Solving partial differential equations with neural networks
FYS-STK4155 - Project 3

Ingvild Olden Bjerkelund & Jenny Guldvog
University of Oslo
(Dated: December 19, 2025)

Physics-informed neural networks (PINNs) have recently emerged as a promising alternative to classical numerical methods for solving partial differential equations by embedding physical laws directly into the training process. In this project, we investigate the performance of PINNs for solving the one-dimensional heat equation and systematically compare them with both the analytical solution and a classical finite-difference scheme based on the Forward-Time Central-Space (FTCS) method. Two independent PINN implementations are developed using automatic differentiation with Autograd and the PyTorch framework, respectively, and their performance is evaluated in terms of convergence behaviour, accuracy, and computational efficiency. We further examine the influence of grid resolution, optimization strategy, activation functions, and network architecture through an extensive hyperparameter study. Our results show that the PyTorch-based implementation combined with stochastic gradient descent and the Adam optimizer provides the most stable and computationally efficient PINN training. Among the activation functions considered, sigmoid yields the lowest training cost and physical error, with tanh closely following, while ReLU performs significantly worse for this smooth problem. Despite careful tuning, the PINN approach exhibits higher error and substantially higher computational cost than the FTCS scheme for the one-dimensional heat equation. We conclude that, for this low-dimensional problem with a known analytical solution, the classical FTCS method remains superior in both accuracy and efficiency, while PINNs may be more suitable for higher-dimensional or more complex problems where traditional grid-based methods become impractical.

I. INTRODUCTION

Differential equations play a fundamental role in physics as they describe how physical quantities change in space and time. Many of the most important laws in physics are expressed as differential equations, i.e. Newton's second law. Another prominent example is the heat (or diffusion) equation. It is a fundamental equation, as it models heat transfer, particle transport, molecular diffusion in fluids, and it possesses a well-known analytic solution, making it an ideal example for verification of the neural network [1–3].

Traditionally differential equations are solved numerically with finite difference methods, acknowledged as one of the most effective approaches [e.g., 4]. One of these schemes is the Forward-Time Central-Space. In this approach, temporal derivatives are approximated using forward differences, while spatial derivatives are approximated using central differences, resulting in an explicit time-stepping scheme for the diffusion equation.

With recent advances in machine learning, a new class of deep learning methods has emerged for solving differential equations, known as Physics-Informed Neural Networks (PINNs). Compared to conventional neural networks, PINNs are forced to satisfy the physical laws by incorporating the physics directly into the training process.

In this project we want to investigate whether neural networks can replace the numerical methods in execution time and in accuracy. It is also worth noting that in the finite element case the solution is not expressed in closed analytical form as in the neural case, and additional interpolations are required in order to find the solution at an arbitrary point in the domain [5].

The theoretical motivation for using neural networks in this context is provided by the universal approximation theorem, which states that a multilayer perceptron with one hidden layer can approximate any sufficiently regular function to arbitrary accuracy [6, § 4]. This suggests that neural networks can, in principle, represent differential equations, as previously shown in [e.g. 5]. However, standard neural networks trained without physical constraints may produce solutions that are physically inconsistent or violate conservation laws. PINN on the other hand, enforce explicitly the network to satisfy physical conditions [7, 8].

PINNs are a class of deep learning algorithms that integrate the physical aspect, making them more robust in physically consistent predictions. As shown in box 2 in [8], we can have three different methods to introduce the appropriate physics; observational, inductive, and learning biases. These biases help guide the training towards physically consistent solutions.

In this project we will use inductive biases to introduce the physics. This involves introducing prior assumptions, such as initial and boundary conditions, which ensure that the predictions satisfy the physical laws. This can be done by creating a trial function, which is constructed to satisfy the physical conditions, while also leaving one part with freedom through the neural network. The result is then the construction of a solution written in a differentiable, closed analytical form [5].

Early work by [5] demonstrated the potential of using neural networks to solve differential equations. They also compared their approach with traditional numeri-

cal methods, such as the Galerkin finite element method [GFEM; e.g., 9]. They found that the neural method could achieve superior accuracy, and that the accuracy of the neural method was controlled by increasing the number of hidden layers. In terms of execution time, they showed that the time, normalized by the number of parameters increased linearly for the neural method, while increasing almost quadratically for the finite element method [5]. This shows that the potential of the neural method lies in dealing with a large number of parameters.

The finite element solution employed by [5] was very accurate at training points, and sometimes outperforming the neural method. However, its accuracy at interpolation points is orders of magnitude worse. By contrast, the neural method yields excellent and comparable accuracy at both training and interpolation points. Also note the FEM accuracy degrades on coarser grids.

In this project, we investigate whether neural networks, and in particular PINNs, can be effectively applied to solve the one-dimensional heat equation. We begin by implementing two PINNs, one using only `Autograd` and one including `PyTorch`. We also tested three different minimization methods, in terms of accuracy and execution time. To test the accuracy, we implement the analytical solution. Using this, the best PINN implementation with the best minimization method was found. Then the impact of PINN architecture is systematically explored, in terms of activation functions and network depth and width. Finally, we compare the best PINN implementation with the classical FTCS scheme to assess and compare accuracy, stability, and computational efficiency. Our goal is to evaluate whether PINN offer a practical advantage over traditional numerical approaches for this problem.

The remainder of this report is structured as follows. We first introduce the one-dimensional heat equation, including its analytical solution and the FTCS finite-difference scheme. This is followed by a presentation of the theoretical framework underlying the physics-informed neural network (PINN) approach. The results are then presented and discussed, before the report concludes with a summary and outlook. All code developed for this work is available in our GitHub repository.[1]

II. METHODS

A. One-Dimensional Heat Equation

The partial differential equation considered in this project is the one-dimensional heat equation. In this setting, the physical problem describes the evolution of the temperature distribution in a homogeneous rod of length $L = 1$, defined on the spatial domain $x \in [0, L]$ [10]. For simplicity, we restrict the spatial domain to $x \in [0, 1]$ and the temporal domain to $t \in [0, 1]$, thereby avoiding additional scaling of the data later.

The differential equation we want to solve is given by

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \tag{1}$$

or, equivalently, $u_{xx} = u_t$, with the initial and boundary conditions. The initial conditions at $t = 0$ are set to $u(x, 0) = \sin(\pi x)$, and the boundary conditions are homogeneous Dirichlet, given by $u(0, t) = u(L, t) = 0$ for $t \geq 0$.

1. *Analytical solution*

The closed-form analytical solution of the one-dimensional heat equation, subject to the specifications given in the previous section, is given by [11, § 10.2.4]:

$$u(x,t) = \sin(\pi x)\, e^{-\pi^2 t}. \tag{2}$$

The complete derivation is provided in Appendix A. This solution is used as a reference to validate the numerical methods considered in the following sections.

2. *Finite difference solution*

Finite difference methods are among the most established and widely used approaches for the numerical solution of partial differential equations [e.g., 4]. As such, they provide a natural baseline to assess the performance of physics-informed neural networks (PINNs), both in terms of accuracy and computational efficiency. These methods rely on explicit discretizations of the spatial and temporal domains, leading to a variety of numerical schemes depending on the choice of finite-difference approximations.

In this work, we employ the Forward-Time Central-Space (FTCS) scheme as a classical reference method. The FTCS scheme is simple to implement, computationally efficient for low-dimensional problems, and well suited for smooth solutions such as those of the one-dimensional heat equation considered here.

To solve Eq. (1) using FTCS, we follow the discretization described in [11, § 10.2.1]. The full derivation can be found in Appendix B, and the final update formula can be written in the compact form

$$u_{i,j+1} = \alpha u_{i-1,j} + (1 - 2\alpha) u_{i,j} + \alpha u_{i+1,j}, \tag{3}$$

where $\alpha = \Delta t / \Delta x^2$. The FTCS scheme requires $\alpha \leq 1/2$ for stability [11, § 10.2.1].

[1] https://github.com/ingooboo/FYS-STK-Project-3

The FTCS method is first-order accurate in time and second-order accurate in space, resulting in a local truncation error of [11, § 10.2.1]

$$\mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2). \tag{4}$$

The FTCS scheme therefore provides a suitable classical reference for evaluating the accuracy and computational efficiency of the PINN solutions presented in this work.

B. Physics-Informed Neural Network with Inductive Biases

To investigate the potential of neural networks for solving partial differential equations, we employ PINN, in which physics is informed by introducing inductive biases [7]. In this section, we present the theoretical background and methodology underlying this approach.

Following the formulation introduced [5], we consider a general differential equation of the form:

$$G(\vec{x}, \Psi(\vec{x}), \nabla\Psi(\vec{x}), \nabla^2\Psi(\vec{x})) = 0, \qquad \vec{x} \in D \tag{5}$$

where $\vec{x} = (x_1, ..., x_n) \in R^n$ denotes the spatial and/or temporal coordinates, $D \in R^n$ is the computational domain, and $\Psi(\vec{x})$ is the unknown solution. The differential equation is also supplemented by boundary and initial conditions.

As a first step, we apply the collocation method [5], in which the domain D and its boundary S are discretized into finite sets of collocation points. This results in the following system of equations:

$$G(\vec{x_i}, \Psi(\vec{x_i}), \nabla\Psi(\vec{x_i}), \nabla^2\Psi(\vec{x_i})) = 0, \qquad \forall \vec{x_i} \in D \tag{6}$$

1. The Trial function

To approximate the solution of the differential equation, we introduce a trial function with adjustable parameters, denoted by $\Psi_t(\vec{x}, \vec{p})$. The problem in Eq. 6 is then transformed as an unconstrained minimization problem:

$$\min_{\vec{p}} \sum_{\vec{x_i} \in D} \left[G(\vec{x_i}, \Psi_t(\vec{x_i}, \vec{p}), \nabla\Psi_t(\vec{x_i}, \vec{p}), \nabla^2\Psi_t(\vec{x_i}, \vec{p})) \right]^2 \tag{7}$$

where the parameters $\vec{p}$ correspond to the weights and biases of a feedforward neural network (FFNN).

The trial function is constructed such that it satisfies the initial and boundary conditions of the differential equation by design. This is achieved by decomposing it into two terms,

$$\Psi_t(\vec{x}) = A(\vec{x}) + F(\vec{x}, N(\vec{x}, \vec{p})) \tag{8}$$

where $A(\vec{x})$ is fixed function that satisfies the prescribed initial and boundary conditions, and $N(\vec{x}, \vec{p})$ is a single-output FFNN. The function F is constructed such that it vanishes on the boundaries and at the initial time, ensuring that the neural network contribution does not violate the constraints.

