Numerical and Machine Learning Approaches to Solving the One-Dimensional Diffusion Equation

FYS-STK4155

Selma Beate Øvland
University of Oslo

(Dated: December 18, 2025)

We investigate the application of deep neural networks for solving differential equations, with a focus on the one-dimensional diffusion equation. Differential equations (DE) problems have significant scientific interest, spanning from fluid mechanics to quantum systems, and are traditionally solved using numerical discretization methods. We compare the performance of a Physics-informed Neural Network (PINN) solution with that of the established explicit forward Euler finite difference scheme and the analytical solution. We evaluate and compare the accuracy, stability, and computational behavior of the PINN and Euler methods. Our results show that PINNs can achieve high accuracy and low error compared to the explicit Euler scheme, but at the cost of increased computational complexity and sensitivity to hyperparameter selection. The study highlights both the potential and the limitations of PINNs as a tool for solving PDEs. The results indicate that PINNs are better for problems where traditional grid-based methods become impractical, rather than for simple low-dimensional PDEs.

I. INTRODUCTION

Neural networks (NNs) are sophisticated computational systems that have demonstrated the ability to learn complex tasks. The Universal Approximation Theorem [1] [2] confirms that feed-forward neural networks (FFNNs) can approximate any continuous function to arbitrary accuracy, provided the network has sufficient complexity. This versatility has given NNs popularity across different natural sciences, including the solution of fluid dynamics problems and the study of quantum systems [1].

This project focuses on solving differential equations using machine learning methods. The diffusion equation is a standard partial differential equation (PDE) used to model the gradual spread of heat, particles, or chemicals through a material. In this project, we consider the one-dimensional diffusion equation

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

We approach this problem in two stages:

1. Standard Numerical Approach - The solution provided by the explicit forward Euler finite difference scheme, a conventional numerical technique for PDEs, requiring discretization of time and space.

2. Deep learning Approach (PINN) - The solution derived from a neural network explicitly trained to satisfy the PDE, using methods such as automatic differentiation.

Traditional numerical solvers, such as finite difference methods, rely on explicit discretization of space and time into a grid, which introduces some limitations. For the method to be stable, the relationship between spatial and temporal step size needs to be respected through what is known as Courant-Friedrichs-Lewy (CFL) condition, shown in equation 8. To avoid instability and divergence, these relationships must be respected, which requires taking small time steps, a computationally expensive approach. The methods are also susceptible to truncation errors, because the derivatives are approximated using finite differences and are not exact. Reducing either the spatial or temporal time step reduces this error, but the runtime greatly increases. If we were to evaluate a PDE at a higher dimension than 1D, the grid size could be huge, which significantly affects computational cost and memory usage.

Physics-informed Neural Networks (PINNs) offer an alternative method where the solution is represented as a continuous function approximation. Instead of computing u(x,t) at fixed points, the network learns a function defined everywhere in the domain. As classical solvers can suffer from an exploding grid at higher dimensions, PINNs do not require structured spatial grid, but instead evaluate the PDE residual at randomly sampled collocation points. These methods are suitable for high-dimensional PDEs, irregular geometries, or problems where grid generation is difficult. But they also have some challenges. PINN performance is sensitive to architecture choices, can suffer from exploding or vanishing gradients, and convergence can be very slow. We define a trial solution, $g_t(x,t,P)$, which satisfies the initial and boundary conditions of the diffusion equation. This trial function incorporates the output of the deep neural network $N(x,t,P)$, whose weights and biases (P) are iteratively optimized. The optimization procedure involves defining a cost function $C(P)$ based on the PDE residual obtained when the trial solution is substituted back into the PDE itself, often corresponding to the MSE of this residual. The network then minimizes $C(P)$ using gradient descent and automatic differentiation through Keras [3] [4].

Both classical numerical solvers, such as the Euler scheme, and PINNs have their advantages depending on the situation. In this project, we aim to explore when PINNs provide better performance than the classical Euler scheme, or when the PINN fails. The objective of this project is to implement both conventional and deep learning solutions and provide a critical assessment comparing their results, numerical stability, and computational requirements, particularly regarding the impact of varying network complexity (e.g., number of hidden layers, nodes, and activation functions) on the solutions' accuracy.

II. THEORY AND METHODS

A. Problem definition and Analytical Solution

The core problem involves solving the one-dimensional diffusion equation on a rod of length L = 1. This equation models quantities such as temperature or concentration as a function of space (x) and time (t).

1. Governing Partial Differential Equation (PDE)

The PDE states that the second spatial derivative of $u(x,t)$ equals its first time derivative:

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, \qquad t > 0,\ x \in [0, L] \tag{2}$$

2. Initial and Boundary Conditions

The solution must satisfy:

- Initial Condition (IC) (at t = 0):

$$u(x,0) = \sin(\pi x), \quad 0 < x < 1 \tag{3}$$

- Boundary Conditions (BCs) (at x = 0 and x = 1): The temperature (or concentration) at the ends of the rod is fixed at zero:

$$u(0,t) = 0, \qquad u(1,t) = 0, \qquad t \geq 0 \tag{4}$$

3. Analytical solution

For this specific set of initial and boundary conditions, the exact solution is a single decaying Fourier mode:

$$u(x,t) = e^{-\pi^2 t} \sin(\pi x) \tag{5}$$

The analytical solution is crucial for validating the accuracy of the numerical and neural network methods.

