A Physics-Informed Neural Network Approach to the 1D Diffusion Equation
FYS-STK4155 - Project 3

Amund Solum Alsvik, Kristoffer Stalker, and Jørgen Vestly
University of Oslo
(Dated: December 18, 2025)

Deep learning is a versatile tool, applicable in many fields of study. Solving differential equations is central in many scientific disciplines, hence having a tool that can reliably solve PDEs would be valuable. In this project, we test the suitability of using neural networks to solve the 1-dimensional diffusion equation, which is a commonly used equation to model heat and other phenomenons in physics. We compare the results from the physics-informed neural network, to a classic solver algorithm, namely the forward Euler method. We find that the network can gives comparable results to the forward Euler scheme, without being constrained by stability conditions. Moreover, we also found that for fewer collocation points, the network did significantly better than the forward Euler method, which suffered under these conditions. Moreover, we also perform a grid search to find good hyperparameters for the network, while also briefly exploring the effectiveness of different optimizing methods, showing that ADAM gives the best results.

1. INTRODUCTION

Neural networks have proven to be very versatile. In Alsvik et. al. [1], neural networks solved two seemingly different problems, being classification of handwritten digits and regression. It turns out that neural networks can be used for another class of problems, namely solving partial differential equations. In physics and other scientific disciplines, solving differential equations is crucial for understanding various behaviors and models. In this report, we are specifically looking at the 1-dimensional diffusion equation, given by

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

We chose this equation because it is sufficiently complex to demonstrate the versatility of neural networks while still being manageable as an introductory example. By using a *physics-informed neural network* (PINN), we are able to integrate the physical properties of the diffusion equation into the architecture of the network, allowing us to approximate a solution. While classification, regression and solving differential equations are perhaps conceptually different, a neural network approaches these problems somewhat similarly. All of these types of problems can be viewed as approximating some continuous function, thus as explained in Alsvik et. al. [1], the *universal approximation theorem*, ensures that there exists some network that can approximate a solution for any degree of accuracy.

In section 2, we briefly discuss the diffusion equation, while also explaining how equations can be solved numerically with the Forward Euler method. We also introduce the architecture of the PINN used to solve the diffusion equation. In section 3, we show how the code is implemented, along with discussion of testing and the practical aspect of the research. Section 4 contains our main findings, where we compare the results from our PINN to the Forward Euler method. We also discuss pros and cons of both methods, while presenting various customization options in our PINN architecture, regarding hyperparameters and activation functions. We end with a summary of our findings in section 5, where we discuss their implications and perspective for future work.

2. THEORY AND METHODS

In this section, we introduce partial differential equations, focusing on diffusion equations in particular. We also talk about the architecture of a physics-informed neural network. We assume that the reader is familiar with feed forward neural networks and how they are trained. For an introduction to this, see Alsvik et. al. [1].

2.1. Partial Differential Equations and Diffusion equations

Put simply, a partial differential equation (PDE) is an equation which involves a function of multiple variables and its partial derivatives with respect to those variables. For a function $u(x_1, \ldots, x_N)$ with N-variables, any PDE with respect to u can be written as

$$f\left((x_1, \ldots, x_N), \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_N}, \frac{\partial u}{\partial x_1 \partial x_2}, \ldots, \frac{\partial^n u}{\partial x_N^n}\right) = 0,$$

where f can contain any combination of partial derivatives. The PDE we are looking at in this project can much simpler be expressed as

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

where $u(x,t)$ is an unknown function depending on a 1-dimensional coordinate $x \in [0, L]$ and a time parameter $t \in [0, T]$. This is what we call a diffusion equation, and is in one dimension categorized by the fact that the derivative with respect to t is equal to the second derivative with respect to x. We also have an initial and two boundary conditions, namely that at $t = 0$, we set

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

while for for $x \in \{0, L\}$, we set

$$u(0, t) = 0,$$
$$u(L, t) = 0.$$

We interpret $u(x, t)$ as the temperature of a rod of length $L = 1$, where we try to model heat at different points on the rod, at different time steps. From Hjort-Jensen [2], we know that equation 1 has an analytical solution given by

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

Moreover, we can also solve this equation numerically using the Forward Euler algorithm 1. We will use both the analytical and numerical solution as a benchmark for our PINN.

2.2. Forward Euler Algorithm

As mentioned, we use the venerable *Forward Euler alogrithm* as a benchmark for our PINN. The algorithm goes as follows: We partition the spatial interval and the time interval into equally spaced points, giving us points on the form (x_i, t_j). We denote the spacing of the intervals as Δx and Δt for each interval respectively. To approximate the derivatives of u at each of these points, we use the following formulas

$$u_t(x_i, t_j) \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t},$$
$$u_{xx}(x_i, t_j) \approx \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)}{\Delta x^2}.$$

For brevity we set $x_i + \Delta x = x_{i+1}$, $x_i - \Delta x_i = x_{i-1}$, $t_j + \Delta t = t_{j+1}$ and $u(x_i, t_j) = u_i^j$. By substituting these expressions into equation 1, we get that

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

Hence, we get the update formula

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

As explained in Massing [3], for the diffusion equation, the forward Euler explicit scheme has a stability criterion

$$\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}.$$

As can be seen in our results section, violation of this criterion gives numerical errors as t grows. In our implementation we therefore choose Δt in accordance with this bound for each choice of Δx.

