Going with the Flow: A Comparative Study of Physics-Informed Neural Networks and Finite Element Method for Diffusion and Navier-Stokes Equations in Poiseuille Flow

FYS-5429

Written by:

Odin Johansen

Department of Physics UiO

UiO : University of Oslo

June 14, 2024

Contents

1. Abstract

This study compares the Finite Element Method (FEM) and Physics-Informed Neural Networks (PINNs) for solving two-dimensional Navier-Stokes and Diffusion equations. We analyzed the impact of hyperparameters, including network architecture, regularization, training iterations, and activation functions. For the Diffusion equation, the SiLU activation function was most effective, achieving a minimum loss of $1.0 \cdot 10^{-4}$ after 8000 epochs, whereas Leaky ReLU reached an loss of $1.0 \cdot 10^{-6}$ but struggled with geometric accuracy. Sigmoid and Tanh functions exhibited significant discrepancies compared with the analytical solution. For the Navier-Stokes equation, SiLU achieved a minimum loss of $1.0 \cdot 10^{-3}$ outperforming the other actuation functions with Tanh as a close competitor. FEM outperformed PINNs in accuracy and computational efficiency for both equations, with low error rates and short run-times. These findings highlight the critical role of activation functions and hyperparameter tuning in PINNs for precise predictions. While PINNs are promising, further research is required to achieve the performance and efficiency of FEM. Future work should focus on advanced network architectures, adaptive learning strategies, and GPU acceleration to enhance PINNs' applicability and scalability for complex physical systems.

2. Introduction

The advent of Machine Learning (ML), particularly deep learning, has opened new avenues for addressing computational challenges in solving partial differential equations (PDEs). Among these, Physics-Informed Neural Networks (PINNs) stand out for their innovative integration of physical laws into the learning process. PINNs ensure that solutions are not only data-driven but also conform to underlying physical principles. By encoding PDEs directly into the architecture and training process of neural networks, PINNs enable the simultaneous learning of both the solution to the equation and the identification of unknown parameters within the physical model. This approach promises to revolutionize how we solve PDEs by leveraging the approximative power of neural networks while maintaining physical accuracy.

The diffusion equation, fundamental to describing heat distribution, material diffusion, and other phenomena involving temporal and spatial variable changes. Its solutions underpin critical analyses in fields as diverse as thermodynamics, materials science, geophysics, and biological systems modeling. Traditionally, this PDE has been tackled using numerical methods such as finite difference, and spectral methods. While effective, these methods often demand extensive computational resources and face challenges in handling complex boundary conditions, high dimensionality, and nonlinearities.

The Navier-Stokes equations describe the motion of fluid substances and are crucial in the study of fluid dynamics. These equations account for various factors such as velocity, pressure, temperature, and density of the fluid, making them integral in applications ranging from aerospace engineering to meteorology and oceanography. Solving the Navier-Stokes equations accurately is vital for predicting flow patterns and understanding the behavior of fluids under different conditions. However, like the diffusion equation, traditional numerical methods used to solve the Navier-Stokes equations can be computationally intensive and complex, particularly for turbulent flows and intricate geometries.

The Finite Element Method (FEM) is a powerful numerical technique used to find approximate solutions to PDEs, including the diffusion and Navier-Stokes equations. FEM divides a complex problem domain into smaller, simpler parts called finite elements, allowing for the systematic approximation of solutions. This method is highly versatile and can handle complex geometries and boundary conditions. However, it often requires significant computational resources and expertise in mesh generation and refinement. By comparing FEM with PINNs, this study aims to evaluate the strengths and limitations of both methods in solving diffusion and Navier-Stokes equations, particularly in the context of Poiseuille flow, which involves the laminar flow of a viscous fluid in a pipe.

3. Theory

i. Physics-informed neural networks

We begin by introducing physics-informed neural networks and the theory behind them.

1. *Introduction*

Physics-informed neural networks, commonly referred to as PINNs, are feedforward neural networks [10] which incorporate physical laws described as partial differential equations (PDEs) in order to more efficiently train. PINNs are still quite new but have shown great promise for both solving the Backward and Forward-problem for PDEs. The Backward-problem is; given points in our domain to determine the hyperparameters of the PDE which the points stem from. The Forward-problem is to determine what the solution of the PDE is in its domain. This article will focus on solving the Forward-problem, for more on the Backwards-problem see [8], and references therein.

A feedforward neural network $\theta = \{W_i, b_i\}_{i=1}^{k}$ consists of a set of weights W and biases b. An input is transformed into an output using θ, by multiplying it with the first weight and and adding the first bias, after which one commonly applies a non-linear activation function, before sending the output of this into the second layer of weight and bias. Repeating this process, layer by layer, one ends up with an output, and using the back-propagation algorithm [7], it is then possible to tune the weights and biases in the network, to get a more accurate prediction, referred to as training the network.

2. *Cost function*

Writing a PDE in the general form, with the problem and its boundary condition being described as

$$\mathcal{L}(u) = f, \;\; \mathbf{x} \in \Omega, \qquad u(\mathbf{x}) = g \; \forall \, \mathbf{x} \in \partial\Omega. \tag{1}$$

Where $\mathcal{L}$ is a nonlinear operator, $\Omega \subseteq \mathbb{R}^d$ is the domain of the PDE with $\partial\Omega$ being the boundary, $\mathbf{x} \in \mathbb{R}^d$ the input of our PDE, $f, g : \mathbb{R}^d \to \mathbb{R}$ are some functions and $u(\mathbf{x})$ is the true solution. In addition, it is also necessary to specify an initial solution if the equation is time-dependent.