B. The Explicit Forward Euler Scheme

The numerical solution of the PDE uses the explicit forward Euler algorithm. This is done by converting the continuous PDE into a discrete difference equation using spatial and temporal grids.

1. Discretization

To obtain a numerical solution, we discretize the PDE. We define a uniform grid for space and time:

- **Space grid:** $x_i = i\,\Delta x$, where $i = 0, 1, \ldots, N_x$ and $\Delta x = \frac{1}{N_x}$.
- **Time grid:** $t_n = n\,\Delta t$, where $n = 0, 1, \ldots, N_t$.
- The numerical solution at a grid point is denoted $u_i^n \approx u(x_i, t_n)$.

2. Finite Difference Approximations

The derivatives in the PDE is replaced by finite difference approximations:

- **Time derivative** u_t (forward difference): This uses the values at times t_n and t_{n+1}:

$$u_t(x_i, t_n) \approx \frac{u_i^{n+1} - u_i^n}{\Delta t}.$$

- **Spatial second derivative** u_{xx} (centered difference): This uses the values at x_{i-1}, x_i, x_{i+1} at time t_n:

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

3. Explicit Update Formula

Substituting these approximations into the PDE $u_t = u_{xx}$ and solving for u_i^{n+1} gives the update equation:

$$u_i^{n+1} = u_i^n + \frac{\Delta t}{\Delta x^2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right). \tag{6}$$

Defining the diffusion number $\alpha = \frac{\Delta t}{\Delta x^2}$, the formula simplifies to:

$$u_i^{n+1} = u_i^n + \alpha\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \qquad i = 1, \ldots, N_x - 1. \tag{7}$$

4. Stability Condition

For the explicit forward Euler method to be numerically stable, the diffusion number α must satisfy the condition:

$$\alpha \leq \frac{1}{2}, \ \frac{\Delta t}{(\Delta x)^2} \leq 0.5 \tag{8}$$

Even when this stability condition is satisfied, every forward Euler step introduces error from the derivative approximation. Because Euler is a first-order accurate method in time, the errors grow with each iteration, never cancel out, and scale linearly in time. The longer we simulate, the more the error accumulates. Euler also tends to smooth the solution over time, which can make the peak of the solution appear lower than it should be, make the solution decay faster than the analytical solution, or reduce curvature. This is known as numerical diffusion [5].

C. Neural Network Method

The solution using neural networks relies on a method known as Physics-informed Neural Networks (PINNs), which doesn't directly output the solution. Instead, PINN learns a function $u(x,t)$ by minimizing the residual error of the PDE, not by discrete approximations. Unlike supervised learning models, which require labeled input-output pairs, PINNs integrate physical laws (typically expressed as differential equations) directly into their loss function, altering both how they learn and what data they need. Instead of matching provided examples, PINNs are trained to produce outputs that satisfy these equations (with boundary/initial conditions) [6]. We use TensorFlow to compute derivatives automatically.

1. The trial solution

The solution $u(x,t)$ is approximated by a trial solution $g_t(x,t,P)$ designed to satisfy the initial and boundary conditions. The trial solution structure involves the output of a deep neural network $N(x,t,P)$, parameterized by weights and biases P [3]:

$$g_t(x,t) = h_1(x,t) + h_2(x,t,N(x,t,P)) \tag{9}$$

For the diffusion equation with the given conditions $u(x,0) = \sin(\pi x)$ and $u(0,t) = u(1,t) = 0$, the appropriate trial solution is [3]:

$$g_t(x,t) = (1-t)\,u(x) + x(1-x)\,t\,N(x,t,P) \tag{10}$$

- The term $x(1-t)u(x)$ ensures the initial condition $u(x,0) = sin(\pi x)$ is met.
- The term $x(1-x)tN(x,t,P)$ ensures the overall solution satisfies the boundary conditions at $x = 0, x = 1$ (because of to the $x(1-x)$ factor). The initial condition at $t = 0$ ensures that the neural network contribution is zero under the specified conditions.

This is implemented in our code **PINN.py** in the function **g_trial_tf**:

```
h1 = (1.0 - t) * u(x)
h2 = x*(1-x)*t*N_val
g = h1 + h2
```

The choice of the trial solution plays a crucial role in the success of PINNs. A correct construction of the trial solution means that PINN never violates the initial or boundary conditions, because they are built into the form of the solution. If the network had to learn the boundary and initial conditions, it would require a more complex model and more training. When they are embedded into the trial solution, the optimization becomes simpler (with less training, fewer parameters, and fewer collocation points) and converges faster. This, however, can reduce flexibility and prevent the network from learning the true solution if the PDE was more complex.

2. Cost Function

The goal is to adjust the network parameters P such that the trial solution g_t satisfies the differential equation 1. The residual error f of the PDE is defined as:

$$f(x,t,g_t,g_t',\ldots) = \frac{\partial g_t}{\partial t} - \frac{\partial^2 g_t}{\partial x^2} \approx 0 \tag{11}$$

We compute the loss as:

$$L = \sum_{i=1}^{N} \left[g_t(x_i,t_i) - g_{xx}(x_i,t_i) \right]^2, \tag{12}$$

Where derivatives are obtained automatically via TensorFlow's **GradientTape**.