With this construction, the enforcement of initial and boundary conditions is separated from the minimization process, resulting in an unconstrained minimization problem that is easier to handle.

In our case, the physical problem is the temperature gradient in a rod, and the differential equation we want to solve is given in eq. 1. The trial function is given as

$$\Psi_t(x, t; \theta) = A(x, t) + B(x, t) \cdot h_2(x, t; \theta) \tag{9}$$

Here $h_2(x, t; \theta)$ is the neural network, with parameters θ. $A(x, t)$ is the initial condition u_0, and defined as

$$A(x, t) = \sin(\pi x), \tag{10}$$

and the B-term ensures that the conditions are applied to the neural network. To enforce the boundary conditions for all $t \geq 0$, the neural network term is further multiplied by a factor that vanishes at $x = 0$ and $x = L$. Choosing

$$B = x \cdot (1 - x) \cdot t, \tag{11}$$

the final trial function then becomes:

$$\Psi_t(x, t; \theta) = \sin(\pi x) + x \cdot (1 - x) \cdot t \cdot h_2(x, t; \theta). \tag{12}$$

2. Feed Forward Neural Network

The expressive power of neural networks is supported by the universal approximation theorem, which states that a neural network with a single hidden layer and a sufficient number of neurons can approximate any continuous function on a compact domain to arbitrary degree of accuracy [6, § 4]. By systematically combining simple nonlinear transformations in deeper networks, complex functions can be represented step by step in multiple hidden layers.

A FFNN is a class of artificial neural networks composed of an input layer, one or more hidden layers, and an output layer. Each layer consists of neurons that perform affine transformations of their inputs, followed by nonlinear activation functions [6, 11–13]. Training is performed by adjusting weights and biases, using algorithms such as backpropagation to minimize a defined cost function [6].

In a FFNN, information propagates strictly in one direction, from the input layer through the hidden layers to the output layer, without feedback connections [6, 14]. While more specialized architectures exists, such as convolutional neural networks (CNNs), and recurrent neural networks (RNNs), allowing feedback connections [14], FFNNs provide a flexible and well-understood framework that is well suited for function approximation in PINN. For a more detailed description of FFNN, the is reader is referred to an earlier project [15].

3. Activation functions

Activation functions determine how individual neurons transform their inputs within a neural network and are essential for enabling the representation of nonlinear relationships. In the absence of nonlinear activation functions, a neural network would reduce to a composition of linear mappings, regardless of its depth, and would therefore be equivalent to a linear regression model. Nonlinear activation functions are thus fundamental for allowing neural networks to model complex and hierarchical structures [13, § 12].

In this project, three commonly used activation functions are considered: the sigmoid function, the tanh function, and the rectified linear unit (ReLU). These functions are selected due to their different characteristics with respect to nonlinearity, smoothness, and gradient propagation. As the cost function contains the derivatives of the trial function, it is desirable that the activation functions are at least twice differentiable.

a. Sigmoid Function

The sigmoid activation function closely resemble how biological neurons behave [13]. Here the inputs is squeezed between 0 and 1, and the output could be interpreted as the probability that the neuron becomes activated. However, for input values with large magnitude, the sigmoid function saturates, leading to very small gradients. As a consequence, the sigmoid activation function is susceptible to the vanishing gradient problem [13, § 12], in which gradients diminish during backpropagation, resulting in slow and potentially ineffective learning. This causes inefficient learning and risk of getting stuck in local minima.

Despite this, the sigmoid function is smooth and twice-differentiable, making it suitable for our problem, where we have derivatives inside the cost function.

The sigmoid function is defined as

$$\sigma(z) = \frac{1}{1+e^{-z}}, \tag{13}$$

with the derivative

$$\frac{d\sigma(z)}{dz} = \sigma(z)\big(1-\sigma(z)\big). \tag{14}$$

b. Tanh Function

The hyperbolic tangent activation function is a smooth, zero-centered function, with output values ranging from -1 to 1. Compared to the sigmoid function, this zero-centered property generally leads to improved gradient flow during training and can partially mitigate vanishing gradient effects. The tanh function is therefore frequently employed in PINN as it is infinitely differentiable and well suited for problems involving higher-order derivatives. Due to its zero-centered nature, it is common practice to scale the input data such that it is also approximately zero-centreed.

The tanh function is defined as

$$\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \tag{15}$$

with the derivative

$$\frac{d}{dz}\tanh(z) = 1-\tanh^2(z) \tag{16}$$

c. ReLU Function

The rectified linear unit (ReLU) is a widely used activation function that mitigates the vanishing gradient problem for positive input values. In this regime, the function is linear, which results in non-vanishing gradients and typically faster convergence during training. However, ReLU remains nonlinear overall and is capable of representing complex functions, it is associated with the so-called dead neuron problem [11, §§ 12 & 17]. This phenomenon occurs when neurons repeatedly receive non-positive inputs, causing their gradients to vanish and effectively preventing further learning.

Moreover, the ReLU function is only piecewise linear and not smoothly differentiable, and not differentiable at $z = 0$, which can be disadvantageous in PINN where derivatives of the network output enter directly into the cost function.

The ReLU function is defined as

$$a = \mathrm{ReLU}(z) = \max(0, z). \tag{17}$$

with the derivative

$$\frac{da}{dz} = \begin{cases} 0, & \text{if } z < 0, \\ 1, & \text{if } z > 0\ . \end{cases} \tag{18}$$

4. Cost function

Again following the methodology of [5], the minimization of the trial function in Eq. 7 is considered a procedure of training the neural network, where the objective is to drive the residual function, $G(\vec{x_i})$, to zero at a set of collocation points.

Compared to conventional training, where the cost function typically measures the mean squared error between network predictions and labeled training data, no explicit target data are available in the present setting. Instead, the network is trained to minimize the $G(\vec{x_i})$. Therefore, the computation of the cost function involves not only the network output, but also its derivatives with respect to the inputs.

Returning to the one-dimensional heat equation, (Eq. 1), we define the cost function as the squared residual

of the PDE (the difference between the LHS and RHS), and substituting the trail function $\Psi_t = \Psi$ into the PDE equation yields the cost function:

$$C(x,t;\hat{\theta}) = \left(\frac{\partial\Psi(x,t;\theta)}{\partial t} - \frac{\partial^2\Psi(x,t;\theta)}{\partial x^2}\right)^2 = (\Psi_t(x,t;\theta) - \Psi_{xx}(x,t;\theta))^2 \tag{19}$$

where $\Psi_t = \partial\Psi/\partial t$ and $\Psi_{xx} = \partial^2\Psi/\partial x^2$.

It is important to note that the cost function depends solely on the neural-network-based trial solution, which is constructed to satisfy the initial and boundary conditions by design. As a result, no explicit training dataset is required, and there is no need to partition data into training and validation sets in the traditional sense.

5. *Minimization methods*

The training of PINN involves minimizing the cost function, or the PDE residual squared. This results in a highly non-convex optimization problem. In particular, the nonlinearity of the neural network architecture implies that the associated cost function does not satisfy the conditions for convexity, and therefore lacks guarantees of convergence to a global minimum [7, 12, Sec. 4.3].

However, gradient-based optimization methods are widely used for training neural networks and have been shown to perform well in practice on such non-convex objectives. Consequently, standard first-order optimizers based on gradient descent are employed to minimize the PINN cost, aiming to efficiently reduce the cost function and constraint violations rather than to guarantee convergence to the global optimum [12, § 5.9].

We implement three different minimization methods: plain gradient descent, gradient descent with Adam, and stochastic gradient descent with Adam.

The first optimization method is simple gradient decent [14], where the parameter update for each iteration is given by

$$\theta_{t+1} = \theta_t - \eta\nabla f(\theta_t). \tag{20}$$

Here $\nabla f(\theta)$ is the gradient of the objective function $f(\theta)$, and η is the learning rate.

To speed up the training and help mitigate falling into local minima, we also implement Adam, or Adaptive Moment Estimation, which combines the advantages of Momentum and RMSProp [16]. In this method the parameter update for each iteration is given by:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(\theta_t) \tag{21}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla f(\theta_t)^2. \tag{22}$$

Where the final update is given by:

$$\theta_{t+1} = \theta_t - \frac{\eta m_t}{\sqrt{v_t + \epsilon}}. \tag{23}$$

Here, the first moment is the mean (m_t), and the second moment is the variance (v_t) of the gradients, and the β terms define how much influence the previous gradients have on the learning rate.

Stochastic gradient descent (SGD), only handles a subset of the data per iteration, where data corresponds to collocation points, making it a more memory efficient method [e.g., 6, 12, 13]. If one assume that the data can be written as a sum over n data points, the cost function and the derivative of it can thus also be written as a sum over n data points:

$$\nabla_\theta C(\theta) = \sum_{i=0}^{n-1} \nabla_\theta c_i(\boldsymbol{x_i}, \theta). \tag{24}$$

By choosing random subsets of size M from a dataset of n samples, randomness is introduced, and the number of subsets is $\frac{n}{M}$, where each subset is drawn with replacement

$$\theta_{j+1} = \theta_j - \eta_j \sum_{i\in B_k}^{n} \nabla_\theta c_i(\boldsymbol{x_i}, \theta). \tag{25}$$

III. IMPLEMENTATION

A. Numerical solution using the FTCS scheme

Algorithm 1 summarises the FTCS scheme for the one-dimensional heat equation. The spatial second derivative is approximated using central finite differences, while time integration is performed explicitly with a forward step. Homogeneous Dirichlet boundary conditions are imposed at each timestep, and stability is ensured by requiring $\Delta t/\Delta x^2 \leq 1/2$.

Algorithm 1 FTCS scheme for the 1D heat equation

procedure FTCS($N_x, N_t, \Delta x, \Delta t$)
  $\alpha \leftarrow \Delta t/\Delta x^2$ ▷ Stabilizing criteria
  Initialize spatial grid and initial condition
  for $i = 0, 1, \ldots, N_x$ **do**
    $x_i \leftarrow i\Delta x$
    $u_i^0 \leftarrow \sin(\pi x_i)$ ▷ Initial condition $u(x,0)$

  Time stepping
  for $j = 0, 1, \ldots, N_t - 1$ **do**
    $u_0^j \leftarrow 0$ ▷ Left boundary $u(0,t) = 0$
    $u_{N_x}^j \leftarrow 0$ ▷ Right boundary $u(1,t) = 0$
    for $i = 1, 2, \ldots, N_x - 1$ **do**
      $u_i^{j+1} \leftarrow u_i^j + \alpha\left(u_{i+1}^j - 2u_i^j + u_{i-1}^j\right)$ ▷ Update interior points

B. Physical informed neural network

1. Set up a trial function that satisfies the initial and boundary conditions.

2. Set up the cost function, which is the difference between LHS and RHS squared.

3. Select an activation function, optimization method and hyperparameters.

4. Gradients are calculated with automatic differentiation, instead of symbolic, using the automatic gradient library `Autograd` [17].