2.3. Physics-Informed Neural Networks

There are several ways to design a PINN, depending on the physical problem one is trying to solve. In this project, however, we limit ourselves to a design which is specific to equation 1. Our network is reminiscent of a standard feed-forward network, where we train the network with back-propagation. There are however some key differences. Firstly, there is no labeled data, meaning the networks never actually see any target values during training. Instead, our cost function embeds equation 1, and is given by

$$C(x, t, \theta) = \left(\frac{\partial^2 u_\theta}{\partial x^2} - \frac{\partial u_\theta}{\partial t}\right)^2, \tag{2}$$

where u_θ will be our prediction, depending on weights from the PINN. We explain how we construct such a function in more detail shortly. Since equation 1 has a unique solution, we know that a function that minimizes equation 2, will be a good approximation for the solution. Since we do not have labeled data, we have no input values in the traditional sense either, instead we have *collocation points*, which are the points we feed into the network, and are chosen from interior, initial and boundary points of equation 1. Lastly, the output of our network is not easy to interpret in itself. Instead of evaluating the output from the forward pass directly, we utilize the trained weights to construct a prediction in the form of a function. The overall algorithm is somewhat analogous to the steps described in Alsvik et. al. [1], and relies on the feed-forward and back-propagation methods. The steps are structured in the following manner:

Step 1. We begin by defining a trial solution. How the trial solution is defined will depend on the problem one is solving, but for equation 1, a natural choice is

$$u_\theta(x, t) = h_1(x, t) + x(1 - x)tN(x, t, P),$$

where $N(x, t, P)$ is the output of our networks in terms of its set of weights and bias terms, while

$$h_1(x, t) = (1 - t)\sin(\pi x).$$

This trial solution is taken from Hjort-Jensen [2], and is defined specifically so that it satisfy the initial and boundary conditions. Hence, our final prediction will be u_θ with respect to the trained set of weights and bias terms from our PINN. A benefit of defining the trial function in this way is that it removes the need of terms for the boundary and initial conditions points in the cost function, as the residuals will be zero in each of these points.

Step 2. We initialize weights as we would with a standard neural network, and using the cost function 2, evaluate our first trial solution.

Step 3. From here we back-propagate and use our preferred method for gradient descent, as we would with a standard neural network, to update our set of weights and bias terms. If we have a stopping criterion, we terminate whenever our cost is sufficiently small.

Step 4. Using our trained set of weights and bias terms, we now compute the final trial solution. Since we have an analytical solution for this problem, an easy measure of the network accuracy is to compare the analytical solution with the trial through the mean squared method.

2.4. Interpretability and Non-linear Least Squares

Our PINN gives us a set of N points on the form $((x_i, t_i), y_i)$, meaning we are able to plot our result in comparison with the analytical solution. Although this gives a good impression of how well our model has performed, it does not give us a closed form expression for the approximated solution. When solving a differential equation, not getting an expression for the solution could feel somewhat unsatisfying. A natural approach to translating our plot to an analytical expression is *nonlinear least squares.* We know that the analytical solution takes some form:

$$u(x,t) = Ae^{Bt}\sin(Cx)$$

for $A, B, C \in \mathbb{R}$. Hence, by defining the regression predictions as $\hat{y}_i(A,B,C) = Ae^{Bt_i}\sin(Cx_i)$, and defining residuals

$$r_i(A,B,C) = \hat{y}_i(A,B,C) - y_i,$$

we get the nonlinear least squares problem

$$\min_{A,B,C\in\mathbb{R}} \frac{1}{N}\sum_{i=1} r_i(A,B,C)^2.$$

Hence, by solving this problem, we should get a good approximation of our PINN output, on the same form as the analytical solution. A problem on this form can easily be solved using the Pytorch library, as will be discussed in our Implementation section.

3. IMPLEMENTATION

The field of computer programming has been transformed during the last decade due to increased automation, a rapid growth of programming language libraries, and big data, and this has brought an influx of effective tools still under development. In our project we chose the *Pytorch*[4] open-source deep learning library to structure our program around, due to the high degree of optimization found its algorithms.

In this section, we will consider the specific implementation of our network and the forward Euler algorithm. Furthermore, we consider practical aspects of programming with Pytorch, such as the inclusion of GPU. We also consider benchmark testing and give accounts for testing modalities. Finally, we discuss our use of AI tools.

3.1. Methods and data

The next natural step after considering the theoretical underpinnings related to our study, is to transform it into working code for testing and verification. We will begin by writing pseudocode to translate the mathematics from section 2 into high-level abstraction, and implement this in Python algorithms.