D. Neural Network Architecture using Keras

The neural network is created using Keras for a simplified process. It provides the **Sequential** model class, which allows layers to be easily appended to build standard feed-forward networks. It also includes the **Dense** layer class for defining fully connected layers, which is useful in our user case, as we need to map input coordinates x and t to an output N. This is implemented in the **create_network_model** function:

```
model = Sequential()
model.add(tf.keras.Input(shape=(input_dim,)))
model.add(Dense(layers, activation=activation))
model.add(Dense(output_dim, activation='linear'))
```

Keras/TensorFlow also allows us to specify activations for layers easily and offers support for various components needed for complex training, such as optimizers and regularizers.

E. Handling Complex Differentiation

The key advantage of using TensorFlow is automatic differentiation. TensorFlow also provides significantly better execution time than custom NN code, as our search space, defined by the network's hyperparameters and weights, can be massive, leading to long runtimes. In this case, we minimize the cost function based on the residual of the PDE, by calculating high-order derivatives of the trial solution 10. TensorFlow provides efficient mechanisms for automatic differentiation (using **tf.GradientTape**) necessary for computing these complex high-order derivatives during the backpropagation and optimization stages [7] [3]. **tf.GradientTape** is implemented in our code:

```
#Second derivatives w.r.t. x
with tf.GradientTape(persistent=True) as tape2:
    ....
    #Inner tape to compute first derivatives
    with tf.GradientTape(persistent=True) as tape1:
    .....
#Second order derivatives, requires the derivatives of the inner tape
d2_g_t_d2x = tape2.gradient(d_g_t_dx, x)
```

As we require second derivatives, we need to use one tape to compute the first derivative, then another tape to take the derivative of that first derivative. In this codeblock we compute the partial derivative (u_t and u_{xx}) required for the PDE residual:

```
residual = d_g_t_dt - d2_g_t_d2x
loss = tf.reduce_mean(tf.square(residual))
```

The cost function is the MSE of the residual at randomly sampled collocation points; the difference between the time derivative and the second spatial derivative of the trial solution.

F. Custom Training Loop

Because we have a PINN model, we can't use **model.compile** or **model.fit**, since we don't have target values for training. We need to implement a custom training loop:

```
@tf.function #converts the function into a fast, optimized TensorFlow graph
def train_step(X_points):
    with tf.GradientTape() as tape:
    """ 1. Builds the trial solution g_t
        2. Computes g_t
        3. Computes first derivatives
        4. Computes second derivatives
        5. forms the PDE residual
        6. Returns the MSE
        """
        loss = compute_loss(model, X_points)
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    return loss

loss = train_step(X_train)
```

This code block performs one training iteration of our PINN model. It computes the loss and the gradients with respect to all network parameters, and then applies an optimizer. This replaces the more standard **mode.fit()** method, which is more commonly used in classification or regression problems. Adapted to our PDE problem, the function:

1. Evaluates how well the neural network satisfies the PDE at given points.
2. Calculates how changing each weight affects the PDE residual.
3. Adjusts the weights to reduce the residual.

We implemented early stopping for epochs with:

- Patience: 1000 epochs
- improvement threshold: $1x10^{-6}$

This prevents long plateaus or worsening of MSE after initial improvements. Weights are automatically restored to the best-performing epoch.

G. Comparison with Analytical Solution

To evaluate the trained network, we compute:

$$\text{MSE} = \frac{1}{N_x N_t} \sum \left(g_t(x,t) - u(x,t)\right)^2 \tag{13}$$

This is implemented in our **compute_MSE()** function. This separates **Residual Loss** (how well the PDE is satisfied) and **Solution MSE** (how well the PINN matches the analytical solution).

H. Complexity Study

When using PINNs, the network architecture and other hyperparameters affect the accuracy, stability, and convergence of the PDE solution. PINNs' performance depends heavily on the model design. Therefore, we test over several hyperparameters to find a configuration that is both accurate and robust. We evaluate our model using loss and MSE scores, in addition to testing it over several seeds to assess variance, stability, and general performance.

1. Width test

The width controls the capacity of the neural network. Too few nodes can result in underfitting, and too many nodes can lead to instability and a more difficult-to-train model as the number of parameters increases rapidly [7]. The largest width is not necessarily the best one, as PINN loss landscapes can become stiff with increased width. In such cases, the topography can be both steep and flat, making it challenging to find an appropriate step size.

2. Depth test

As stated earlier, the Universal Approximation Theorem states that a network requires only a single hidden layer to approximate any continuous function; increasing depth offers practical advantages. Together with width, depth determines the network's topology and complexity. For very complex problems, increasing the number of layers enables the network to learn abstract features; however, this can require a substantial amount of training data. For many problems, starting with one or two hidden layers is sufficient [4].

For deeper architectures, the challenge is often the unstable gradient problem, like vanishing gradients and exploding gradients. When training deep networks, the error signal becomes too small as it is passed backward through many layers, so the early layers barely get updated, causing learning to slow down or stop entirely. Exploding gradients is the opposite problem; the error signal becomes too large as it moves backward, leading to huge weight updates that make training unstable and cause divergence. Different layers can also learn at different speeds, resulting in uneven learning [4] [7]. Choosing the optimal depth is crucial.

3. Activation test

Activation functions affect the smoothness of the solution, the stability of the gradient backpropagation, and the shape of the function space the network can represent. With the activation function, we can introduce nonlinearity into the network, which is essential when we deal with solutions derived from differential equations [8].

4. Learning rate test

The learning rate determines the size of each optimization step. PINNs are sensitive to this parameter because their loss function combines PDE residuals, boundary conditions, and higher-order derivatives. A learning rate that is too high leads to unstable or diverging training, while a too low value can lead to slow convergence [4].