5. The feed-forward neural network is built using `Autograd` [17] and `PyTorch` [18].

1. Adam

For the optimization runs employing the Adam optimizer, we do not perform an explicit hyperparameter search over the algorithm parameters β_1, β_2, and ϵ. These parameters were investigated in earlier projects [15, 19], where suitable values were identified. Based on those findings, we adopt the standard parameter choices $\beta_1 = 0.9$, $\beta_2 = 0.99$, and $\epsilon = 10^{-8}$, corresponding to the default settings in `PyTorch` [18].

2. Stochastic gradient decent

For the stochastic gradient descent method, the batch size is chosen to be 1.2% of the total training dataset, corresponding to 10 samples for a grid resolution of $N_x = 30$.

3. Automatic differentiation

Compared to earlier projects, [15], an additional computational challenge arises from the necessity to evaluate first- and second-order derivatives of the trial function at each optimization step. Automatic differentiation therefore plays a central role in the implementation, enabling efficient and accurate computation of these derivatives and significantly reducing the computational power during training. All of the gradients are computed using the implementation of `Autograd`, which computed the gradients by tracing the sequence of elementary operations performed by the program to build a computational graph, and then applying the chain rule to that graph to obtain derivatives automatically [17].

4. Seed

To improve reproducibility we set a pseudo-random seed used by all components. Note that `NumPy` and `PyTorch` implement different RNG algorithms and weight-initialization routines. Therefore, the same integer seed does not necessarily produce identical initial parameters or exactly identical training dynamics across the two frameworks. To obtain a fair comparison we selected a seed that yields comparable initial costs in both implementations. Empirically we found that using seed = 36 produces similar initial loss values for both the `Autograd`-based and `PyTorch`-based PINN implementations; this seed was used for all paired comparisons reported in the paper.

C. Software

Our implementation is written in Python and uses several open-source libraries. For automatic differentiation in our own PINN implementation, we use `Autograd` [17]. For the `PyTorch`-based implementation, we use `PyTorch` [18]. Numerical computations are performed with NumPy [20], and figures are generated using Matplotlib [21]. For data handling and visualization, we use Pandas [22].

D. Use of AI tools

Artificial Intelligence (AI) tools, namely ChatGPT [23] and GPT UiO [24], has been used for parts of this project. The usage has been limited and primarily used for debugging code, formatting of figures, making descriptive documentation strings for functions, and grammatical help when writing the report. All usage have been quality checked by the authors.

IV. RESULTS AND DISCUSSION

A. Problem setup and evaluation metrics

In this section, we analyze the performance of PINNs for solving the one-dimensional heat equation introduced in Section II. The equation is solved on the spatial domain and temporal domain $x, t \in [0, 1]$, with homogeneous Dirichlet boundary conditions and the initial condition $u(x, 0) = \sin(\pi x)$, which admits a known analytical solution. The training of the PINN is performed by minimizing a physics-informed cost function, defined as the mean squared residual of the governing partial differential equation. The residual is evaluated using a trial function that incorporates the initial and boundary conditions, ensuring that these constraints are satisfied throughout training. It is important to note that a low physics-informed training loss does not necessarily imply a low physical error with respect to the analytical solution, as the loss only measures PDE residuals at the collocation points.

B. Testing PINN implementations

To assess the impact of implementation choices, we first compare our own PINN implementation based on automatic differentiation using `Autograd` with an equivalent implementation using the `PyTorch` library. In both cases, the governing PDE, trial function (Eq. 12), and cost function (Eq. 19) are identical, allowing for a controlled comparison of the gradient computation and optimization methods. For this initial comparison, we employ a shallow network consisting of two hidden layers with two neurons per layer and the `tanh` activation function. The spatial and temporal grid resolutions are set to $N_x = N_t = 10$, while all other hyperparameters are kept fixed. For both implementations (`Autograd` and `PyTorch`) we evaluate three optimizers—plain gradient descent (gd), gradient descent with Adam (gd-Adam), and stochastic gradient descent with Adam (sgd-Adam), across four learning rates: $0.1, 0.01, 0.001$, and 0.0001.

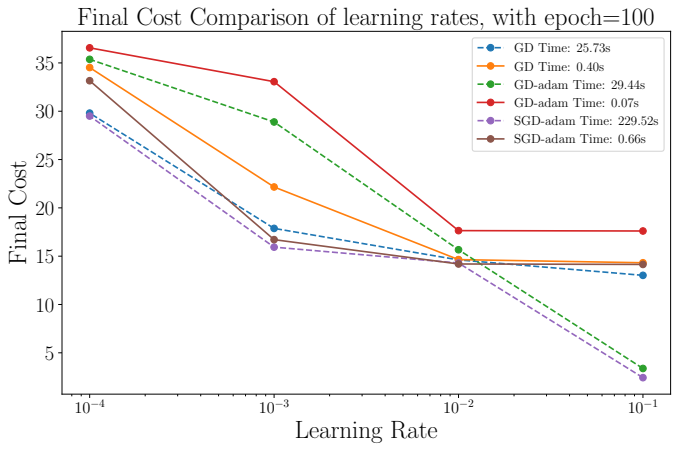


Figure 1: Final training cost as a function of the learning rate for different optimization strategies, evaluated after 100 epochs. Results are shown for gradient descent (GD), gradient descent with Adam, and stochastic gradient descent with Adam (SGD–Adam) implemented with both Autograd (dashed) and PyTorch (solid). The legend reports the corresponding wall-clock training times for each configuration.

At the highest learning rate, $lr = 0.1$ (see Figure 7 in Appendix D), the lowest final cost is obtained using the `Autograd` implementation with SGD–Adam, where the cost drops below 5 after approximately 25 epochs. The `Autograd` GD–Adam configuration also reaches a final cost below 5, but only after nearly 100 epochs. In contrast, none of the `PyTorch` configurations reproduce this behaviour: both GD and SGD–Adam converge rapidly to a higher final cost of approximately 15, while GD–Adam stabilizes at an even larger value of around 18. Plain GD in `Autograd` similarly converges to a final cost of roughly 15.

A qualitative difference between the two implementations is evident in the training dynamics. The `PyTorch` loss curves exhibit smooth and monotonic convergence, with only minor oscillations, whereas the `Autograd` loss histories at $lr = 0.1$ are highly irregular, characterized by pronounced spikes and abrupt jumps.

At a moderate learning rate of $lr = 0.01$ (see Figure 8 in Appendix D), all optimizers and both implementations converge smoothly to a similar final cost of approximately 15. At this learning rate, SGD–Adam performs best for both `Autograd` and `PyTorch`, reaching the final cost after roughly 25 epochs. For both implementations, plain GD slightly outperforms GD–Adam, with `Autograd` showing a marginally lower final cost than `PyTorch`. Notably, the instabilities observed at $lr = 0.1$ are absent at this learning rate.

Execution-time measurements reveal a substantial difference between the two implementations. As indicated in Figure 1, the `Autograd` implementation consistently requires significantly longer wall-clock time than the corresponding `PyTorch` runs across all optimizers and learning rates.

Taken together, these experiments reveal two clear trends. First, at moderate learning rates, `PyTorch` offers more stability, reproducibility, and computational efficiency: all optimizers converge smoothly to similar final cost, with SGD–Adam providing the fastest convergence. Second, while `Autograd` occasionally attains a substantially lower final cost at an aggressive learning rate ($lr = 0.1$), this behaviour is accompanied by highly unstable training dynamics and is not reproduced by the `PyTorch` implementation [18]. Consequently, it remains unclear whether this represents a robust advantage or an isolated outcome sensitive to stochastic effects. One possible explanation for this could be `PyTorch`'s highly optimized backends and batching strategies that tend to promote stable convergence. The large execution-time for `Autograd` further reduces its practicality as a primary implementation unless the better cost can be reliably reproduced and explained.

Based on these findings, all subsequent experiments are performed using the `PyTorch` implementation with the SGD–Adam optimizer and a learning rate of 0.01. This configuration provides a favorable balance between computational efficiency, smooth and reproducible convergence. For cases where convergence is slower, training is extended to 100–200 epochs.

C. Grid resolution and collocation density

Both the finite-difference scheme and the PINN rely on a discretized representation of the space–time domain, but the role of the grid resolution differs fundamentally between the two approaches.

For the FTCS scheme, the spatial and temporal resolutions directly control the numerical error through the truncation error of the method. As discussed in

Sec. II A 2, the FTCS scheme is first-order accurate in time and second-order accurate in space. Consequently, refining the grid by decreasing Δt and Δx leads to a systematic reduction of the discretization error. In the limit $\Delta t \to 0$ and $\Delta x \to 0$, the local truncation error vanishes, and, provided the stability condition is satisfied, the numerical solution converges towards the analytical solution. Increasing the grid resolution therefore results in a monotonic improvement of the FTCS solution at the expense of increased computational cost, but without any risk of overfitting to the collocation grid.

An illustrative example of this behaviour for the FTCS scheme is shown in Figure 6 in Appendix C, where the numerical solution is compared to the analytical solution at two time instances, $t = 0.05$ and $t = 0.5$, for two different spatial step sizes, $\Delta x = 0.1$ and $\Delta x = 0.01$. In accordance with the truncation error analysis, refining the spatial grid leads to a visibly improved agreement with the analytical solution at both time instances. The corresponding wall-clock runtimes are 0.37 ms for $\Delta x = 0.1$ and 22.60 ms for $\Delta x = 0.01$, reflecting the increased computational cost associated with finer spatial and temporal discretization, where Δt is chosen to satisfy the stability condition. Despite this increase, the absolute computational cost remains negligible for the grid resolutions considered here, highlighting the efficiency of the FTCS method for low-dimensional problems.

In contrast, for PINNs the grid resolution determines the number of collocation points at which the PDE residual and the initial and boundary conditions are enforced. A finer grid corresponds to a larger set of training points, increasing the amount of information provided to the network. However, unlike classical finite-difference methods, increasing the number of collocation points does not guarantee improved accuracy. Instead, a higher collocation density can make the optimization problem more challenging, requiring increased network capacity and more careful tuning of hyperparameters. In practice, this may lead to slower convergence or degraded performance due to optimization difficulties or effective overfitting to the collocation grid.