Examples of the nonlinear operator $\mathcal{L}$, is

$$\mathcal{L}(u) = u'', \qquad \mathcal{L}(u) = u^2 + u', \qquad \mathcal{L}(u) = \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}. \tag{2}$$

Having described the problem we want to solve, using our network θ described in section 3 i 1, we can get a prediction u_θ of u where u is the solution of equation (1). This is quite useful, but in order to be able to train our network, we need a way of measuring the error between u_θ and u referred to as the cost function. Discretizing equation (1) we can achieve this, and get the following measures of error for our approximation of u

$$\mathrm{MSE}_u = \frac{1}{N_u} \sum_{i=1}^{N_u} |u_\theta(\mathbf{x}_i^b) - u(\mathbf{x}_i^b)|^2, \qquad \mathrm{MSE}_{\mathcal{L}} = \frac{1}{N_{\mathcal{L}}} \sum_{i=1}^{N_{\mathcal{L}}} |\mathcal{L}(u_\theta)(\mathbf{x}_i^{\mathcal{L}}) - f(\mathbf{x}_i^{\mathcal{L}})|^2. \tag{3}$$

Where $\{\mathbf{x}_i^b\}_{i=1}^{N_u}$ are points sampled from $\partial\Omega$, and $\{\mathbf{x}_i^{\mathcal{L}}\}_{i=1}^{N_{\mathcal{L}}}$ are points sampled from $\Omega \backslash \partial\Omega$. It is important to mention that we are able to compute $\mathcal{L}(u_\theta)$ due to automatic differentiation, see [1]. Combining these two measures of errors we get the following cost function for our PINN

$$\mathrm{MSE} = \lambda_u \mathrm{MSE}_u + \lambda_{\mathcal{L}} \mathrm{MSE}_{\mathcal{L}}. \tag{4}$$

Where $\lambda_u, \lambda_{\mathcal{L}} \in \mathbb{R}_+$, are constants determining how much the error on the boundary versus the error in interior points should affect the training. Having determined the loss function of our network, it is thus possible to train it, using the back-propagation algorithm.

3. Regularisation

When training our network it is often beneficial to add some type of regularisation to our weights. This means our network automatically punishes too large weights, which is beneficial due to it making it hard for one attribute of our input to dominate the prediction. The easiest way to do this is to rewrite our cost function to

$$\text{MSE} = \lambda_u \text{MSE}_u + \lambda_{\mathcal{L}} \text{MSE}_{\mathcal{L}} + \lambda ||W||_2. \tag{5}$$

Where λ determines how much regularisation there is, and $|| \cdot ||_2$ is the regular L_2-norm.

4. Useful properties of PINNs

Having introduced the theory behind PINNs, we deem it also important to mention some of the attributes which make them a promising new alternative for solving PDEs.

- PINNs do not suffer from the curse of dimensionality, where the time complexity grows exponentially when increasing the dimensionality of the PDE. For more on this, see [5] and references therein.
- After training our network, it is possible to evaluate our solution at any point, without having to reprocess our solution using for example interpolation.
- PINNs utilize the formulation of the PDE in addition to its boundary conditions, thus making it quite simple to set up once the main framework for the feedforward neural network is in place.
- It is not required to generate training data beforehand, to train PINNs. Instead, through its use of the PDE and randomly sampling the domain. It can generate it's own training data, by predicting randomly sampled points.

ii. Activation functions

As mentioned previously an important aspect of PINNs, and all feedforward neural networks for that matter, is the choice of activation function. For this paper, we will consider the following activation functions.

1. SiLU

Sigmoid Linear Units, shortened SiLU, is given as

$$h(x) = \frac{x}{1+e^{-x}}, \qquad h'(x) = \frac{e^{-x}(1+e^{-x}+x)}{(1+e^{-x})^2}. \tag{6}$$

The main strength of using SiLU is that it combines the properties of both linear and non-linear activation functions, allowing for smooth and non-monotonic transformations. This smoothness helps mitigate issues like the vanishing gradient problem [4] while providing better performance in some neural network architectures compared to ReLu.

2. Leaky ReLu

Leaky Relu, given as

$$h(x) = \begin{cases} x \text{ for } x > 0 \\ 0.01x \text{ for } x \leq 0 \end{cases}, \quad h'(x) = \begin{cases} 1 \text{ for } x > 0 \\ 0.01 \text{ for } x \leq 0 \end{cases}. \tag{7}$$

It is very similar to ReLu, and as such leaky ReLu exhibits many of the same properties. Although the introduction of a linear component for $x \leq 0$, gives the networks more freedom and it therefore sometimes works better than ReLu.

3. Logistic

The logistic function is the same activation function we used for logistic regression and is defined as

$$h(x) = \frac{1}{1+e^{-x}}, \quad h'(x) = \frac{e^x}{(1+e^x)^2}. \tag{8}$$

First introduced in 1838 by Pierre François Verhulst [3], to model population growth. Its sigmoid shape and smooth well-defined derivative make it a good activation function.