Algorithm 1 Explicit Forward-Euler algorithm

```
Require: position points $x_i$, $i = 0,\dots,N_x$; time points $t^n$,
    $n = 0,\dots,N_t$; initial condition $u_0(x)$; boundary data.
 1: Set $\Delta x = x_{i+1} - x_i$, $\Delta t = t^{n+1} - t^n$
 2: for $i = 0,\dots,N_x$ do
 3:     $u_i^0 \leftarrow u_0(x_i)$
 4: end for
 5: for $n = 0,\dots,N_t - 1$ do
 6:     impose boundary values $u_0^n$, $u_{N_x}^n$
 7:     for $i = 1,\dots,N_x - 1$ do
 8:         $u_{xx,i}^n \leftarrow \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$
 9:         $u_i^{n+1} \leftarrow u_i^n + \Delta t\, u_{xx,i}^n$
10:     end for
11: end for
12: return $\{u_i^n\}$
```

Algorithm 1 describes the explicit Forward Euler method for solving the one–dimensional heat equation. The solution is initialized from the given initial condition and then evolved forward in time. At each time step, the curvature of the solution in space is approximated using neighboring points, and this information is used to update the solution at the next time step. Repeating this procedure produces a numerical approximation of the temperature distribution over time.

Considering the neural network, in our use-case we do not have any labeled data for the training and testing of the network. Our algorithm for the network simply returns all weights based on backpropagation. This implies that our method is not supervised, but *self-supervised* (see Bergmann [5]). However, since we have a trial function to begin with we have a *signal* that is nudging the training in the right direction. We do not have any reason or way of doing a "conventional" full feed-forward pass to get predictions as a result of this.

When it comes to data, we have two intervals, $x \in [0,1]$, and $t \in [0,T], T \geq 0$. We can also adjust the *resolution*,

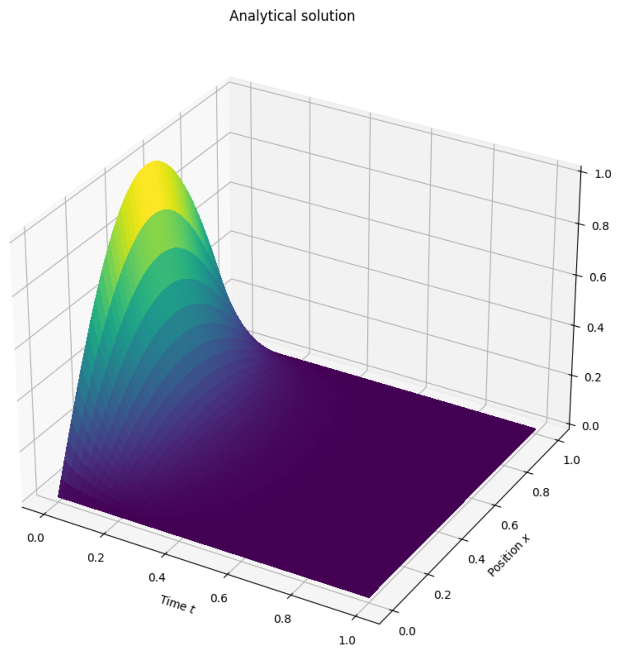

Figure 1: Example of analytical solution with resolution=100, and $t \in [0, 1]$.

i.e., the amount of points the function is evaluated on. We have an analytical solution which we compare our solution from the network to, plotted in Figure 1.

Our methods for implementing a physics-informed neural network are adapted from *The diffusion equation* in Hjorth-Jensen [2]. Although the cited implementation does not use Pytorch, the logic between that and the one of ours, is preserved. Algorithm 2 is a high-level explanation of how our network is implemented.

Algorithm 2 Class `PINN` for the diffusion equation

```
Require: network architecture; activation function; initial condition $u_0(x)$; collocation set $\mathcal{S}$
    Method Initialize(architecture, activation)
 1: Build neural network $N_\theta(x,t)$
 2: Initialize parameters $\theta$
    Method TrialSolution$(x,t)$
 3: return $g(x,t) = (1-t)u_0(x) + x(1-x)t\, N_\theta(x,t)$
    Method Residual$(x,t)$
 4: Compute $g_t = \partial_t g(x,t)$
 5: Compute $g_{xx} = \partial_{xx} g(x,t)$
 6: return $r(x,t) = g_t - g_{xx}$
    Method Loss$(\mathcal{S})$
 7: Evaluate residuals $r(x_i,t_i)$ for $(x_i,t_i) \in \mathcal{S}$
 8: return $\mathcal{C} = \frac{1}{|\mathcal{S}|}\sum_i r(x_i,t_i)^2$
    Method Train$(\mathcal{S}, \eta, K)$
 9: for $k = 1, \dots, K$ do
10:     Compute loss $\mathcal{C}(\theta)$
11:     Compute gradient $\nabla_\theta \mathcal{C}$
12:     Update parameters $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{C}$
13: end for
    Method Predict$(x,t)$
14: return $g(x,t;\theta^\star)$
```

3.2. Choice of parameters and testing procedure

Mathematics provides an elegant framework for establishing theoretical results, often relying on idealized models to clarify relationships between quantities and underlying structures. By contrast, the practical side of machine learning is largely experimental, emphasizing implementation, optimization, and systematic testing rather than purely analytical guarantees. One must therefore actively ensure that the data is correctly initialized, shaped, and of proper type to propagate through the network and obtain predictions. We will briefly showcase our reasoning behind the choices related to these issues.

Since digital computers operate on discrete logical units, we must *discretize* our model space. We initialized the collocation points, being the points in where the function is evaluated both *uniformly* and *randomly*. The former meaning that the distance between the points in the input space are equal across the whole interval, and the latter that the points are randomly distributed within a defined interval.