The choice of grid resolution therefore represents a trade-off between accuracy, representational capacity, and computational cost. In this work, a fixed grid resolution is used for both methods to enable a controlled comparison. The total number of collocation points for the PINN is given by

$$N_{\text{collocation}} = N_x \times N_t.$$

Figure 2 illustrates the effect of increasing the spatial grid resolution N_x (and thereby N_t) on both the final training cost and the maximum absolute error, MAE, against the analytical solution for the PINN. As the grid resolution increases, the final training cost generally decreases and begins to level off, indicating diminishing returns from further refinement. The MAE shows a similar overall trend, but does not decrease monotonically, highlighting that a denser collocation grid does not guarantee

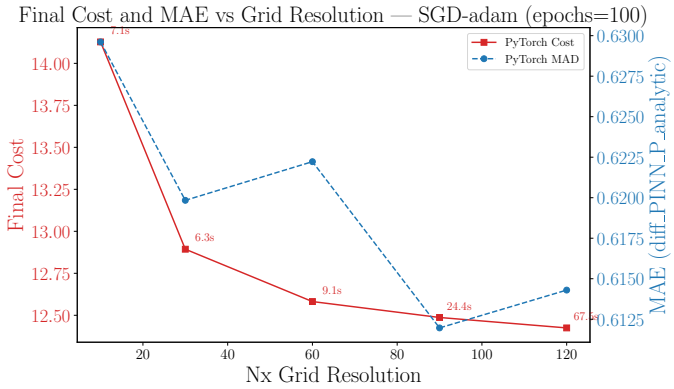


Figure 2: Final training cost (left axis) and maximum absolute error (MAE, right axis) with respect to the analytical solution as functions of the spatial grid resolution N_x for the PINN trained using using the PyTorch-SGD-Adam implementation, $lr = 0.01$, and epoch $= 100$. The annotations indicate the corresponding wall-clock training times.

improved physical accuracy. At the same time, the wall-clock training time increases rapidly with N_x, reflecting the higher computational cost associated with a larger number of collocation points. This growth in computational cost is substantially steeper than for the FTCS scheme, where grid refinement leads to only a modest increase in runtime for the resolutions considered here. This behaviour illustrates a clear trade-off between accuracy and computational efficiency. Based on these observations, a grid resolution of $N_x = 30$ is chosen for subsequent experiments, as it provides a reasonable balance between solution quality and training time while avoiding unnecessary computational overhead.

D. Effect of activation functions, network depth and width

To study the influence of activation functions, network depth and width, we performed a systematic hyperparameter sweep using sigmoid, tanh, and ReLU activations. For each activation function, networks with 1–6 hidden layers and 5–70 neurons per layer were trained. For every configuration, we recorded both the final training cost and the maximum absolute error (MAE) with respect to the analytical solution.

Due to the computational cost of this sweep, an early stopping criterion was introduced: training was terminated if the cost failed to decrease by at least 10^{-4} for 20 consecutive epochs. As a consequence, some architectures could have terminate before full convergence.

The heatmaps for the sigmoid activation, shown in Figures 9 and 10 in Appendix E, reveal a non-uniform performance across the hyperparameter space. While certain combinations of depth and width achieve low final cost

and low error, multiple architectures yield consistently high cost and error values. In particular, increasing network depth is associated with a larger fraction of poorly performing configurations, indicating a strong sensitivity to architectural choices and training dynamics.

The corresponding results for the tanh activation, shown in Figures 11 and 12, do not exhibit a clear improvement over the sigmoid activation. While some combinations of depth and width yield moderate final cost and error values, a large fraction of architectures display high cost and large maximum error. Overall, the well-performing regions observed for the tanh activation are comparable to those obtained with the sigmoid activation. However, compared to sigmoid, tanh appears more sensitive to network depth, with deeper architectures more frequently associated with high final cost and large maximum error.

In contrast, the ReLU activation generally yields higher final cost and larger maximum error than sigmoid and tanh for small and moderately sized networks. The heatmaps in Figures 13 and 14 show a clear improvement in performance with increasing network depth and width, with the lowest ReLU costs obtained for the deepest and widest architectures considered in this study. However, even the best-performing ReLU configurations do not achieve cost or error levels comparable to those obtained with sigmoid or tanh.

Furthermore, the error heatmap indicates that a reduction in training cost for ReLU does not necessarily translate into improved agreement with the analytical solution. This suggests that, despite improved optimization performance at higher model capacity, ReLU-based PINNs struggle to accurately approximate the smooth solution of the heat equation.

The observed ReLU behaviour is consistent with the theoretical considerations discussed in Section II B 3. Since ReLU is only piecewise linear and not smoothly differentiable, the evaluation of higher-order derivatives appearing in the PDE residual is limited, due to being zero almost everywhere. As a consequence, accurately representing smooth solutions such as those of the heat equation may require substantially increased model capacity. This is reflected in the heatmaps, where ReLU performance improves with increasing depth and width, yet remains inferior to sigmoid and tanh across the explored hyperparameter space.

Across all three activation functions, the heatmaps further demonstrate that a low final training cost does not necessarily imply low physical error with respect to the analytical solution. This effect is particularly pronounced for ReLU, where improvements in cost are not consistently accompanied by corresponding reductions in the maximum absolute error. These results highlight the importance of evaluating PINNs using complementary accuracy metrics rather than relying solely on the training loss.

Although sigmoid and tanh exhibit comparable regions of good performance, neither activation function demonstrates uniformly robust behaviour across the full hyperparameter space. In particular, deeper architectures are associated with an increased fraction of poorly performing configurations for both activations, with tanh appearing slightly more sensitive to network depth. Consequently, selecting architectures solely based on the minimum achieved cost may favour isolated configurations rather than models with reliable performance. Nevertheless, due to the limited scope and time constraints of the present project, the final architecture used in subsequent experiments is selected based on the minimum achieved training cost. This corresponds to a sigmoid-activated network with two hidden layers and 40 neurons per layer.

In addition to architectures employing a single activation function across all hidden layers, we also explored mixed activation-function networks, allowing different activation functions to be used in different layers. However, these mixed-activation configurations did not yield improved performance compared to the best single-activation architectures. In particular, the lowest training cost was again obtained for a sigmoid-based network with two hidden layers of 40 neurons each.

We therefore proceed by selecting this architecture as the best-performing PINN configuration, and use it as the reference model for subsequent comparisons with the analytical solution and the FTCS finite-difference scheme.

E. The best PINN: a summary

After evaluating several PINN implementations, optimization algorithms, learning rates, and grid resolutions, we identified the best-performing configuration in terms of accuracy versus computational cost. The observed sensitivity to learning rate, architecture, and optimization strategy is consistent with the highly non-convex loss landscape of PINNs, where convergence to different local minima can be strongly influenced by initialization and stochastic effects. Nevertheless, the preferred implementation employs PyTorch with a SGD-Adam optimizer, a learning rate of 0.01, and 100 epochs (with stopping criteria implemented). To balance accuracy and runtime we trained on a 30×30 spatial-temporal grid. The optimal network uses sigmoid activations and a feedforward architecture with two hidden layers of 40 neurons each. This model converged to a final cost of 2.54e-04. The resulting PINN solution for the 1D heat equation is shown in Figure 4a.

F. Comparing PINN with FTCS

As a classical numerical reference, we solved the heat equation using the FTCS finite-difference scheme. We use FTCS as a baseline to analyze the performance and computational cost of our PINN. To ensure a fair and controlled comparison, both the PINN and the FTCS

scheme are evaluated on identical spatial–temporal domains and compared against the same analytical reference solution. Results are shown in Figures 4 and 5.

From Figure 4 we see that both the PINN and FTCS reproduce the analytical solution qualitatively, with temperatures in the same range, across similar temporal and spatial domains. However, Figure 5 makes the differences become more clear. Both methods satisfy the initial and boundary conditions to a negligible error, indicating they respect the physical constraints.

At later times, we see that the FTCS solution (Figure 5b) constantly under predicts the temperature. This is most significant shortly after the initial time, and gradually decreases. This likely stems from the strong temperature gradient in the analytical solution that FTCS cannot fully resolve. This is also seen in Figure 6.

For FTCS, accuracy is governed by spatial and temporal discretization. In Figure 3, the MAE decreases with finer grid resolution, with a significant improvement after 10x10 grid resolution. This is in accordance with the truncation error analysis and results shown in Section IV C. After 30x30 grid points, the improvement stops increasing significantly. However, execution time nearly doubles for every 30-point increase in grid resolution, and beyond 30x30 there is little benefit relative to the decreased MAE.

After the initial conditions, the PINN also deviates significantly from the analytical solution (Figure 5a), exhibiting larger overall errors than FTCS. The errors also have a more complex pattern, compared to FTCS which simply under predict the temperature gradient. We see that PINN initially under-predicts temperatures, then around $t \approx 0.25$ it begins to over-predict. It also has some interesting asymmetrical features, where the upper and lower side of the rod differentiate in temperatures for the same time.

We tested how the PINN performed on different grid resolutions, after being trained on a 30x30 grid, seen in Figure 3. We see that for the PINN, the MAE is lowest for a grid resolution of 10x10, and keeps increasing for higher grid resolution, which is opposite of the FTCS. The two approaches differ fundamentally in the origin and behavior of their numerical errors. For FTCS, the error arises from spatial and temporal discretization and is directly controlled by the grid resolution and time step. This discretization error accumulates deterministically over time but can be systematically reduced by refining the grid, subject to the stability constraints of the method. The PINN generalizes best near or below the resolution it was trained on (30x30). When evaluated on grids finer than the training resolution, it cannot capture details it was not trained to represent.

In terms of computational cost, training the PINN on a 30×30 grid took just under 16 seconds. While the PINN approach successfully reproduces the analytical solution, it comes at a substantially higher computational cost for the present problem. It is also worth noting that fine-tuning the neural networks required repeated runs, extensive hyperparameter tuning, and careful selection of network architectures and optimizers, resulting in several weeks of experimentation. In contrast, the FTCS scheme is straightforward to implement and executes in negligible time once the spatial and temporal discretization has been specified. The execution time for the PINN solution on different grid sizes are significantly higher compared to the FTCS. In general, PINN has a higher MAE compared to FTCS, with exception of the 10x10 grid.