4. Hyperbolic tangent function

The hyperbolic tangent function, shortened tanh, is given as

$$h(x) = \tan^{-1}(x), \quad h'(x) = \frac{1}{1+x^2}. \tag{9}$$

Similar to the logistic function, its sigmoid shape, and smooth well-defined derivative makes it a good activation function.

iii. Finite Element Method (FEM)

The Finite Element Method (FEM) is a computational technique for approximating solutions to partial differential equations (PDEs). It is extensively used in disciplines like physics, engineering, and applied sciences where complex PDEs are common. FEM works by dividing a continuous domain into a finite number of smaller subdomains called finite elements. This division process splits the domain Ω into N_e non-overlapping subdomains $\Omega^{(e)}$, as shown below:

$$\Omega = \bigcup_{e=0}^{N_e-1} \Omega^{(e)}. \tag{10}$$

In FEM, local basis functions, which are non-zero only within certain elements, are used to span a vector space:

$$V_N = \text{span}\{\psi_j\}_{j=0}^{N}, \tag{11}$$

where V_N is the space spanned by N basis functions ψ_j. The approximate solution to the PDE is expressed as:

$$u_N(x) = \sum_{j=0}^{N} \hat{u}_j \psi_j(x), \tag{12}$$

where $\hat{u}_j$ are coefficients determined by solving the following equation:

$$(\mathcal{R}_N, v) = \int_\Omega \mathcal{R}_N v \, d\Omega = 0, \quad \forall v \in V_N, \tag{13}$$

Here, $\mathcal{R}_N = \mathcal{L}(u_N) - f$ is the residual, and $\mathcal{L}(u)$ is an operator defined by the PDE. Solving this equation yields the coefficients $\hat{u}_j$, thus providing the solution to the PDE. In practice, software such as FenicsX [2] is used to perform these numerical computations.

iv. The Diffusion Equation

The diffusion equation, also known as the heat equation, is a partial differential equation that describes the distribution of heat in a given region over time.

The general form of the diffusion equation at location $\mathbf{r}$ is given by:

$$\frac{\partial \phi(\mathbf{r}, t)}{\partial t} = \nabla \cdot [D(\phi, \mathbf{r}) \nabla \phi(\mathbf{r}, t)] \tag{14}$$

where:

- $\phi(\mathbf{r}, t)$ represents the density of the diffusing material at location $\mathbf{r}$ at time t.
- $D(\phi, \mathbf{r})$ is the collective diffusion coefficient for the aforementioned density $\phi(\mathbf{r}, t)$.
- ∇ is the vector differential operator nabla.

For this paper, we make the assumption that the collective diffusion coefficient $D(\phi, \mathbf{r})$ is a constant. This assumption reduces equation (14) into a linear differential equation.

$$\frac{\partial \phi(\mathbf{r}, t)}{\partial t} = D \nabla^2 \phi(\mathbf{r}, t) \tag{15}$$

Thus our equation becomes:

$$\frac{\partial u}{\partial t} = D \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{16}$$

where u = $\phi(\mathbf{x}, \mathbf{y}, t)$

1. Diffusion of a Gaussian function

One of the problems we set out to solve in this paper is how a Gaussian curve diffuses out with Dirichlet boundary conditions. Our initial function for the gaussian hill is:

$$u_0(x, y) = e^{-ax^2 - ay^2}. \tag{17}$$

v. The Navier-Stokes Equations

The Navier-Stokes equations describe the motion of fluid substances such as liquids and gases. In two dimensions, these equations govern the behavior of incompressible fluid flow and are fundamental in fluid dynamics. For our problem we are looking at the incompressible Navier-Stokes equations. In two spatial dimensions it is given by:

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = \nabla\cdot\sigma(\mathbf{u},p) + \mathbf{f}, \tag{18}$$

$$\nabla\cdot\mathbf{u} = 0, \tag{19}$$

where:

- $\mathbf{u} = (u, v)$ represents the velocity field of the fluid.
- p is the pressure field.
- ρ is the fluid density.
- $\mathbf{f}$ is the external force per unit volume.
- $\sigma(\mathbf{u},p) = 2\mu\epsilon(\mathbf{u}) - p\mathbf{I}$ is the stress tensor for a Newtonian fluid.
- $\epsilon(\mathbf{u}) = \frac{1}{2}(\nabla\mathbf{u} + (\nabla\mathbf{u})^T)$ is the strain-rate tensor.
- μ is the dynamic viscosity.

l

vi. Solving the 2D Diffusion Equation with FEM

1. *The PDE Problem*

The time-dependent PDE for the heat equation is given by:

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \qquad \text{in } \Omega\times(0,T], \tag{20}$$

$$u = u_D \qquad \text{on } \partial\Omega\times(0,T], \tag{21}$$

$$u = u_0 \qquad \text{at } t = 0. \tag{22}$$

Here, the temperature distribution u is a function of both space and time, e.g., $u = u(x,y,t)$ in a two-dimensional domain Ω. The source term f and boundary condition u_D can also vary with space and time, whereas the initial condition u_0 depends only on the spatial coordinates.