Curvature is an interesting property of our heat equation, since in points of higher curvature, the rate of temperature change will be higher than lower curvature. We focus on choosing values of t that both maximize and minimize this curvature. The curvature is given by the second derivative with respect to spacial coordinates, x, of our equation $u(x,t)$. We have the two following optimizations, given by the second derivative

$$\frac{\partial^2 u(x,t)}{\partial x^2} = -\pi^2 e^{-\pi^2 t} sin(\pi x) = 0,$$

$$\underset{x\in[0,1]}{\arg\max} \left| -\pi^2 e^{-\pi^2 t} \sin(\pi x) \right| .$$

We chose values $t1 \in \{0.01, 0.05\}$ and $t2 \in \{0.25, 0.40\}$ for maximizing and minimizing curvature, respectively.

Finding the best model reduces to finding the hyperparameters corresponding to the best performance, in our case being the deviation from our model to the analytical solution, and to the Euler scheme. The hyperparameters relevant for this research project, is learning rate η, depth and width of network (memory), number of epochs, number of collocation points, the interval of t, and whether the values are randomly or uniformly distributed. To find the set of all these, we do classical grid search as done in the previous projects (see full description in Alsvik et. al. chapter 3 [1]). We also use different activation functions, being sigmoid, tanh, ReLU, and GELU to test which one yields the highest performance. We also consider the time complexity of the different hyperparameters and activation functions. In this research, we limit ourselves to use only one optimizer, being ADAM, since this has been proven to give consistently better results than the other optimizers such as RMSprop and AdaGrad. This is discussed in Section 4.3 (4).

3.3. Remarks on high-level deep learning

The use of *Pytorch* and *autograd* for implementing the PINN, has alleviated some of the tediousness of implementing a neural network from scratch, as done in Alsvik et. al. [1]. However, it has also introduced some unfamiliar concepts and possibilities.

An obvious advantage of using deep learning libraries, is the time-saving of (i) writing the code, but also (ii) testing that the code works. This might, however, be a double edged sword, since the level of abstraction might be too high in order to understand the implementation fully. Nevertheless, when having a decent understanding of the workings of a neural network, there's no need to actually implement a backpropagation method. Instead, we can simply call the method *backward* on a cost function object, *loss*, and obtain the gradients (see listing 1).

Listing 1: PINN training w/PyTorch

```python
def train_pinn(self, x, t, optimizer,
    epochs):
    optimizer = torch.optim.Adam(model.
        parameters(), lr=lr)
    for e in range(epochs):
        optimizer.zero_grad()
        loss = model.loss(x_coll, t_coll)
        loss.backward()
        optimizer.step()
```

All in all, this approach is more pragmatic than writing our own network. Although, it is important to have a solid enough background to understand what *Pytorch* and *autograd* does.

Another interesting integration by *Pytorch*, is the use of NVIDIAs GPU interface, *cuda*. This makes it simple for computers with graphical processing units to run code faster and more efficient, which is crucial for deep learning models. The test code for this research runs on NVIDIAs processor, *Geforce RTX*.

With *Pytorch* being the benchmark network in our previous research, see Alsvik et. al. [1], it is now the network itself, assuring solid and consistent results. Therefore, we can allow ourselves to focus more on the problem we are solving along with the data, rather than the actual network setup.

3.4. Presenting the solution

Since our main objective is to find a function approximating the analytical solution to the diffusion equation, a closed form expression of this function will be more informative than a mere comparison plot (see 2.3 2). Listing 2 shows the implementation in Python for this method. We use scientific python, *scipy*, to do a nonlinear least squares regression. The method *least_squares* solves a *nonlinear least-squares problem with bounds on the variables.* We use "trf" as method, which is a common robust method for *large sparse problems with bounds.* This is taken from the scipy API, see [6]. We pick random first guesss for *theta*, and find that for enough epochs, the impact of the choice of theta values being initiated is negligible. Calling x from the least squares object, *result*, we get the optimal coefficients A,B, and C.

Listing 2: Nonlinear least squares w/scipy and numpy

```python
from scipy.optimize import least_squares
import numpy as np

def fit_ABC(x, t, u, theta0):
    # theta = (A, B, C)
    def residual(theta):
        A, B, C = theta
        return A*np.exp(B*t)*np.sin(C*x) -
            u

    # minimize sum_i residual_i(theta)^2
    result = least_squares(residual, theta0
        , method="trf")
    return result.x
```

In order to visualize how the solution evolves over time, we implemented 3D animations. Listing 3 outlines how the animations are implemented. The solution $u(x,t)$ is first computed on a fixed grid using NumPy, and Matplotlib is then used to visualize its time evolution. The animation is created with `FuncAnimation`, where either the surface colours of a fixed cylindrical rod updated for each time step or the solution for each time step similar to the one in Figure 1.