For the simple, low-dimensional heat problem considered here, FTCS therefore outperforms PINNs both in terms of accuracy and computational efficiency. However, PINNs may become advantageous for higher-dimensional PDEs, more complex geometries, or problems with sparse, irregular, or partially observed data, where traditional grid-based methods become prohibitively expensive or difficult to implement. In such settings, the increased computational cost of training a PINN may be offset by its flexibility and ability to incorporate physical constraints without explicit meshing. For the present problem, however, the FTCS scheme remains both more accurate and computationally efficient.

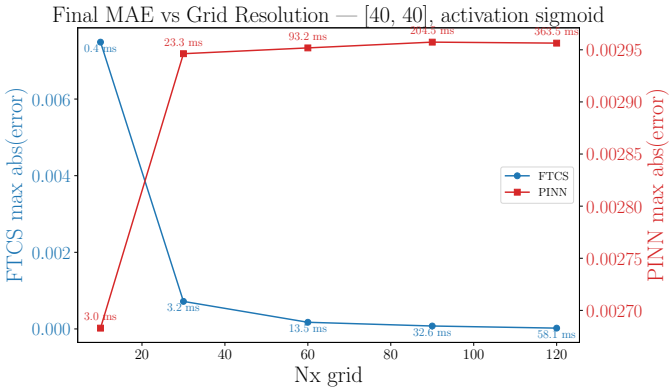
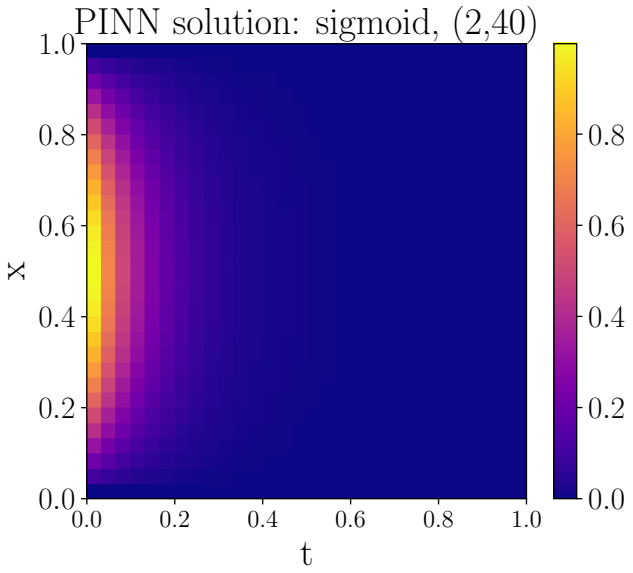
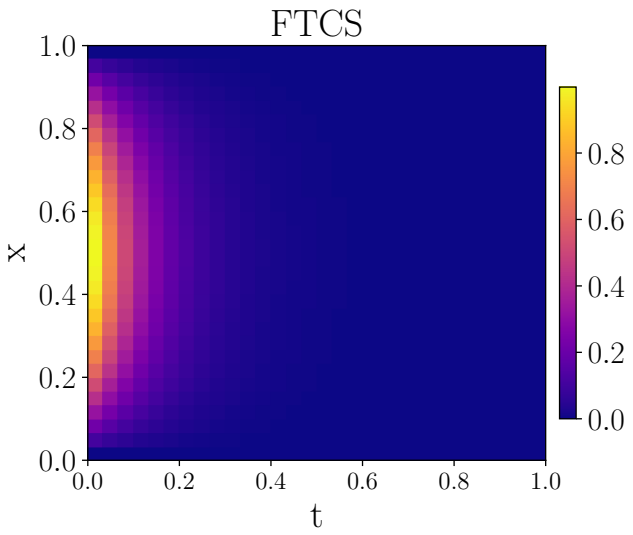


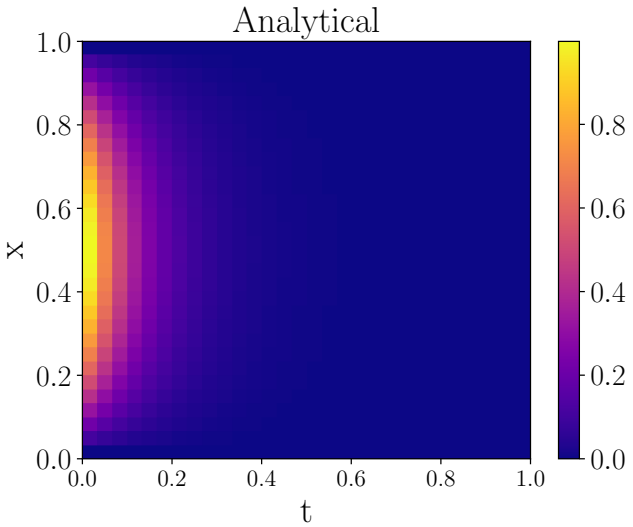
Figure 3: The max absolute error, MAE, between the FTCS (left axis) and PINN (right axis) against the analytical solution for different grid resolution N_x. The PINN is trained using a grid resolution of 30x30, using the PyTorch-SGD-Adam implementation, $lr = 0.01$, and epoch = 100. The grid resolution tested was [10, 30, 60, 90, 120]. The annotations indicate the corresponding wall-clock training times.



(a) Best PINN solution heatmap.

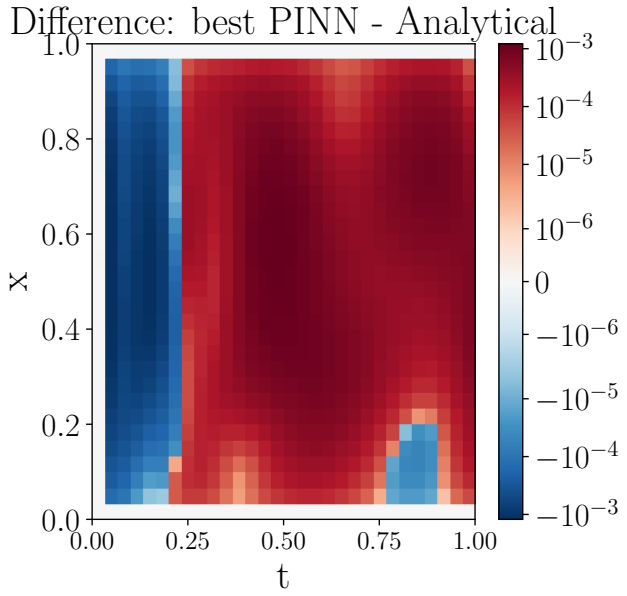


(b) FTCS solution heatmap.

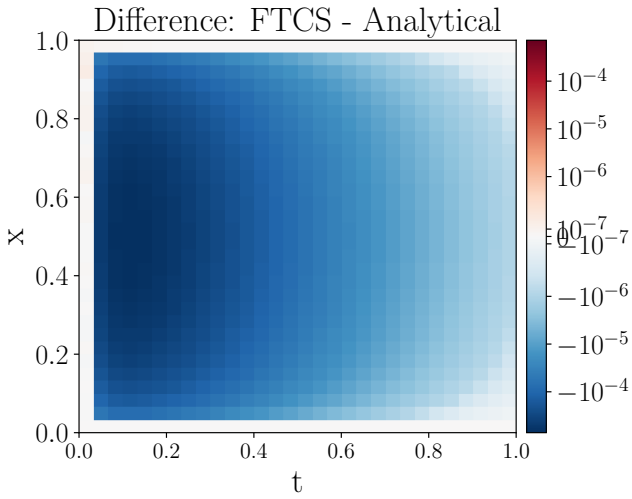


(c) Analytical solution heatmap.

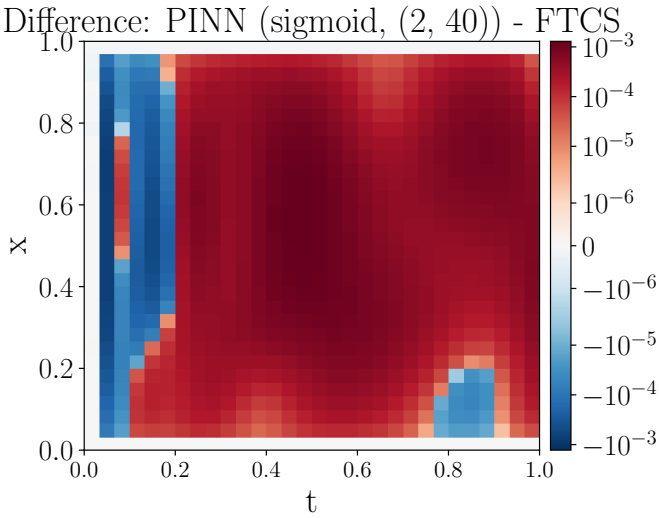
Figure 4: PINN, FTCS, and analytical solution to the heat equation.



(a) Best PINN solution minus analytical solution.



(b) FTCS minus analytical solution.



(c) Best PINN solution minus FTCS solution.

Figure 5: Comparison between the PINN, FTCS solution and the analytical solution.

V. CONCLUSION

In this project, physics-informed neural networks (PINNs) were employed to solve the one-dimensional heat equation and systematically compared against both the analytical solution and a classical finite-difference scheme based on the Forward-Time Central-Space (FTCS) method. Two independent PINN implementations were developed, one using `Autograd` and one using the `PyTorch` library. A direct comparison between the two implementations showed that the `PyTorch`-based model, trained using stochastic gradient descent with the Adam optimizer (sgd-Adam) and a learning rate of 0.01, was both computationally more efficient and exhibited more stable and reliable convergence. This implementation was therefore selected for all subsequent experiments.

The influence of spatial resolution was investigated for both the FTCS scheme and the PINN approach. In both cases, a clear trade-off between computational efficiency and accuracy was observed. Increasing the grid resolution led to improved solution quality at the cost of increased runtime. Based on these results, a spatial resolution of $N_x = 30$ was identified as a reasonable compromise for both methods, providing satisfactory accuracy while avoiding unnecessary computational overhead.

An extensive hyperparameter study was conducted to assess the impact of activation functions and network architecture on PINN performance. Among the activation functions considered, sigmoid consistently yielded the lowest final training cost, with tanh showing comparable but slightly inferior performance, while ReLU performed significantly worse. For sigmoid and tanh, wider networks with more neurons per layer generally improved performance, whereas increasing network depth led to worse results. In contrast, ReLU-based networks exhibited lower training cost for deeper and wider architectures, but this did not translate into improved physical accuracy, as indicated by a higher error with respect to the analytical solution. Overall, the best-performing PINN consisted of a sigmoid-activated network with two hidden layers of 40 neurons each, converging to a final cost of 2.54×10^{-4}.