2. *The Variational Formulation*

To solve time-dependent PDEs using the finite element method, we first approximate the time derivative using a finite difference method, which results in a sequence of stationary problems. Each of these stationary problems is then converted into a variational formulation. We use a superscript n to denote the time level, so u^n represents the temperature at time t_n. The finite difference approximation at time t_{n+1} is:

$$\left(\frac{\partial u}{\partial t}\right)^{n+1} \approx \frac{u^{n+1} - u^n}{\Delta t}, \tag{23}$$

where Δt is the time step size. Substituting this into the PDE gives:

$$\frac{u^{n+1} - u^n}{\Delta t} = \nabla^2 u^{n+1} + f^{n+1}. \tag{24}$$

This results in the time-discrete form of the heat equation, known as the 'backward Euler' or 'implicit Euler' method.

Rearranging the equation to isolate u^{n+1} on the left-hand side yields:

$$u^{n+1} - \Delta t \nabla^2 u^{n+1} = u^n + \Delta t f^{n+1}, \qquad n = 0, 1, 2, \ldots \tag{25}$$

Starting from the initial condition u_0, we can solve for u^0, u^1, u^2, and so on.

To apply the finite element method, we convert the equation into its weak form by multiplying with a test function $v \in \hat{V}$ and integrating by parts. We denote u^{n+1} simply as u and obtain the weak formulation:

$$a(u, v) = L_{n+1}(v), \tag{26}$$

where

$$a(u, v) = \int_\Omega (uv + \Delta t \nabla u \cdot \nabla v) \, \mathrm{d}x, \tag{27}$$

$$L_{n+1}(v) = \int_\Omega \left(u^n + \Delta t f^{n+1}\right) v \, \mathrm{d}x. \tag{28}$$

Alternatively, the variational formulation of the time-dependent problem can be derived by multiplying the diffusion equation by a test function v and integrating over the spatial domain:

$$\int_\Omega \frac{\partial u}{\partial t} v \, d\Omega = \int_\Omega D \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) v \, d\Omega + \int_\Omega f v \, d\Omega. \tag{29}$$

Using integration by parts and assuming appropriate boundary conditions, the weak form can be expressed as:

$$\int_\Omega \frac{\partial u}{\partial t} v \, d\Omega + D \int_\Omega \nabla u \cdot \nabla v \, d\Omega = \int_\Omega f v \, d\Omega. \tag{30}$$

Discretizing in time using a backward Euler method, we get:

$$\frac{u^{n+1} - u^n}{\Delta t} v + D \nabla u^{n+1} \cdot \nabla v = f^{n+1} v. \tag{31}$$

Rearranging terms, the discrete variational form at each time step $n+1$ is:

$$\int_\Omega u^{n+1} v \, d\Omega + \Delta t D \int_\Omega \nabla u^{n+1} \cdot \nabla v \, d\Omega = \int_\Omega (u^n + \Delta t f^{n+1}) v \, d\Omega. \tag{32}$$

3. Initial Condition Projection or Interpolation

To start the time-stepping process, the initial condition $u_0(x, y)$ needs to be approximated in the finite element space. This can be done either through projection or interpolation. The projection of the initial condition is given by solving:

$$\int_\Omega u_0 v \, d\Omega = \int_\Omega u_{\text{init}} v \, d\Omega, \tag{33}$$

where u_{init} is the initial temperature distribution function.

Alternatively, interpolation can be used to directly assign initial values to the finite element nodes based on u_{init}.

vii. Solving the Navier-Stokes Equations for Poiseuille flow with FEM

1. The PDE Problem

The incompressible Navier-Stokes equations are given by:

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u}\right) = -\nabla p + \mu\nabla^2\mathbf{u} + \mathbf{f} \qquad \text{in } \Omega\times(0,T], \tag{34}$$

$$\nabla\cdot\mathbf{u} = 0 \qquad \text{in } \Omega\times(0,T], \tag{35}$$

$$\mathbf{u} = \mathbf{u}_D \qquad \text{on } \partial\Omega\times(0,T], \tag{36}$$

$$\mathbf{u} = \mathbf{u}_0 \qquad \text{at } t = 0. \tag{37}$$

Here, $\mathbf{u}$ is the velocity field, p is the pressure, ρ is the fluid density, μ is the dynamic viscosity, and $\mathbf{f}$ represents external forces.

2. The Variational Formulation

To solve the Navier-Stokes equations using the finite element method, we approximate the time derivative using a finite difference method, resulting in a sequence of stationary problems. We use a superscript n to denote the time level, so $\mathbf{u}^n$ represents the velocity at time t_n. The finite difference approximation at time t_{n+1} is:

$$\left(\frac{\partial \mathbf{u}}{\partial t}\right)^{n+1} \approx \frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t}, \tag{38}$$

where Δt is the time step size. Substituting this into the Navier-Stokes equations gives:

$$\rho\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} + \rho(\mathbf{u}^n\cdot\nabla)\mathbf{u}^{n+1} = -\nabla p^{n+1} + \mu\nabla^2\mathbf{u}^{n+1} + \mathbf{f}^{n+1}, \tag{39}$$

$$\nabla\cdot\mathbf{u}^{n+1} = 0. \tag{40}$$

Rearranging the momentum equation to isolate $\mathbf{u}^{n+1}$ on the left-hand side yields:

$$\mathbf{u}^{n+1} - \Delta t\left(\frac{\mu}{\rho}\nabla^2\mathbf{u}^{n+1} - (\mathbf{u}^n\cdot\nabla)\mathbf{u}^{n+1}\right) = \mathbf{u}^n + \Delta t\frac{\mathbf{f}^{n+1} - \nabla p^{n+1}}{\rho}, \qquad n = 0,1,2,\ldots \tag{41}$$