Listing 3: 3D animation with matplotlib

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
from mpl_toolkits.mplot3d import Axes3D

x, t = spatial_grid(), temporal_grid()
U = compute_solution(x, t)

fig, ax = plt.subplots(subplot_kw={"
    projection": "3d"})

def update(k):
    plot_3d(ax, x, t[k], U[:, k])
    return ax

animation.FuncAnimation(fig, update, frames
    =len(t))
```

3.5. Use of AI tools

We used ChatGPT as a sparring partner for discussion of the theory, as well as for testing our intuition. In the Theory and Methods section, it was also used to clean up spelling mistakes, clarify arguments and to make general improvements in the text. Other than this, ChatGPT have been used for help with the coding, particularly with finding errors in our code and debugging. We've also gotten advice from ChatGPT on how to do animations, but read the documentation ourselves to understand how it worked.

It was also used for help with formatting the project in Overleaf. This time, we also used ChatGPT for generating docstrings on our methods and classes, since this does not bring any interesting insights. All in all, for tedious- and less intellectually stimulating tasks, we consulted with an LLM to automate this, saving us time. We have included an overview of the prompts and replies we got from ChatGPT in our Github repository (see the folder *LLM* in Alsvik et. al. ([7]).

4. RESULTS AND DISCUSSION

4.1. Numerical stability of forward Euler

As mentioned in the theory section 2, the forward Euler algorithm is prone to numerical instability for given choices of parameters. We define a stability constant α to equal $\frac{\Delta t}{\Delta x^2}$. For $\alpha \leq 0.5$, the algorithm is stable. We consider cases where we force instability and compare results for varying values of t, being high vs. low curvature.

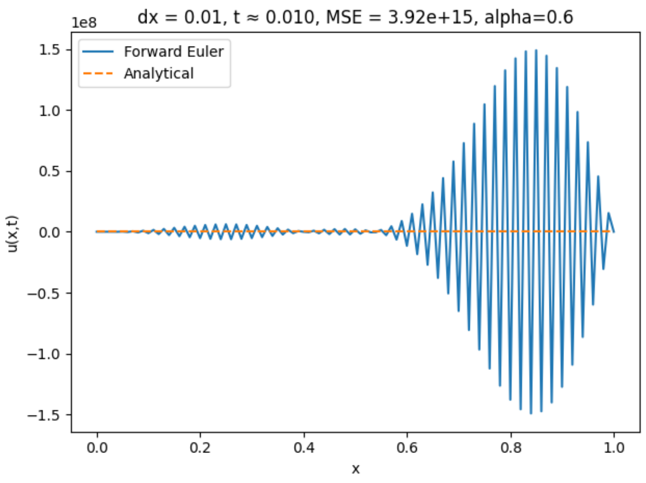

Figure 2: Unstable Euler scheme with $\alpha = 0.6, \Delta x = 0.01$, and $t \simeq 0.01$.

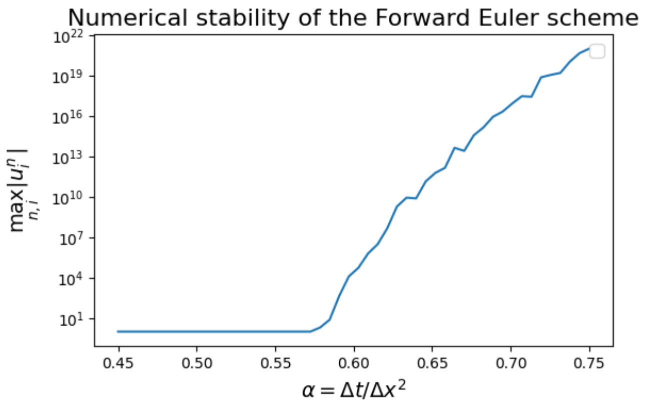

Figure 3: Max absolute value of the temperature function plotted against the stability constant, $\alpha \in [0.45, 0.75]$.

Figure 2 shows a case of an unstable Euler scheme, with $\alpha = 0.6$. We note that the value of $u(x,t)$ are blown up significantly, by eight orders of magnitude higher than for the analytical solution. This inflating u-value is particularly visible for larger values of x. Figure 3 shows the relationship between values of α and numerical instability. We can see that for α equal to about 0.58, the algorithm becomes unstable and its

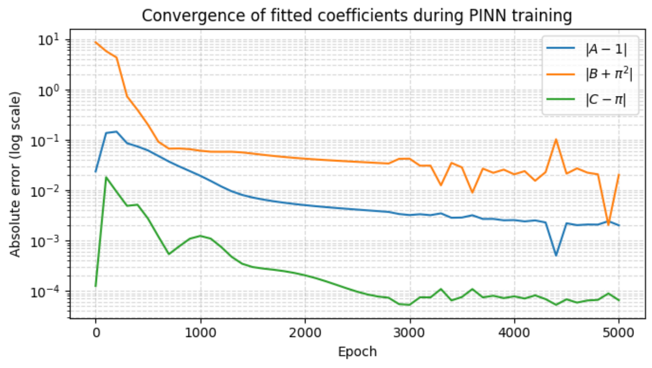

Figure 4: Error convergence for the coefficients A,B and C as function of number of epochs, trained by the PINN. $\eta = 1e{-}4$, 5000 epochs, 5 hidden layers with 200 nodes in each layer, and GELU as activation function.

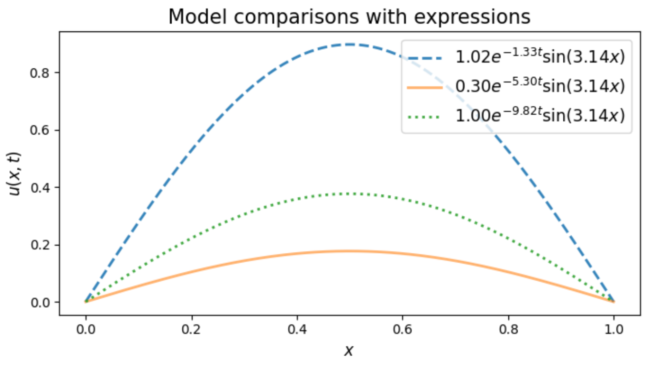

Figure 5: Different models compared with respective expressions for $t = 0.1$. The dashed blue line represents a PINN model that is trained for only 10 epochs, the dashed green for the *optimal* PINN model with 5000 epochs whereas the orange line is for Euler with $\alpha = 0.51$. We use 100 collocation points as default.

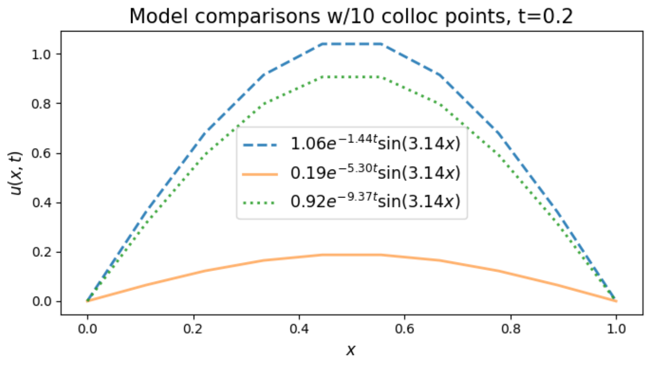

Figure 6: Different methods with respective expressions for t=0.2 and $\alpha = 0.49$

. The dashed blue line represents a PINN that is trained for only 10 epochs, and the dashed green is the optimal PINN model with 5000 epochs. The orange line represents the Euler algorithm with 10 collocation points.

instability increases drastically for higher α-values.

Another flaw of the Euler Scheme is the need for many collocation points to produce the wanted results. We found that for lower collocation points, the network performs steady, but the Euler algorithm becomes very unstable. We will see a concrete example of this in the next subsection.