5. Optimizer test

PINN optimization can be challenging due to stiff gradient landscapes and the computation of high-order derivatives via automatic differentiation. ADAM is fast, but can be unstable for large models, while RMSProp can sometimes be more stable. These are the two optimizers we test. Optimizer choice can drastically change convergence behavior, and we examine which one best handles PDE loss and minimizes the residual reliably [7].

6. Collocation points

We conduct a test with varying numbers of collocation points, using uniform sampling, as the number and distribution of collocation points can strongly influence PINN performance. In regions with steeper gradients, there could be some benefit from a different distribution of sampling points if the model shows performance issues in these regions.

7. Testing across multiple seeds

Training PINNs involves randomness in:

- Weight initialization
- Sampling of collocation points
- Stochasticity of optimizers like ADAM

So a single run is not representative of true model performance. Different seeds can produce:

- Different convergence speed
- Different final MSE score
- Occasional training failures

We repeat each hyperparameter configuration across different seeds and compute mean performance (accuracy) and standard deviation (stability and robustness). This way, we select model configurations that perform consistently.

I. Use of AI

There is a folder in GitHub containing the ChatGPT prompts, which are mostly conceptual questions for understanding PINNs and are not directly related to this report. In addition to this, I have a Grammarly business account through my work that I use for language correction (grammar, spelling, fixing sentences, punctuation, suggesting alternative words, etc).

III. RESULTS

A. Initial tests

Our PINN model for these initial tests has two hidden layers with 6 and 7 nodes, and uses a Sigmoid activation layer.

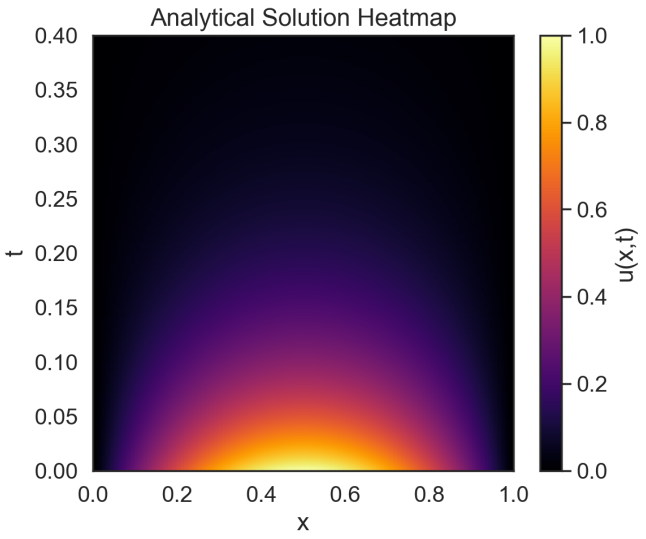


Figure 1: The figure shows how the analytical solution decays exponentially over the domain x [0,1] and t [0,0.4]. There is an initial peaked temperature profile that gradually smooths and decreases in amplitude over time due to diffusion.

Figure 1 shows the behaviour of the analytical solution in the chosen time domain and over the whole spatial domain. The maximum value occurs at x = 0.5 at t = 0. The solution is symmetric around the midpoint, exhibiting a sinusoidal shape that gradually flattens over time. This plot serves as the reference solution to evaluate both the Euler and PINN methods; in addition, it provides our reference temporal domain (reference value for **T_final**).

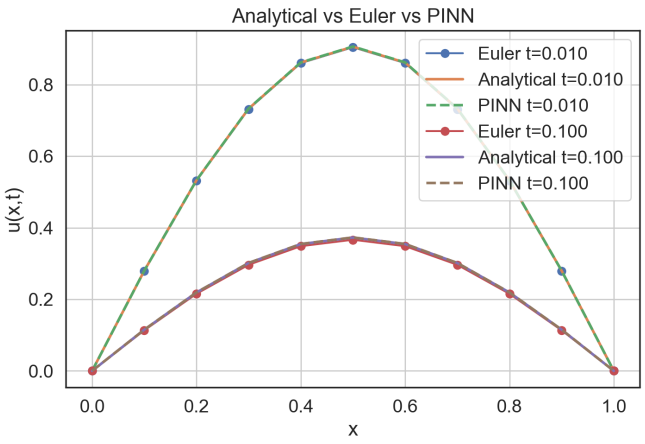


Figure 2: The analytical solution, our Euler scheme, and PINN model tested at two different time points, t1 and t2.

In Figure 2, we present the initial tests of the Euler scheme and PINN model at two time points, t = 0.01 and t = 0.10, plotted against the corresponding analytical solutions at these time points. We have not yet optimized the PINN model and have chosen some arbitrary values. We observe that the Euler FD method lies almost on top of the analytical solution, just as our PINN model does. At t=0.10, a slightly more diffused profile is visible, consistent with the analytical behaviour we showed in Figure 1. From this plot alone, we can see no visible discrepancy.