To apply the finite element method, we convert the equations into their weak forms by multiplying with test functions $\mathbf{v}\in\hat{\mathbf{V}}$ and $q\in\hat{Q}$, and integrating by parts. We denote $\mathbf{u}^{n+1}$ simply as $\mathbf{u}$ and obtain the weak formulation:

$$a(\mathbf{u},\mathbf{v}) + b(p,\mathbf{v}) = L_{n+1}(\mathbf{v}), \tag{42}$$

$$b(\mathbf{u},q) = 0, \tag{43}$$

where

$$a(\mathbf{u},\mathbf{v}) = \int_\Omega\left(\mathbf{u}\cdot\mathbf{v} + \Delta t\frac{\mu}{\rho}\nabla\mathbf{u} : \nabla\mathbf{v}\right)\,\mathrm{d}x - \Delta t\int_\Omega(\mathbf{u}^n\cdot\nabla)\mathbf{u}\cdot\mathbf{v}\,\mathrm{d}x, \tag{44}$$

$$b(p,\mathbf{v}) = -\int_\Omega p\nabla\cdot\mathbf{v}\,\mathrm{d}x, \tag{45}$$

$$L_{n+1}(\mathbf{v}) = \int_\Omega\mathbf{u}^n\cdot\mathbf{v}\,\mathrm{d}x + \Delta t\int_\Omega\frac{\mathbf{f}^{n+1}}{\rho}\cdot\mathbf{v}\,\mathrm{d}x. \tag{46}$$

3. *Initial Condition Projection or Interpolation*

To start the time-stepping process, the initial condition $\mathbf{u}_0(x, y)$ needs to be approximated in the finite element space. This can be done either through projection or interpolation. The projection of the initial condition is given by solving:

$$\int_\Omega \mathbf{u}_0 \cdot \mathbf{v} \, d\Omega = \int_\Omega \mathbf{u}_{\text{init}} \cdot \mathbf{v} \, d\Omega, \tag{47}$$

where $\mathbf{u}_{\text{init}}$ is the initial velocity distribution function.

Alternatively, interpolation can be used to directly assign initial values to the finite element nodes based on $\mathbf{u}_{\text{init}}$.

4. Method

i. Error measure

1. *Mean Squared Error*

Mean squared error (MSE) describes the mean distance between prediction and data, with respect to mean square distance. And it is defined as

$$\text{MSE}(\mathbf{z}, \tilde{\mathbf{z}}) = \frac{1}{n} \sum_{i=0}^{n-1} (z_i - \tilde{z}_i)^2.$$

Where $\mathbf{z} \in \mathbb{R}^n$ is the true value, and $\tilde{\mathbf{z}} \in \mathbb{R}^n$ is the predicted value.

ii. Implementation

Both The FEM and PINN methods were implemented in Python, and the source code can be found at https://github.com/Odin107/FYS5429. To train our PINNs we utilized the ADAM optimizer, see [6].

1. *Hyperparameters for PINN*

For the various hyperparameters for our PINN, we sat

$$\lambda_u = 10, \qquad \lambda_{\mathcal{L}} = 1. \tag{48}$$

As we know the solution at the boundary, we put more emphasis on the values we predict there when training. In addition we set the training rate to be 10^{-3} for all simulations, as this is what is recommended using the ADAM optimizer, see [6]. Finally, when training we always sample $2 \cdot 10^3$ points from the boundary and just under $6.0 \cdot 10^3$ interior points.

2. *Point sampling*

In order to train our PINN, we need to sample points from both the boundary and the interior of the domain. These points are sampled uniformly, and during each training iteration, all points are used to update the weights and biases of the network. To test our models, we will evaluate their predictions on points within the domains $[0, 1]^2$ and $[-2, 2]^2$. The points we chose were randomly sampled, and their distribution can be observed in figure (1).

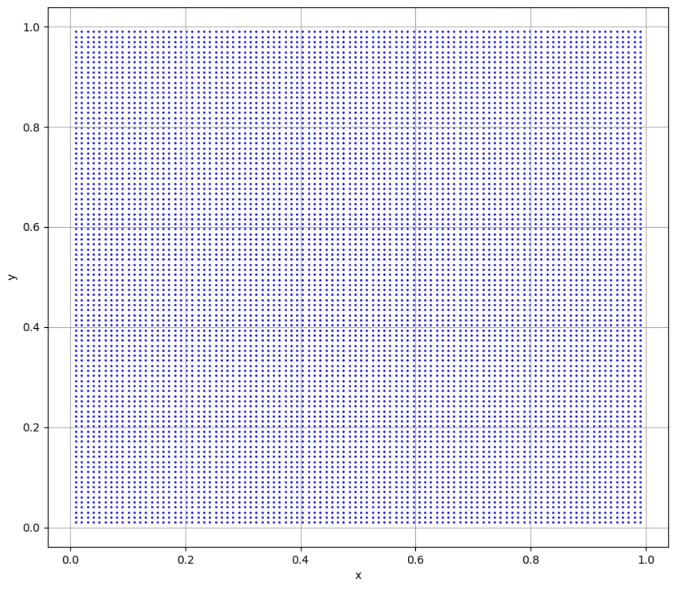

Figure 1. Plot showing the distribution of the 10^4 points used to measure MSE. Each blue dot represents one point.