4.2. PINN results and discussion

We move on to the performance of our PINN network. For values of α less than or equal to 0.5, the forward Euler algorithm performs notably better than the best PINN model, with 1.84e-14 vs. about 2e-08. That being said, it is interesting to consider shortcomings and advantages for the two methods.

The coefficients for the nonlinear least squares regression was randomly initialized at first, figure 4 shows how the absolute error between the predicted coefficients, $\hat{\theta}$, and the true coefficients of the analytical function, θ, changes as function of epochs. The observed differences in both convergence behavior and absolute error may be related to what the coefficients represent in the analytical expression. The amplitude A mainly sets the overall size of the solution and is therefore relatively easy for the network to adjust, although it settles at a higher error since it can partially compensate for inaccuracies in the time dependence. The spatial coefficient C converges quickly and to a lower error, which is reasonable given that the spatial shape of the solution is strongly constrained by the boundary conditions. In contrast, the coefficient B influences how fast the solution decreases as time increases and appears harder to determine precisely. Since even small inaccuracies in B can lead to noticeable differences in the solution at later times, this may help explain why it converges more slowly and remains at a comparatively higher error. However, this being said, it is hard to understand what is producing the prediction of the model in such detail.

Figure 5 shows different models with corresponding expressions on the form $Ae^{Bt}sin(Cx)$. The *untrained* PINN model has an amplitude of $A = 1.02$ but since the time decay, $B = -1.33$, is different than the trained PINN with $A = 1.00$ and $B \approx -\pi^2$, the expressed amplitude in the plot would not be equal since t is different from zero. The Euler scheme gives a worse prediction for $\alpha = 0.51$, likely since its instability has begun to manifest. Since our trial function has defined C to be π, the nonlinear regression does not change this parameter visibly for different models, despite initializing C with a random value. In our Figures folder in Alsvik et. al. [7], there are examples of varying t,- and α-values with prefix *model_comps*.

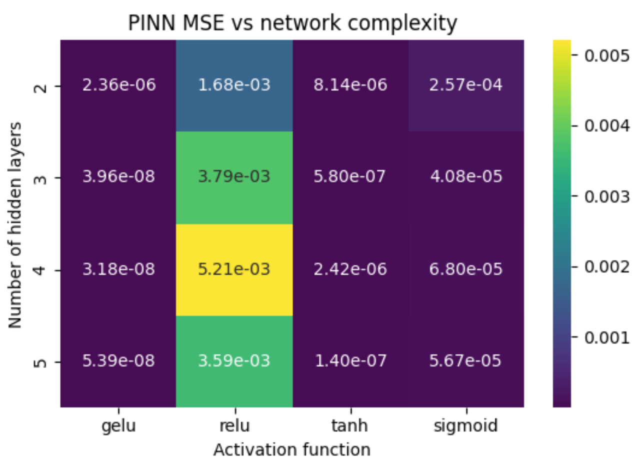

Figure 7: MSE plotted as function of network depth (no. hidden layers) and different activation functions. $\eta = 1e - 3$, epochs=1500 nodes=128, and collocation points=200.

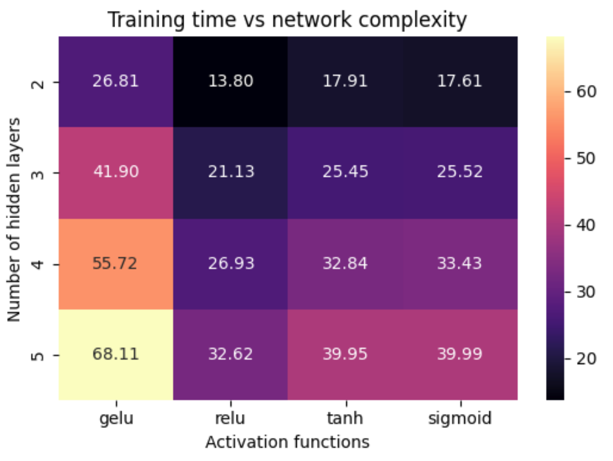

Figure 8: Training time plotted as function of network depth and different activation functions. $\eta = 1e - 3$, epochs=1500, nodes=128, and collocation points=200.

Figure 6 shows the instability of the Euler algorithm, when using only ten collocation points. The model uses $\alpha = 0.49$, which is evidently stable, but predicts a function very far off our true analytical solution. Interestingly, this shows that our PINN needs few collocation points to train on to still perform very well, whereas the Euler needs significantly more. Figures 7 and 8 shows performance and training time respecitvely for different activation functions in the hidden layers. As we can see, *GELU* reaches by far the best performance with MSE=3e-08 for 4 hidden layers. Despite *GELU* being the slowest in training time, it converges quickly to respectable performance. Although *tanh* is noticably faster, it does not produce better performance, and is actually a little unstable for

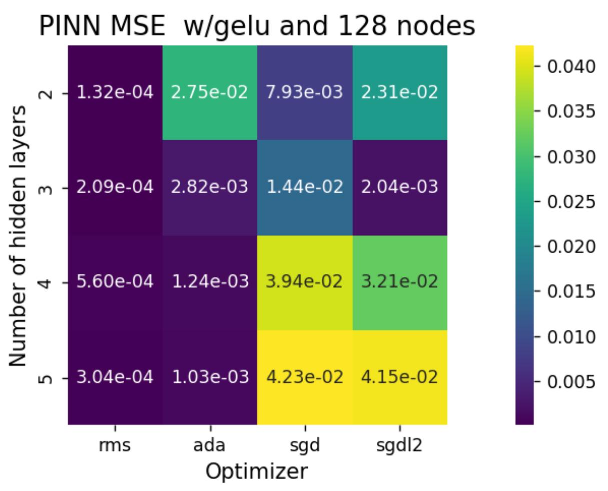