This 'best' PINN configuration was subsequently compared to the FTCS solution, using the analytical solution as a reference. The results clearly demonstrate that, for this problem, the FTCS scheme outperforms the PINN in terms of both accuracy and computational efficiency. Furthermore, the PINN solution exhibited more complex and more asymmetric error patterns compared to the analytical solution, indicating limitations in the network's ability to fully capture the underlying structure of the temperature distribution.

In conclusion, for the one-dimensional heat equation considered in this study, the classical FTCS method provides superior performance compared to the PINN approach implemented here. Given the increased computational cost and reduced accuracy of the PINN, there is no clear justification for its use in this setting, where efficient and well-established numerical methods are readily available.

VI. LIMITATIONS AND FUTURE WORK

Although the FTCS scheme clearly outperformed the PINN approach for the simple one-dimensional problem considered in this study, there remain several scenarios where PINNs may still offer advantages. In higher-dimensional PDEs, problems involving complex geometries, or settings with sparse, irregular, or partially observed data, traditional grid-based methods can become computationally expensive or difficult to implement. In such cases, the increased computational cost associated with training a PINN may be offset by its flexibility and its ability to enforce physical constraints without relying on explicit meshing.

Furthermore, while the present problem admits an analytical solution, many real-world systems do not. In such settings, PINNs may still provide meaningful approximations even when classical error metrics relative to an analytical reference are unavailable, making them a potentially valuable tool for exploratory or data-driven modelling of complex physical systems.

Finally, continued developments in hardware, optimization algorithms, and neural-network frameworks suggest that the performance gap between PINNs and classical numerical schemes may narrow over time. More extensive hyperparameter optimization, alternative network architectures, improved initialization strategies, and the use of multiple random seeds could yield more accurate and robust PINN solutions. With sufficient computational resources and further methodological advances, it remains possible that a well-optimized PINN could outperform classical finite-difference methods for certain classes of problems.

[1] H. S. Carslaw and J. C. Jaeger. *Conduction of heat in solids*. Clarendon Press ; Oxford University Press, Oxford [Oxfordshire] : New York, 2nd edition, 1959.

[2] John Crank. *The mathematics of diffusion*. Oxford science publications. Clarendon press Oxford university press, Oxford Oxford New York, 2nd edition, 1975.

[3] Lawrence C. Evans. *Partial differential equations*. Number vol. 19 in Graduate studies in mathematics. American mathematical society, Providence (R.I.), 2nd edition, 2010.

[4] O. C. Zienkiewicz. *The finite element method*. London ; New York : McGraw-Hill, 1989.

[5] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, vol. 9(no. 5):987–1000, September 1998.

[6] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[7] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, May 2021.

[8] M. Raissi, P. Perdikaris, and G.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*, vol. 378:686–707, February 2019.

[9] Alexandre Ern and Jean-Luc Guermond. *Theory and Practice of Finite Elements*, volume 159 of *Applied Mathematical Sciences*. Springer New York, New York, NY, 2004.

[10] Morten Hjorth-Jensen. Project 3 on Machine Learning, deadline December 15 (midnight), 2025 — Applied Data Analysis and Machine Learning. Technical report, 2025.

[11] Morten Hjorth-Jensen. *Computational Physics Lecture Notes 2015*. Department of Physics, University of Oslo, Norway, 2015.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[13] Sebastian Raschka. *Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python*. Packt Publishing Limited, Birmingham, 1 edition, 2022.

[14] Morten Hjorth-Jensen. *Applied Data Analysis and Machine Learning — Lecture notes*. 2025. Available from: CompPhysics/MachineLearning/tree/master/doc at master · mhjensen/CompPhysics.

[15] Kjersti Stangeland, Jenny Guldvog, Ingvild Olden Bjerkelund, and Sverre Johansen. kjes4pres/Project_2_fysstk. original-date: 2025-09-02T10:34:05Z.

[16] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014. Version Number: 9.

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

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019. Version Number: 1.

[19] Kjersti Stangeland, Jenny Guldvog, Ingvild Olden Bjerkelund, and Sverre Manu Johansen. Regression analysis and resampling methods, FYS-STK4155 - Project 1. Technical report, University of Oslo, Norway, June 2025.

[20] Charles R. Harris, K. Jarrod Millman, Stéfan J. Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. Van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández Del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[21] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[22] Wes McKinney. Data Structures for Statistical Computing in Python. pages 56–61, Austin, Texas, 2010.

[23] OpenAI. ChatGPT, 2025.

[24] University of Oslo. GPT UiO, 2025.

Appendix A: Derivation of the analytical solution of the diffusion equation

The one-dimensional heat equation is considered in the compact form

$$u_t = u_{xx}, \qquad x, t \in [0, 1],$$

subject to homogeneous Dirichlet boundary conditions

$$u(0, t) = u(1, t) = 0,$$

and the initial condition

$$u(x, 0) = \sin(\pi x).$$

To derive the analytical solution, we follow the separation of variables approach described in [11, § 10.2.4]. Assuming a solution of the form $u(x, t) = X(x)T(t)$ and imposing the boundary conditions yields the spatial eigenfunctions

$$\sin(n\pi x), \qquad n = 1, 2, \ldots,$$

with corresponding eigenvalues

$$\lambda_n = (n\pi)^2.$$

The general solution can therefore be written as a Fourier sine series,

$$u(x, t) = \sum_{n=1}^{\infty} b_n \sin(n\pi x)\, e^{-(n\pi)^2 t}.$$

The coefficients b_n are determined from the initial condition $u(x, 0) = g(x)$ as

$$b_n = 2 \int_0^1 g(x) \sin(n\pi x)\, dx.$$

For the present problem, where $g(x) = \sin(\pi x)$, this becomes

$$b_n = 2 \int_0^1 \sin(\pi x) \sin(n\pi x)\, dx.$$

Using the orthogonality of the sine functions on the interval $[0, 1]$, we obtain

$$\int_0^1 \sin(\pi x) \sin(n\pi x)\, dx = \begin{cases} \frac{1}{2}, & n = 1, \\ 0, & n \neq 1. \end{cases}$$

Consequently, $b_1 = 1$ and $b_{n\neq 1} = 0$.

The analytical solution therefore reduces to the closed-form expression

$$u(x, t) = \sin(\pi x)\, e^{-\pi^2 t}. \tag{A1}$$

Appendix B: Derivation of the FTCS of the diffusion equation

To solve Eq. (1) using FTCS, we follow the discretization described in [11, § 10.2.1]. The spatial and temporal domains are discretized as

$$x_i = i\Delta x, \quad i = 0, \ldots, n+1, \qquad \Delta x = \frac{1}{n+1},$$

and

$$t_j = j\Delta t, \quad j = 0, \ldots, N_t.$$

We denote the numerical approximation of $u(x_i, t_j)$ by u_i^j.

Using standard finite-difference approximations, the temporal and spatial derivatives are given by

$$u_t(x_i, t_j) \approx \frac{u_i^{j+1} - u_i^j}{\Delta t},$$

and

$$u_{xx}(x_i, t_j) \approx \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{\Delta x^2}.$$

Inserting these expressions into the heat equation $u_t = u_{xx}$ yields

$$\frac{u_i^{j+1} - u_i^j}{\Delta t} = \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{\Delta x^2}, \tag{B1}$$

which can be solved explicitly for u_i^{j+1} as

$$u_i^{j+1} = u_i^j + \frac{\Delta t}{\Delta x^2}\left(u_{i+1}^j - 2u_i^j + u_{i-1}^j\right),$$

valid for the interior grid points $i = 1, \ldots, N_x - 1$.

Defining the dimensionless parameter

$$\alpha = \frac{\Delta t}{\Delta x^2},$$

the update formula can be written in the compact form

$$u_{i,j+1} = \alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} + \alpha u_{i+1,j}. \tag{B2}$$

For stability, the FTCS scheme requires $\alpha \leq 1/2$ [11, § 10.2.1].

The FTCS method is first-order accurate in time and second-order accurate in space, resulting in a local truncation error of [11, § 10.2.1]