5. Results and Discussion

i. Physics-Informed Neural Networks (PINNs)

1. *Different Activation Functions and Regularization*

To begin our investigation of PINNs for solving the Diffusion and Navier-Stokes equation, we start by investigating how the activation function affects the MSE. It is important to mention, that from now on we refer to the number of training iterations as epochs.

a. Activation Functions For Solving the Diffusion and Navier-Stokes equations, additional parameters for the simulations are summarized in Tables (I) and (II).

PDE:	**Diffusion**
Parameter	**Value**
Activation function	SiLU, Leaky ReLU, Sigmoid and Tanh
Epochs	15000
Model	3 input, 5 layers, 32 neurons per layer
Optimizer	Adam with learning rate 10^{-3} and weight decay 10^{-4}
Scheduler	Exponential with $\gamma = 0.96$

Table I. Model parameters for the diffusion (PDE) used in this study. Listing the key parameters including activation functions, training epochs, model architecture, optimizer settings, and scheduler details, and L2 regularization constant $\lambda = 10^{-4}$.

2. *Initial Condition for the Diffusion Model*

To begin with, we compare the initial conditions for the Diffusion equation with the analytical solution at time $t = 0$. From figure (2) we observe the true initial condition alongside the predicted initial condition using the SiLU activation function. The model provides a highly accurate prediction with minor discrepancies. These discrepancies likely arise due to the relatively short training period of 15,000 epochs. This limited training time might not be sufficient for the Physics-Informed Neural Networks (PINNs) to fully converge.

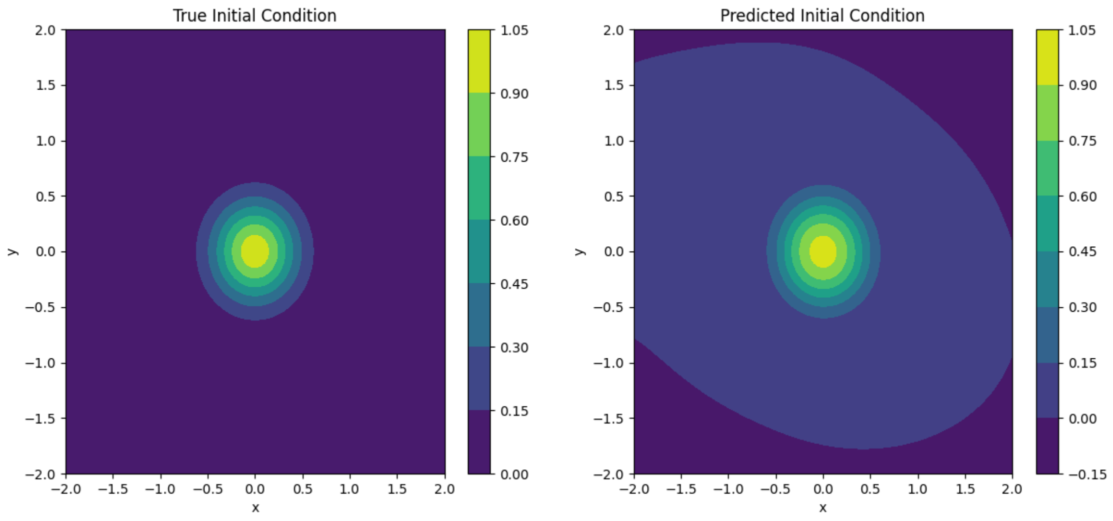

Figure 2. Comparison of the true initial condition (left) and the predicted initial condition (right) using the SiLU activation function on a network with 5 layers with 32 nodes in each, learning rate 10^{-3} and $\lambda = 10^{-4}$. The true initial condition shows a more localized distribution, while the predicted initial condition is more diffused.

We observe figure (3), as it illustrates the importance of choosing the correct activation function for the given system one is solving for, It also highlights the instability of PINNs, if we compare the results from Leaky ReLU with SiLU we see how the values are correct but the gaussian hill has been warped. The true initial condition is highly localized, whereas the predicted condition shows diffusion along a diagonal axis. This behavior could be attributed to the inherent properties of the Leaky ReLU function, which might not capture sharp gradient changes as effectively. The most likely answer is that the hyperparameters are better suited to SiLU than Leaky ReLU as they are the same for all the results, where the only thing changed is the activation function.