Figure 9: MSE as function of depth of the network and various optimizers. Used fullbatch SGD with and without the use of regularization. With L2, we used $\lambda = 1e - 03$.

deeper configurations of the neural network. *ReLU* is the fastest, as expected, since its computation is very cheap, but its performance is significantly worse than the other activation functions, and interestingly, it does not improve with deeper configuration, but performs worse as the number of layers increase. We therefore used *GELU* on a consistent basis, since it serves the performance and stability we looked for.

4.3. Stability of gradient optimizers

Figure 9 shows the various optimizers other than our default, *ADAM*, which is displayed in the first column of Figure 7.The optimizers are initialized by default hyperparameters, meaning $\rho = 0.99$ for RMSprop. We use the best activation function, *GELU*, to compare the optimizers. SGD is by default fullbatch in Pytorch, and using smaller batches would likely only influence the training time and not the actual performance. We note that SGD with the use of L2-regularization with $\lambda = 1e - 03$ gives slighly better performance for 3 layers, but no consistent improvement for a deeper network. This suggests that SGD isn't particularly *stable*, since it does not improve with higher network complexity, and that regularization performs better than without. *RMSprop* and *AdaGrad* achieves a decent performance, but not close to ADAM. AdaGrad does at least improve over network complexity, but still does not reach the performance of RMSProp. Although RMSprop reaches the best performance out of the various optimizers, it does not improve with network complexity. This might suggest that the data the model trains on isn't complex enough to require a very deep configuration. We can also see the likes of this with ADAM w/GELU in figure 7, where 5 layers performs worse than 4 layers. With this in mind, it is fair to say that medium complexity such as 3 or 4 layers with 128 nodes using ADAM w/GELU is the sweetspot between the tradeoff of performance and time.

4.4. Modeling the rod

An important part of inference and prediction, is to model our solution in such a way that it provides an informative and visual understanding of the physical system at hand. Although mathematical equations and figures showing the relationship between different parameters in our model is informative, a visual representation of the solution might give the best intuition of how the system evolves. Figure 10 and 11 shows the temperature of the rod as function of time, t, and space, x. The temperature is plotted as color, corresponding to a physical temperature given by the colorbar to the right of the rod. Note that the y-axis comprises of *dummy points* to represent the rod's geometry, and that the PINN is still trained on one spacial dimension. The rod model in the two figure is the optimal PINN model, based on MSE from model selection. It has the hyperparameters described in the caption for Figure 4. In Figure 10, when $t = 0$, we can see that the temperature is significantly higher in the middle than the one of Figure 11, and decreases smoothly towards the ends. We see the effect of temperature decay in Figure 11, where the overall temperature decreases and becomes more uniform over time.

In the Figures folder of our GitHub repository (see Alsvik et. al., [7]), you can see the full 3D animation by clicking *heat_pole.gif*, being the 3D model of the rod in Figure 10 and 11, and *heat_rod.gif*.

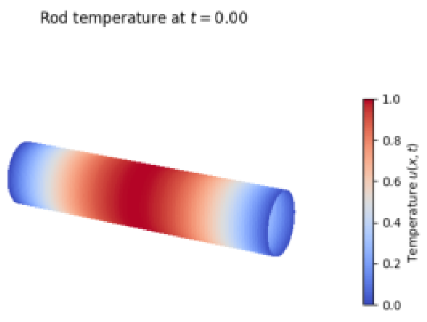

Figure 10: Initial thermal distribution in a 2D rod, i.e. t=0.0s for optimal PINN model.

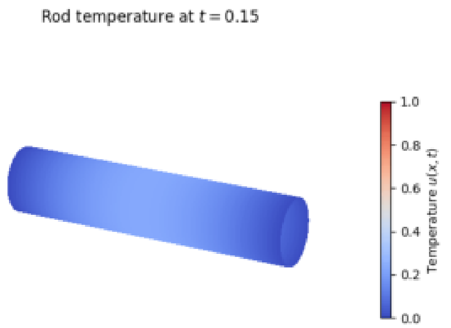

Figure 11: Thermal distribution in a 2D rod when t=0.15s for optimal PINN model.

5. CONCLUSION AND FUTURE IMPLICATIONS