$$\mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2).$$

Appendix C: FTCS Δx check

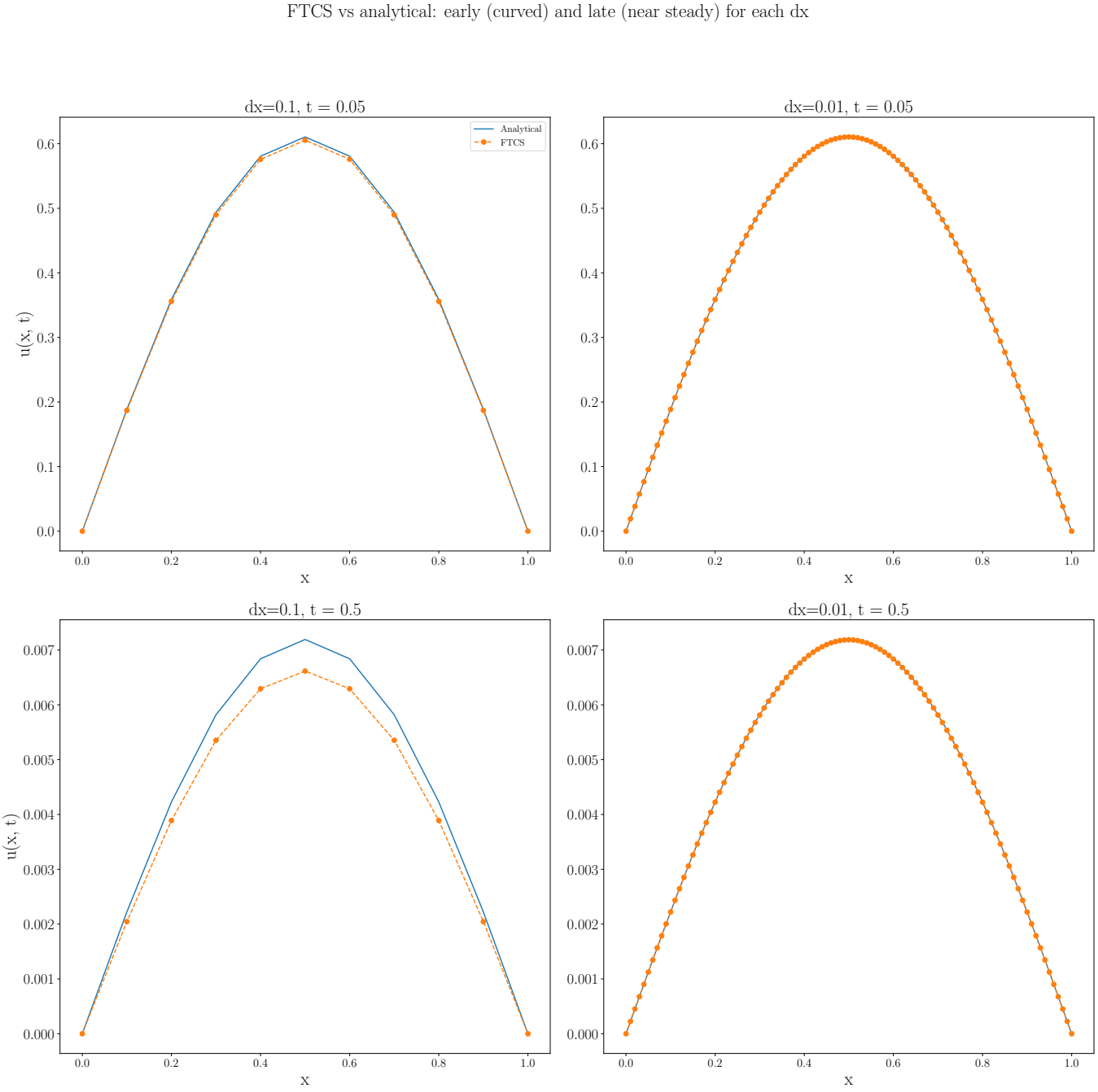


Figure 6: Comparison between the FTCS solution and the analytical solution of the one-dimensional heat equation at an early time ($t = 0.05$) and a later time ($t = 0.5$) for two different spatial step sizes, $\Delta x = 0.1$ and $\Delta x = 0.01$. The upper row shows the solution profiles at early times, where the curvature is more pronounced, while the lower row shows the solution closer to the steady-state regime. As the spatial resolution is refined, the FTCS solution approaches the analytical solution more closely at both time instances.

Appendix D: Cost history figures

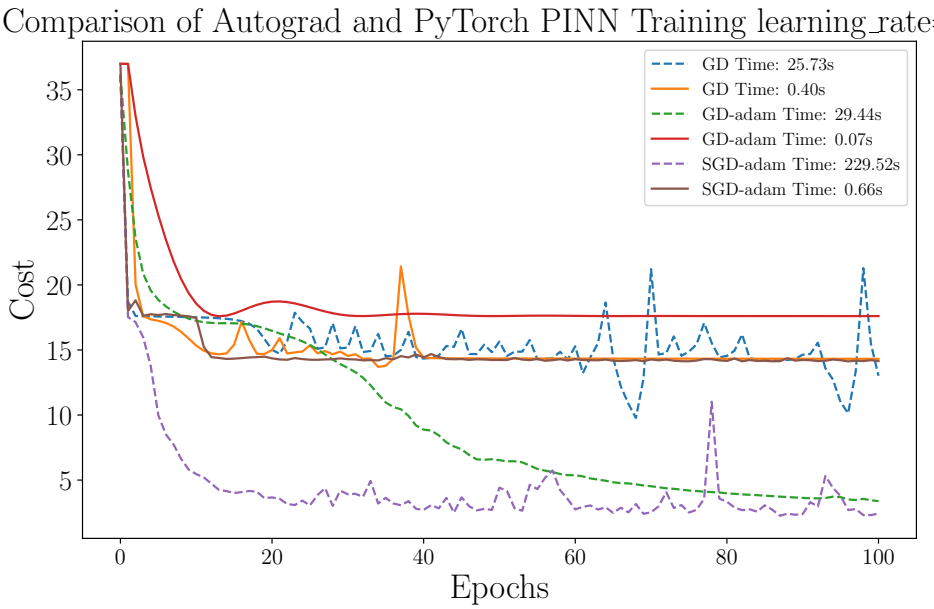


Figure 7: Training loss as a function of epochs for different PINN optimization strategies using a learning rate of 10^{-1}. Results are shown for gradient descent (GD), GD with the Adam optimizer, and stochastic gradient descent with Adam (SGD–Adam), implemented using both Autograd (dashed) and PyTorch (solid). The legend reports the corresponding wall-clock training times for each configuration.

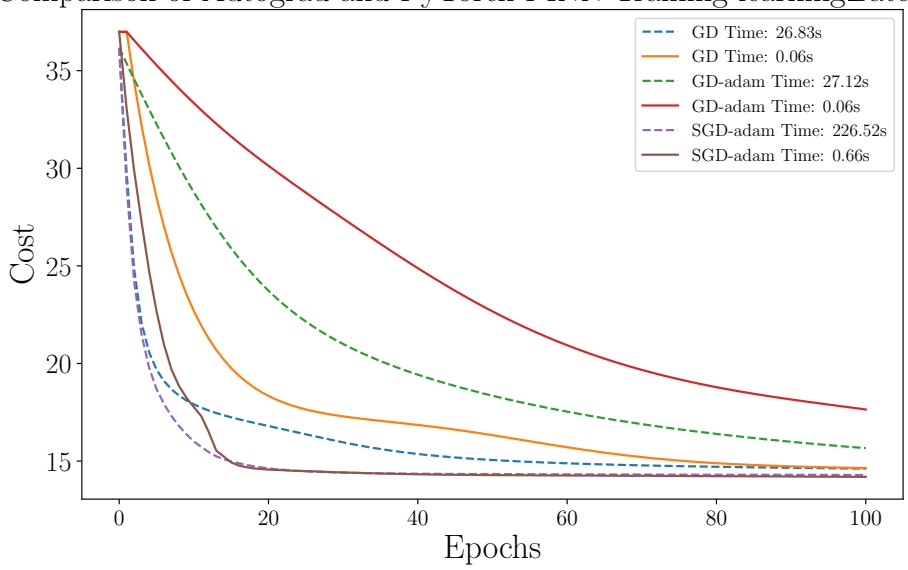


Figure 8: Training loss as a function of epochs for different PINN optimization strategies using a learning rate of 10^{-2}. As in Figure 7, results are shown for GD, GD–Adam, and SGD–Adam implemented with both Autograd (dashed) and PyTorch (solid), with wall-clock training times reported in the legend.

Appendix E: Heat matrices

1. Sigmoid

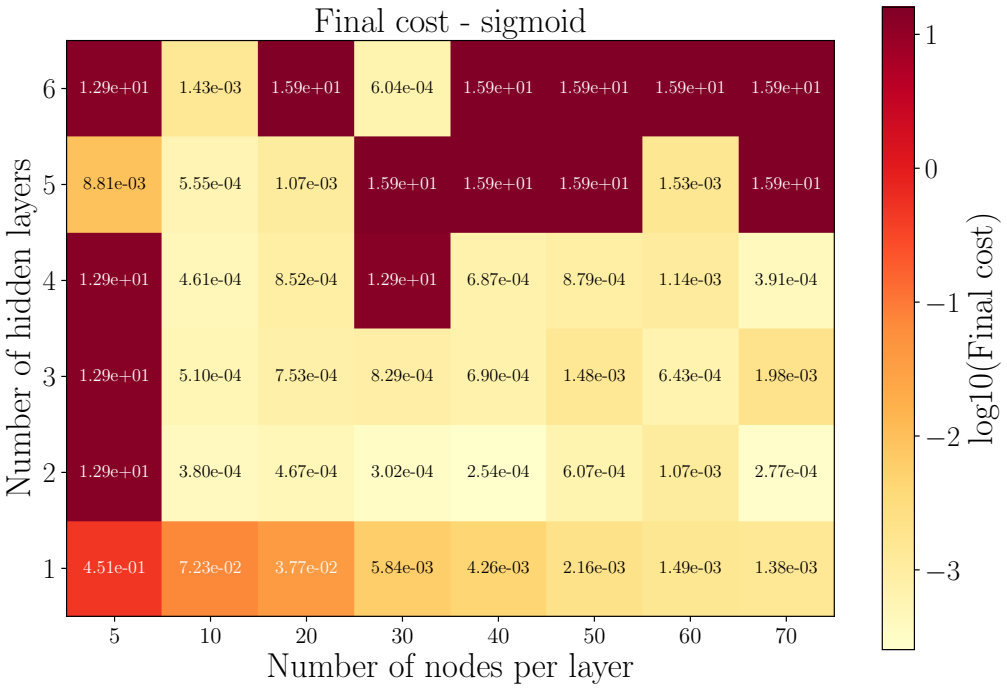


Figure 9: Final training cost as a function of network depth and width for the sigmoid activation.

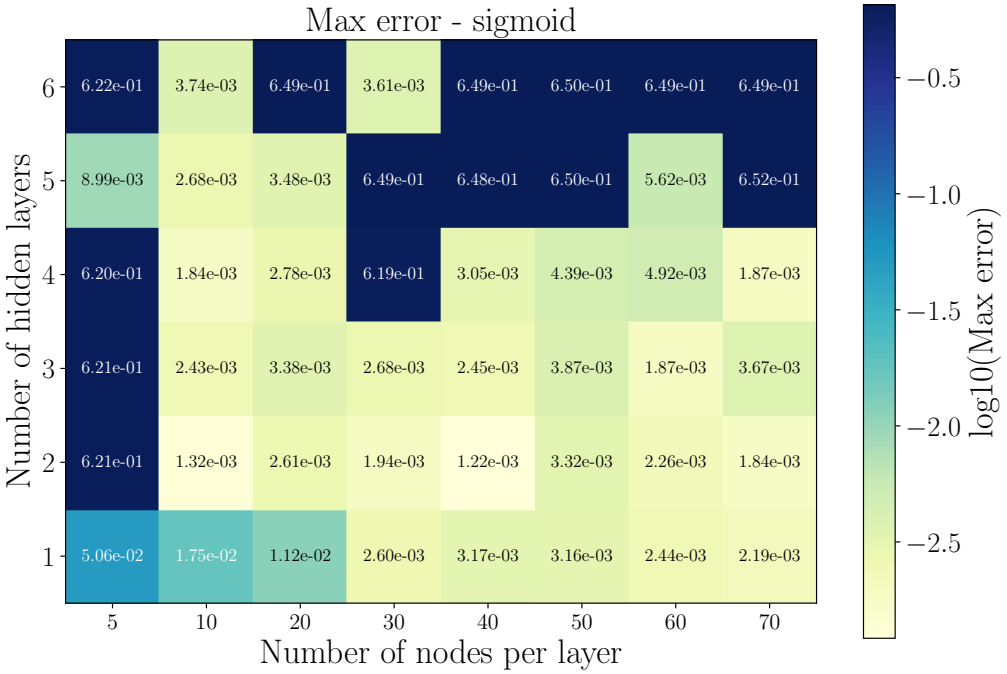


Figure 10: Maximum absolute error with respect to the analytical solution as a function of network depth and width for the sigmoid activation.

