[u_0(x) - \left(\left(1 - \frac{x}{L}\right)u_0(0) + \frac{x}{L}u_0(L)\right)\Big] + t_*\, v_0(x) \quad (0.9)$$

$$A(t, x) = (1 - t_*^2)u_0(x) + t_*\, v_0(x)$$

and

$$B(t,x,\mathcal{N}) = t_*^2\left(1-\frac{x}{L_x}\right)\frac{x}{L_x}\,\mathcal{N}(t,x,P) \tag{0.10}$$

giving the trail function for the wave equation as

$$u_t(t,x,\mathcal{N}) = (1-t_*^2)u_0(x) + t_*\; v_0(x) + \\ +t_*^2\left(\frac{x}{L}-\frac{x^2}{L^2}\right)\,\mathcal{N}(t,x,P) \tag{0.11}$$

Appendix D: FDM discretization of equations

The focus of the project has not been on the finite difference method specifically, but some background was given in the main body of the report. All discretization of the PDEs follows the methods outlined in the lecture notes in (Mortensen, 2024). The final form of the four versions will be stated here for completeness, but the details will be omitted.

1. Diffusion equation

The discretization of the one-dimensional diffusion equation was done using a forward difference stencil in time and central difference stencil in space, yielding the equation

$$\boldsymbol{u}^{n+1} = (\boldsymbol{I} + C_d D^{(2)})\boldsymbol{u}^n + f(t_n, x_i) \tag{1.1}$$

for $\boldsymbol{u}^n = (u_i^n)_{i=0}^{N_x}$, $C_d = (D\Delta t)/\Delta x^2$, the Courant number, and $D^{(2)}$ as a second order differentiation matrix, similar to the one in Eq.(C.12).

For two dimensions in space, using the same stencils are used, which gives

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^n + D\Delta t\left(D_x^{(2)}\boldsymbol{U}^n + \boldsymbol{U}^n(D_y^{(2)})^T\right) + \\ +f(t_n, x_i, y_j) \tag{1.2}$$

with

$$D_\alpha^{(2)} = (\Delta\alpha^2)^{-1}D^{(2)},\ \alpha = \{x, y\}$$

and

$$\boldsymbol{U}^n = (U_{ij}^n)_{i,j=0}^{(N_x,N_y)}$$

2. Wave equation

Similarly to the diffusion equation, the wave equations can be discretized using central differences in space. Since this equation also has a second order derivative wrt. time, central differences are used for that term as well. In one dimension, this gives the equation

$$\boldsymbol{u}^{n+1} = 2\boldsymbol{u}^n - \boldsymbol{u}^{n-1} + C_w^2 D^{(2)}\boldsymbol{u}^n + f(t_n, x_i) \tag{2.1}$$

for $C_w = ((c\Delta t)/\Delta x)^2$ as the Courant number.

In two spatial dimensions, using the same stencils, the final equation to solve is

$$\boldsymbol{U}^{n+1} = 2\boldsymbol{U}^n - \boldsymbol{U}^{n-1} + \\ +(c\Delta t)^2\left[D_x^{(2)}\boldsymbol{U}^n + \boldsymbol{U}^n(D_y^{(2)})^T\right] + f(t_n, x_i, y_j) \tag{2.2}$$

with the same $D_\alpha^{(2)}$ and $\boldsymbol{U}^n$ as for the diffusion equation. Solutions to all of these equation are obtained by a marching algorithm, iterating over the N_t time-steps.