1. *Euler vs PINN analysis*

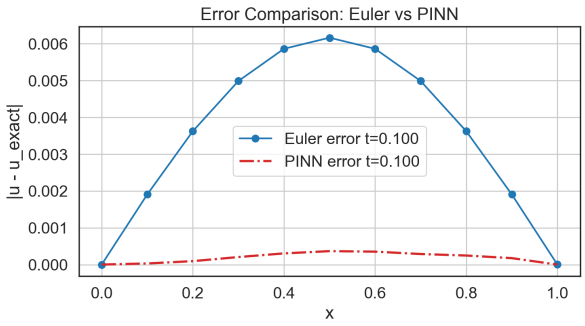
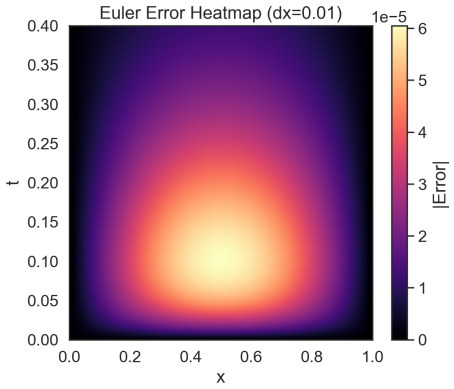


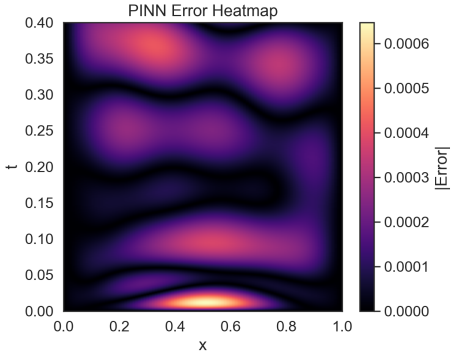
Figure 3: Error (u - u_analytical) for the Euler scheme and PINN method at time = 0.1. The Euler method exhibits larger error throughout the domain, reaching over 0.006, while the PINN method maintains consistently lower error, peaking just above 0.

Figure 3 shows the error at each computed point for t=0.10. The PINN has much lower maximum error and smoother spatial accuracy compared to Euler, demonstrating that PINN provides a more accurate approxi-

mation at this time point. This is consistent with PINN minimizing the residual of the PDE globally [9], while Euler accumulates truncation error. In Figure 4, the error maps show how accuracy evolves over both space and time. The Euler method shows increasing error for larger t, especially around the center x. The PINN maintains low error at small times, but develops a localized band of errors for intermediate t. Overall, PINN maintains a lower maximum error. Figure 5 examines the long-time behaviour. At t = 0.2, both numerical methods accurately track the analytical curve. At t = 0.5, the Euler method begins to underpredict the amplitude, while the PINN slightly overestimates it; however, both methods still capture the parabolic shape and remain close to the analytical solution.

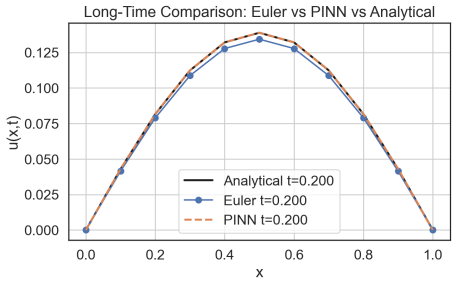


(a) Euler error heatmap

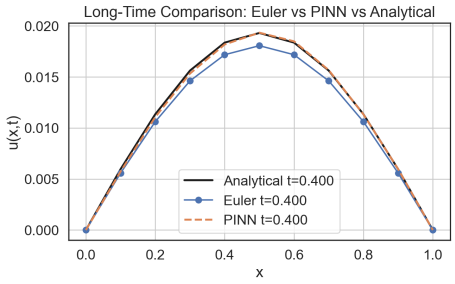


(b) PINN error heatmap

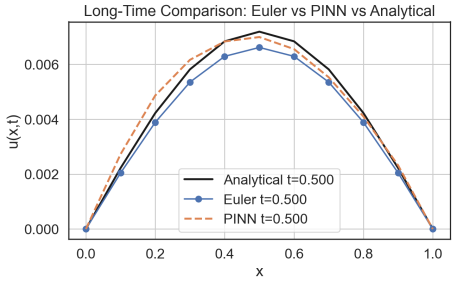
Figure 4: Comparison of Euler and PINN error heatmaps. Euler's error grows with time and remains largest in the spatial center. The PINN shows low error at small t, but error grows near t ≈ 0.03-0.05 before decaying again.



(a) Long time comparison of Euler, PINN and the analytical solution at t=0.2.



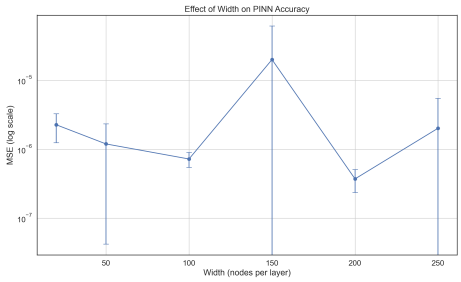
(b) Long time comparison of Euler, PINN, and the analytical solution at t=0.4.



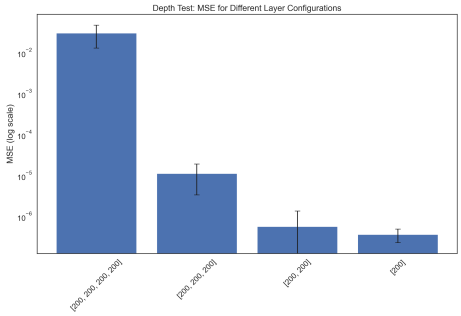
(c) Long time comparison of Euler, PINN, and the analytical solution at t=0.5.