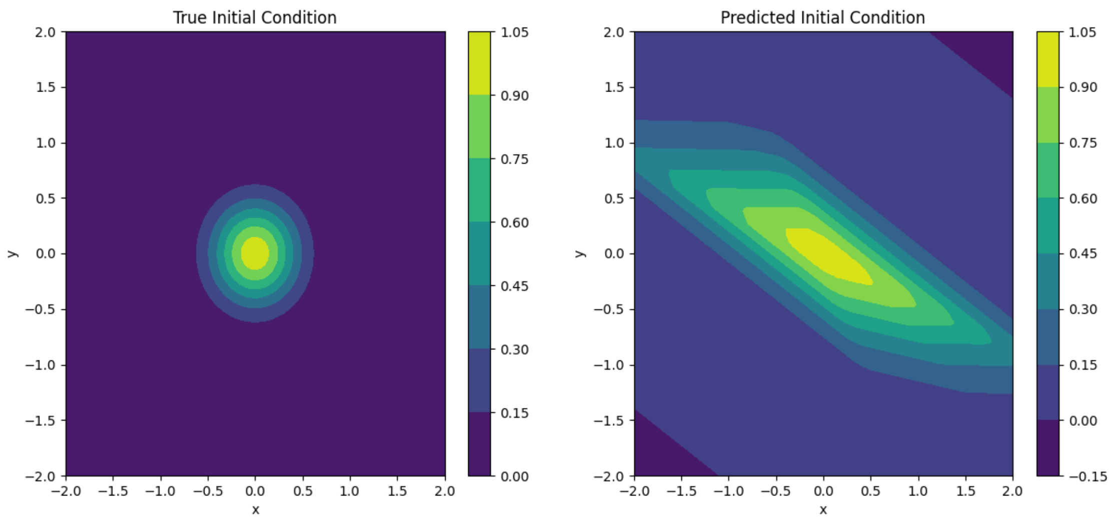

Figure 3. Comparison of the true initial condition (left) and the predicted initial condition (right) using the Leaky ReLU activation function on a network with 5 layers with 32 nodes in each, learning rate 10^{-3} and $\lambda = 10^{-4}$. The true initial condition shows a more localized distribution, while the predicted initial condition is more diffused along a diagonal axis.

We move on to figure (4), where we observe the Sigmoid activation function portraying the largest discrepancy out of all the activation functions, this is not unexpected as the Sigmoid function wants to give values close to either 0 or 1, and this results in a less precise gradient modeling giving us the uniform value across the entire domain.

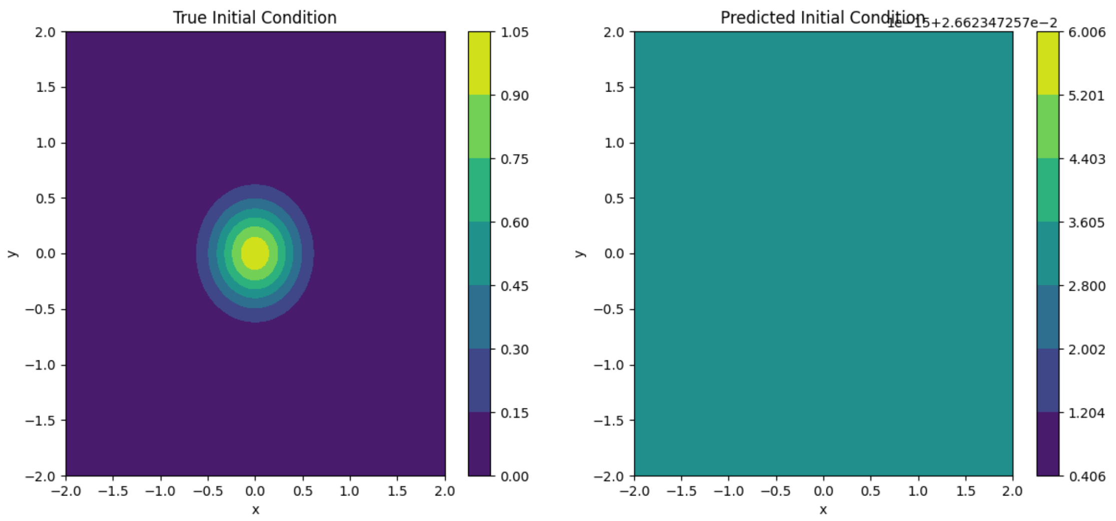

Figure 4. Comparison of the true initial condition (left) and the predicted initial condition (right) using the Sigmoid activation function on a network with 5 layers with 32 nodes in each, learning rate 10^{-3} and $\lambda = 10^{-4}$. The true initial condition shows a more localized distribution, while the predicted initial condition appears to have a uniform high value.

Finally, we see how the model predicts the initial conditions with the Tanh activation function. The prediction, is similar to that of the Leaky ReLU function, as it shows diffusion along a diagonal axis. Although the Tanh function can capture more complex relationships than the Sigmoid function, it still has limitations in accurately predicting the localized initial condition as observed from figure (5).