2. Tanh

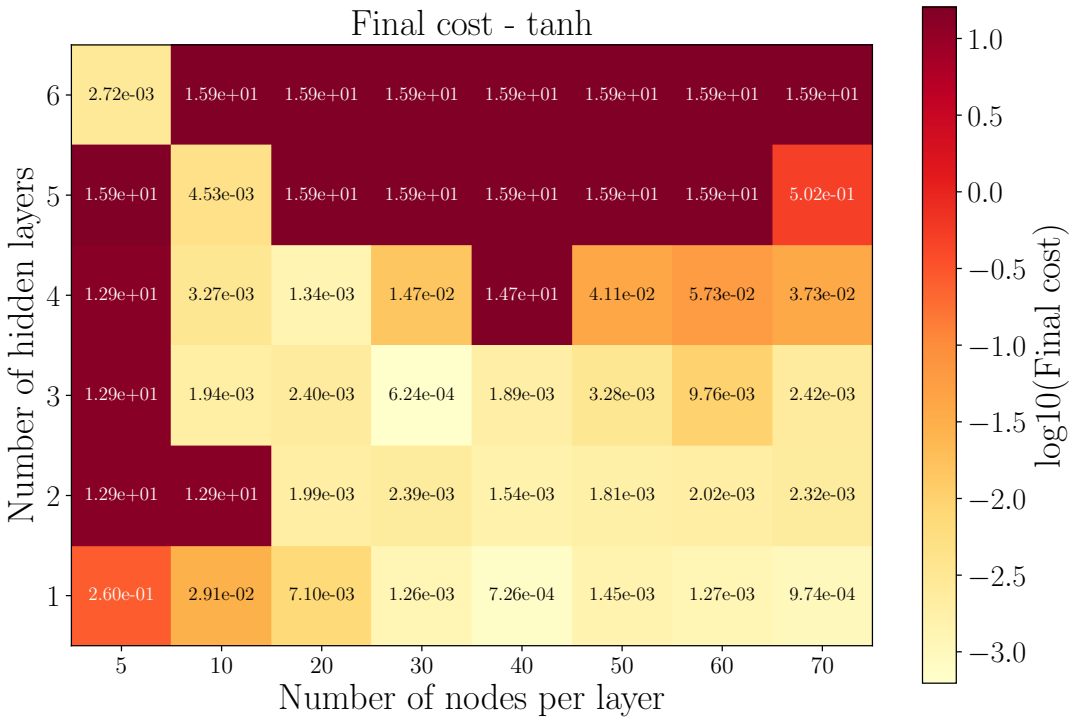


Figure 11: Final training cost as a function of network depth and width for the tanh activation.

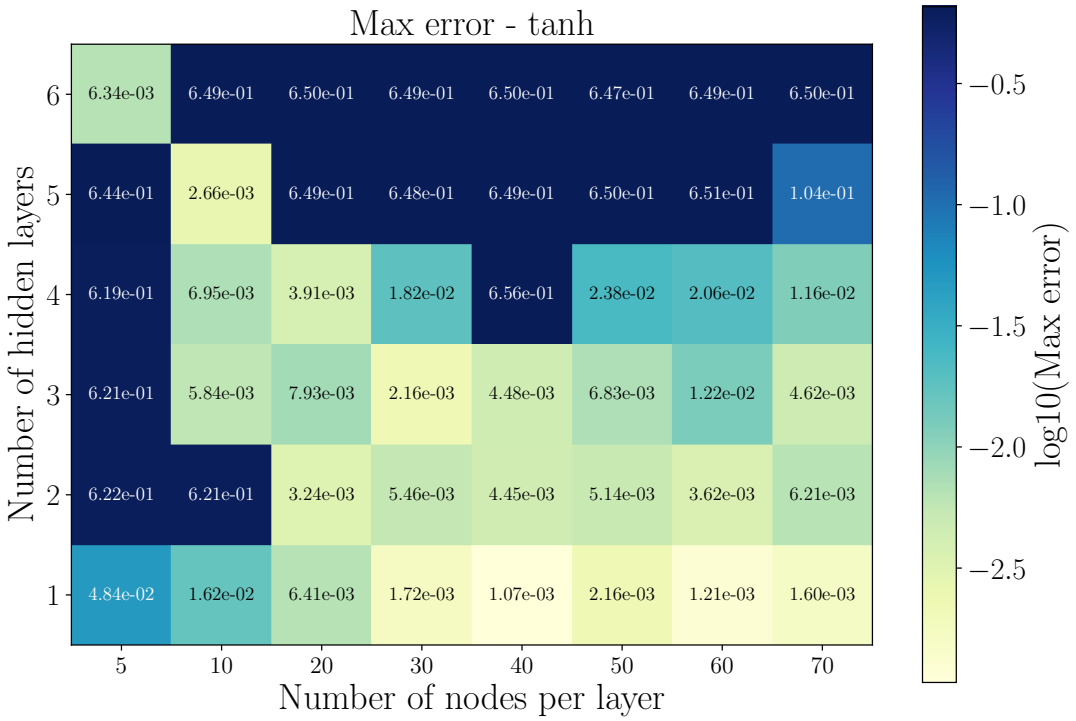


Figure 12: Maximum absolute error with respect to the analytical solution as a function of network depth and width for the tanh activation.

3. Relu

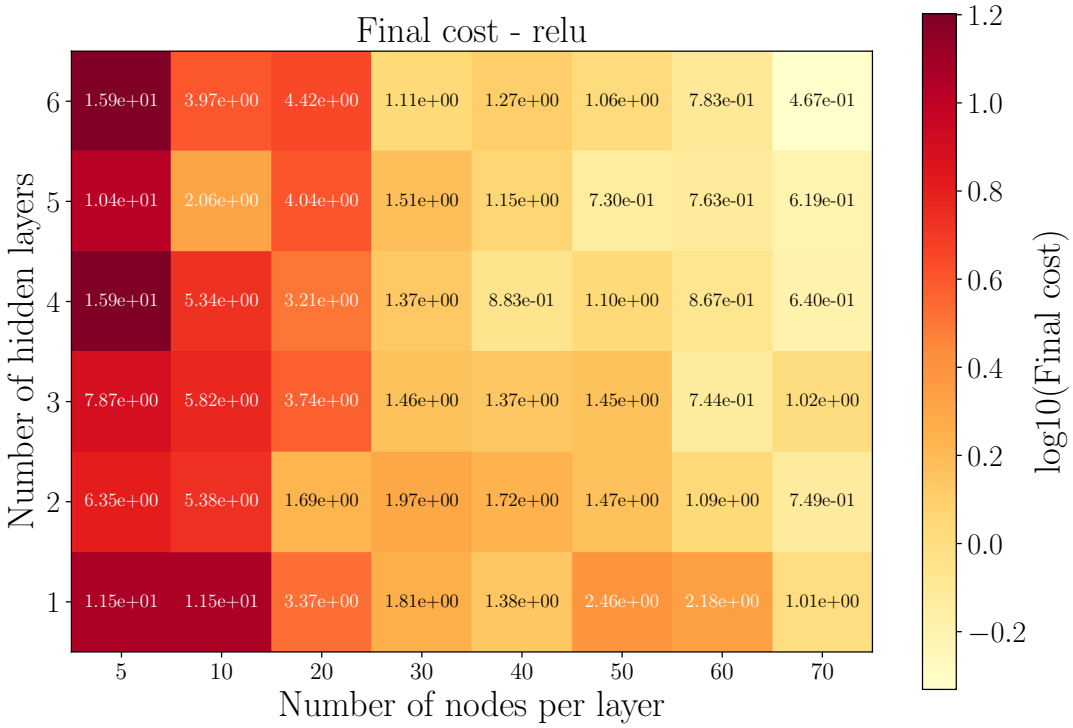


Figure 13: Final training cost as a function of network depth and width for the ReLU activation.

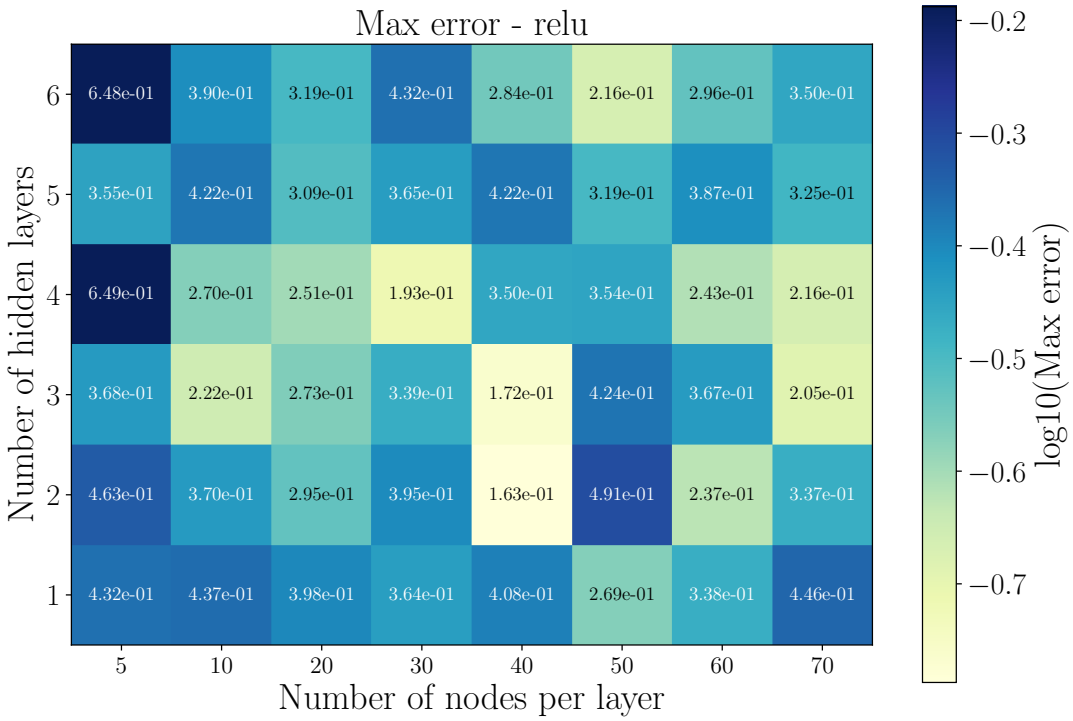


Figure 14: Maximum absolute error with respect to the analytical solution as a function of network depth and width for the ReLU activation.