Figure 5: Long-time behaviour of Euler, PINN, and the analytical solution.

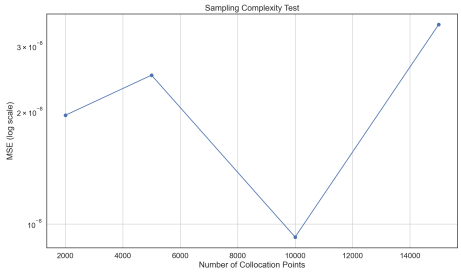
2. *Hyperparameter testing*



(a) Effect of layer width (nodes per layer). Best performance occurs at a width of 100-200 nodes, while overly wide networks displayed variance and instability.

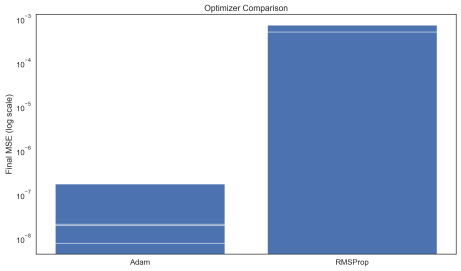


(b) Networks of varying depth. Very deep architectures show poor performance, while moderate depths achieve the lowest error. Note that the best result could be either one or two hidden layers, as the two-layer model had a much better loss value (not shown) while the one-layer model had a better MSE value.

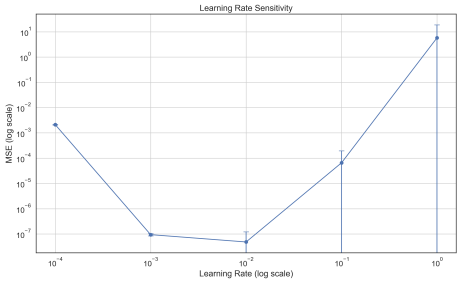


(c) Number of collocation points. Using the best results from the former tests, we observe that performance improves as the number of collocation points increases up to 10,000; beyond that, we encounter mild instability.

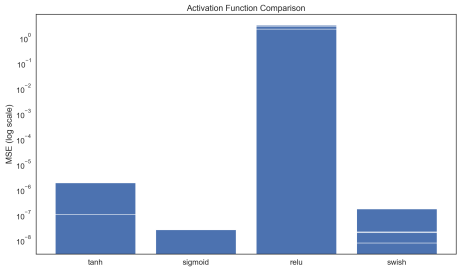
Figure 6: Architecture tests.



(a) Final MSE for ADAM and RMSProp. Adam achieves substantially lower error, indicating better suitability for our PINN model.



(b) MSE vs learning rate. Extremely small or extremely large learning rates degrade performance. Optimal accuracy is achieved around 10^{-2} to 10^{-3}



(c) MSE vs activation function.

Figure 7: Optimizer tests.

The hyperparameter studies show several clear patterns. In Figure 7, Adam significantly outperforms RMSProp in final MSE, and learning rate tests confirm that optimal training occurs at 10^{-2} to 10^{-3}, where smaller and larger learning rates reduce accuracy. For the optimizers, the figure shows that tanh, sigmoid, and swish all achieved comparable MSE values, with swish and tanh as the most consistent ones. ReLU performed extremely poorly, with significant errors. In figure 6, we see that increasing the number of collocation points improves accuracy up to 10,000 samples. Moderate network depth (1-2) layers and width (100-200 nodes) produce the lowest errors, while deeper or excessively wide networks become unstable.

3. *Final model after hyperparameter testing*

The PINN was finally trained with an architecture of [200, 200] layers and nodes, using tanh activation, Adam optimizer, early stopping for epochs, and T_final = 0.5 (which, as shown in Figure 1, corresponds to the point where diffusion stops). With this setup, we achieved an improvement in MSE from a magnitude of 10^{-8} to 10^{-9}.

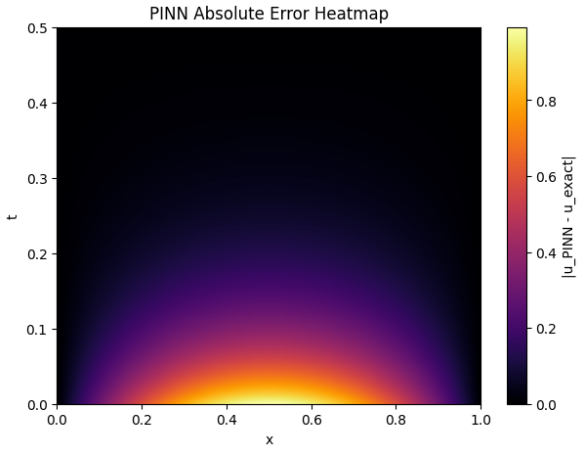


Figure 8: PINN error heatmap - final model. The error is dominated in x=0.5, t=0, consistent with where the analytical solution has its largest amplitude.

In the initial PINN in Figure 4b, we saw wavy, oscillatory bands across time and higher error concentrated in a stripy pattern. These patterns indicate instability and fluctuations in the learned PDE solution. Our final PINNs error heatmap, shown in Figure 8, displays a minimal, smooth error that decays over time, which is consistent with a diffusion behavior. There are no oscillations, and the largest error occurs at the midpoint, where the analytical solution has its maximum. The final model captures the structure of the diffusion equation more accurately, and appears to have learned the correct global behavior.