For the research, we presented four categories of results. We looked at the stability of the Euler algorithm by varying the stability constant, α. For time step $\Delta x = 0.1$, we found that α-values larger than 0.58 blew up the values of the temperature $u(x, t)$ exponentially. We also saw that the temperature was oscillating frequently on specific values of x, by using $\alpha = 0.6$, and $\Delta x = 0.01$. Also, for fewer collocation points the Euler scheme becomes unstable.
We proceeded to compare solutions for the Euler scheme and the PINN, and to look at perfomance and time as function of network complexity for the PINN. For $\alpha \leq 0.5$, the Euler's algorithm performs significantly better than the PINN model, by as much as six orders of magnitude, e-14 for the former vs. e-08 for the latter. However, the best PINN model is able to approximate the expression for the analytical solution to arbitrary detail, and is stable across multiple parameters, using ADAM optimizer. We found that GELU was the best fit as an activation function despite higher time-consumption.

We looked at different gradient optimizers as function of network complexity, and found ADAM to stand out in performance and stability. Interestingly, the other optimizers did not improve as complexity grew. This was most notable for fullbatch SGD and RMSprop. Also, regularization did not seem to have the expected effect, as it actually gave better performance for full-batch SGD than without the use of regularization. In conlusion, since complexity does not converge toward the performance of Euler's method, our data might not be complex enough to perform better with a deeper neural network.

Lastly, we showed the results for modeling the rod in two different ways, with using 3D animations.

Despite testing on many different PINN models, we did not show better performance than for the forward Euler algorithm. Nevertheless, the function expression for PINN is still very close to that of the analytical solution, and this might limit the actual emphasis on MSE explaining how good the solution is. Also, the forward Euler is a more specialized algorithm for solving our particular problem, whereas the neural network is a more general problem solver, stable across many parameters.
For more complex differential equations such as the *Navier Stokes equation* and *Schrödinger's equation*, forward Euler might have struggled more visibly. It is too reductive to conclude that forward Euler solves the diffusion problem better than PINNs solely based on our research. The interesting part of the research was namely to employ a PINN to solve a differential equation, and our model did so accurately. We consider this an exciting result in itself.

[1] S. K. Alsvik, Amund Solum and J. Vestly, Project Report FYS-STK4155 Project 2, University of Oslo (2025).

[2] M. Hjort-Hensen, *Lecture notes: Week 43 – deep learning: Constructing a neural network code and solving differential equations*, `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week43.html#solving-the-equation-using-autograd` (2025), accessed: 2025-05-12.

[3] N. Andre Jürgen Massing, Department of Mathematics, *Numerical methods for the heat equation*, `https://www.math.ntnu.no/emner/TMA4125/2021v/lectures/NumericalMethodsHeatEquation.pdf` (2021), accessed: 2025-12-08.

[4] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, in *Automatic differentiation in PyTorch* (2017).

[5] Dave Bergmann, *What is self-supervised learning?*, `https://www.ibm.com/think/topics/self-supervised-learning`, accessed: 2025-12-03.

[6] SciPy Developers, *scipy.optimize.least_squares* (2024), accessed: 2025-01-10, URL `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html`.

[7] A. Alsvik, J. Vestly, and Kristoffer DH Stalker, *Project source code*, `https://github.com/JVestly/Project-3` (2025).