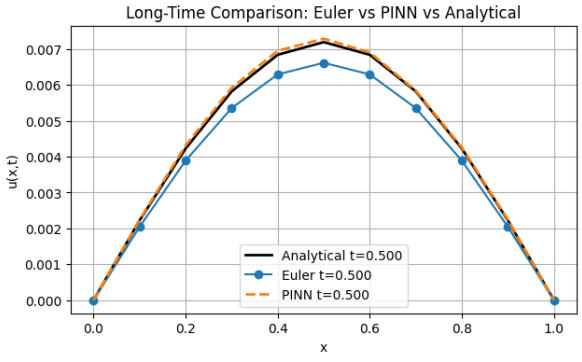


Figure 9: Long-time behavior of our final model at t=0.5.

The initial long-time behavior shown in 5 at t=0.5 showed that our PINN model deviated noticeably from the analytical solution. Our final model in Figure 9 matches the analytical solution almost exactly, and has a much smaller deviation than the Euler model. It also exhibits more accurate behavior near the boundaries, where the initial tests showed an incorrect slope.

IV. DISCUSSION

The initial result comparing the analytical solution, Euler, and PINN demonstrates that both the Euler scheme and PINN model are capable of reproducing the analytical solution with high accuracy for short times and coarse spatial resolution. The explicit Euler scheme performs well because the spatial step size and time step satisfy the stability condition in equation 8. Our relatively small and simple PINN model achieves comparable accuracy to the Euler scheme, and although it doesn't show a clear advantage, it also doesn't fall behind. The PINN model matches the analytical curves at two different times, which suggests it has learned both the temporal and spatial structures of the solution.

Because the diffusion equation is linear and well-behaved, both approaches perform seemingly equally well under these conditions. Any differences are too small to be visible in Figure 2. This motivated the need for additional tests to see where the Euler method might begin to degrade and if the PINN model maintains stability or accuracy in these regimes. Stability in the PINN framework refers to optimization stability rather than numerical stability, as in Euler, as PINNs do not evolve the solution explicitly in time.

From our error analysis, the results highlight a difference between traditional numerical schemes, such as the Euler method, and our PINN model in solving time-dependent PDEs. They both perform well over short times, but their behaviour diverges as time increases. The Euler method underestimated the amplitude of the solution over a longer time, due to its tendency to smooth the solution more than the true PDE does and because of growing error bands, as we could see in the heatmap in Figure 4a. The PINN method demonstrated high accuracy over the time interval for which it was trained on. The error comparison in Figure 2 shows that the network approximates the solution with an error much smaller than Euler, and without the same kind of numerical diffusion. At t = 0.5, it slightly overestimates the peak compared to the analytical solution. This suggests the network has learned the overall shape well, but the long-time behaviour is still sensitive to training choices (which we didn't pay much attention to in these first tests). As this was just an initial test to observe the behavior, we recognize that we need to refine our PINN model and identify the optimal architectures and hyperparameters.

The hyperparameter results guide how to improve our model. Increasing the collocation points improves the model only up to a point, beyond which optimization

becomes harder and variance increases. Similarly, depth and width must be chosen wisely; networks that are too wide or too deep do not improve performance, but rather degrade it. The learning rate tests confirm the expected behavior of Adam, that a moderate learning rate yields good convergence, but more extreme values lead to instability. A surprising result was that the Sigmoid activation function gave quite good results, but it is known to suffer from vanishing gradients in deeper networks, so we did not choose it for our model. Swish also achieved good MSE values, but as tanh achieved comparable results to both swish and sigmoid, we decided to use that. It is a widely used and well-understood function in PINNs, achieving very small errors, only slightly above those of swish and sigmoid.

With our final PINN model, we outperformed the initial model and the Euler scheme. The difference between the initial and final model highlights some discoveries about PINNs:

1. The network must be large enough. The initial network was too small ([6,7]) for the smooth nature of the diffusion equation. Too few nodes can lead to the oscillation observed in Figure 4b, which we helped eliminate by creating a wider and deeper network.

2. Early stopping prevents overfitting the residual. Overfitting could be a reason for the banded, striped pattern we saw in the error heatmap. Early stopping halts training once progress plateaus, and helps runtime stay manageable.

3. Although MSE has not improved drastically, the physical correctness of the problem has improved. The initial PINN didn't have the correct long-time behavior, and we had oscillations in the error heatmap. In the final model, the long-time behavior aligns with the analytical solution and its exponential decay behavior, and the error heatmap displays a smooth error pattern, consistent with diffusion.

While PINNs demonstrated better accuracy in this study than Euler, this comes at a high computational cost. Training a deep neural network with thousands of collocation points and epochs is a lot more expensive than a single Euler solve. However, once the network is trained, we can utilize the solution for any purpose without needing to retrain the model. This is especially useful if we need to repeat evaluations, as the Euler scheme would have to be rerun every time.

A. Limitations and Future Use

The results from this project demonstrate that PINNs can accurately solve the one-dimensional diffusion equation and, in several cases, outperform the explicit Euler method. Some limitations should be addressed, especially when evaluating the future use of this method.

One key limitation is the simplicity of the physical system that was studied. The problem we studied is well understood and highly idealized. Initial and boundary conditions are known, and an analytical solution is available for comparison. The problem is linear and numerically stable. Both methods should be expected to perform well here. It does not show the type of problem where PINNs could be expected to provide a larger advantage than classical numerical solvers. A non-linear PDE with sharp gradients or irregular geometries would pose a greater challenge.

Another challenge could be the limitation of the trial solution, which explicitly enforces the initial and boundary conditions. In problems that aren't as well-defined as the one-dimensional diffusion equation, with possibly more complex or time-dependent boundary conditions, this approach may not be suitable. It can restrict the space of functions that the network can represent. The number and distribution of collocation points can also pose a limitation. Here, the collocation points were sampled uniformly across the spatial and temporal domains, which works well for smooth solutions. If the solution varies rapidly in specific regions or at certain times, the uniform sampling can lead to wasted computational effort in areas where the solution is already well-approximated. In these cases, adaptive sampling, where more points are sampled from areas with steeper gradients, can improve convergence and training time.

Our PINN model was sensitive to hyperparameter choices, and despite extensive hyperparameter testing, some variability was observed between different random seeds. For more complex problems, optimization can be challenging due to stiff and non-convex landscapes. There are methods, such as L-BFGS, that can improve convergence and accuracy in more complex cases, but we did not test them for this problem.

Another factor is the computational cost of PINNs. For this simple problem, the Euler method is inexpensive and straightforward to implement, providing accurate solutions comparable to those of PINN and the analytical solution. The computational cost (and time spent) of training the PINN model might not be justified for problems of this type.

Finally, in this problem, all the physical parameters were known in advance. One of the more promising and interesting applications of PINNs is in problems where the coefficients, source terms, or boundary conditions are derived from limited or noisy data or so-called inverse problems, where some part of the PDE is unknown and must be inferred from measured data. PINN can combine data and physics in a single function and integrate multiple data sources, which could be useful in many real-life settings (technology and engineering companies using machine learning for a better understanding of their product, for instance).

V. CONCLUSION

The initial tests confirm that both PINN and Euler reproduce the analytical solution of the diffusion equation with great accuracy for short times. PINN achieved significantly lower error than the Euler scheme. Error heatmaps show that Euler accumulates error over time, while PINN maintains a uniformly low error across the domain. Further studies over longer times revealed that both methods stay close to the analytical solution. Still, Euler slightly underestimates, and PINN slightly overestimates, primarily due to a lack of correct training. Hyperparameter testing shows that PINN accuracy is highly sensitive to architectural choices and optimizer settings. Optimal performance is achieved using ADAM optimizer, a moderate learning rate (10^{-2} to 10^{-3}), and a network of moderate width and depth. A too-deep network, too many modes, or too many collocation points can cause degraded training stability. For the activation function, although the safe choice was the tanh function, both sigmoid and swish achieved a lower MSE value, suggesting potential for experimentation with different activation functions to improve accuracy.

The final PINN model demonstrates qualitative and quantitative improvements compared to the initial tests. We don't achieve a dramatically lower MSE with our modifications, but our model displays a much more accurate behaviour than earlier and removes previously observed oscillations in the error heatmap. The optimized model:

- Eliminates oscillations seen in earlier error heatmap
- Exhibits correct long-time decay
- Produces accurate solutions for all tested time slices
- Outperforms Euler's method at both short and long times
- Achieves a smoother, more physically meaningful error distribution

Our results demonstrate that architecture, training stability, optimizations, and regularization (like early stopping) are essential for a stable and accurate PINN solution. These findings provide valuable insight into how various factors impact the performance of PINNs, showing potential for solving more complex differential equations with machine learning methods.

The study we have done here demonstrates that PINN can accurately solve the one-dimensional diffusion equation and provides insight into how architectural and training choices affect performance. The results clarify how PINNs behave in general and illustrate their strength and weaknesses. However, the experiments we did here only represent a proof of concept. We demonstrated how PINNs *can* work, but we have not tested all cases, limitations, or real-world scenarios. This is not a comprehensive validation or judgment on the method, as we have yet to fully assess the true capabilities of PINNs. Addressing these limitations would enable a more thorough evaluation of PINNs and their potential role in scientific computing, as well as whether PINNs can complement or replace traditional numerical methods. Overall, the study suggests that PINNs have significant potential but require further validation before determining whether they can be used to solve complex real-world differential problems.

[1] M. Hjorth-Jensen, Week 41: Neural networks and constructing a neural network code, https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week41.html (2025), accessed 2025-11-01.

[2] G. Cybenko, Mathematics of Control, Signals and Systems **2**, 303 (1989).

[3] M. Hjorth-Jensen, Week 43: Deep learning: Constructing a neural network code and solving differential equations, https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week43.html (2025), accessed 2025-11-01.

[4] A. Rasmussen and S. Hannestad, Lecture notes: Machine learning — chapter 10: Regularization and hyperparameters, https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter10.html (2024), accessed: 2025-11-02.

[5] Wikipedia contributors, Numerical diffusion (2025), accessed 2025-12-16.

[6] M. Hjorth-Jensen, Solving differential equations with deep learning — applied data analysis and machine learning lecture notes: Chapter 11, https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter11.html (2025), accessed 2025-12-16.

[7] M. Hjorth-Jensen, Week 42: Constructing a neural network code with examples, https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week42.html (2025), accessed 2025-11-01.

[8] A. Rasmussen and S. Hannestad, Lecture notes: Machine learning — chapter 9: Neural networks, https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter9.html (2024), accessed: 2025-11-02.

[9] M. Hjorth-Jensen, Machine learning: Lecture notes – week 40 (2025), accessed 2025-10-12.