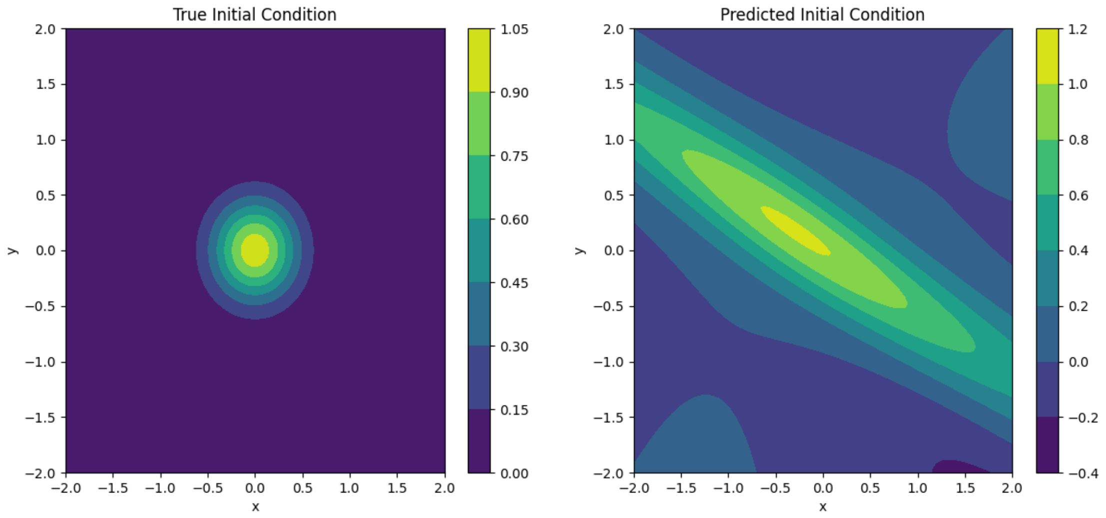

Figure 5. Comparison of the true initial condition (left) and the predicted initial condition (right) using the Tanh activation function on a network with 5 layers with 32 nodes in each, learning rate 10^{-3} and $\lambda = 10^{-4}$. The true initial condition shows a more localized distribution, while the predicted initial condition is more diffused along a diagonal axis.

The accuracy of the predicted initial conditions varies with the choice of activation function. The SiLU function yields the most accurate results with slight amplitude discrepancies, likely due to limited training epochs. Both the Leaky ReLU and Tanh functions show similar patterns of diffusion along the diagonal axis, while the Sigmoid function results in an overly uniform prediction. These observations highlight the importance of selecting appropriate activation functions, ensuring sufficient training time and correctly tuning the hyper-parameters for achieving accurate PINN predictions.

The loss of the Diffusion Model for Different Activation Functions

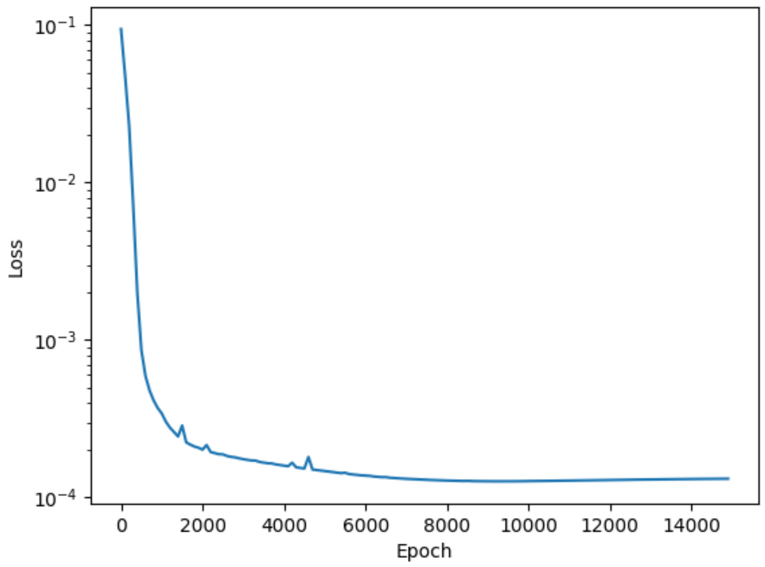

Figure 6. Training loss over epochs using the SiLU activation function on a network with 5 layers with 32 nodes in each, a learning rate of 10^{-3}, and $\lambda = 10^{-4}$. The loss decreases significantly in the initial epochs and gradually plateaus, indicating convergence.

Observing Figure (6) where we use the SiLU activation function, we note that the loss converges after approximately 8000 epochs, with a loss just above 10^{-4}. This result indicates a good prediction of the system, as seen in Figures (10), (11), and (12).

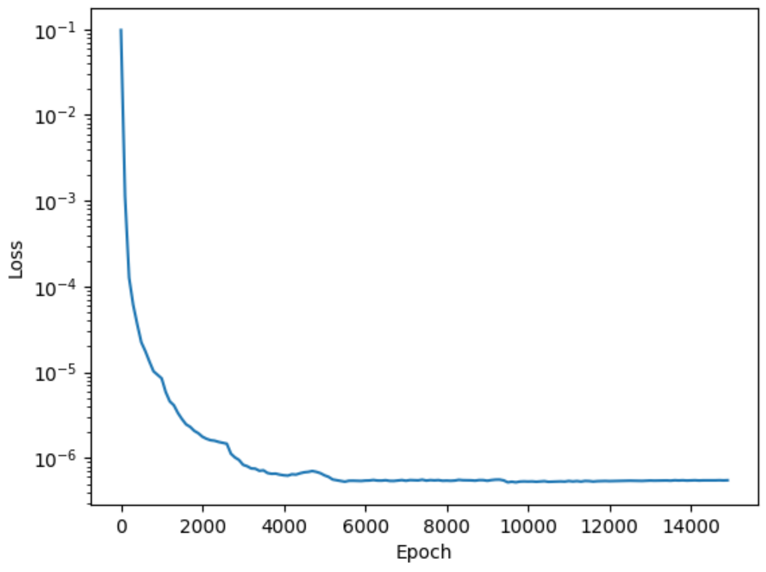

Figure 7. Training loss over epochs using the Leaky ReLU activation function on a network with 5 layers with 32 nodes in each, a learning rate of 10^{-3}, and $\lambda = 10^{-4}$. The loss decreases significantly in the initial epochs and gradually plateaus, indicating convergence.

For Leaky ReLU, the model manages to achieve the lowest loss of just below 10^{-6} after about 6000 epochs, but plateaus after 10000 epochs. The Leaky ReLU activation function shows a significant initial reduction in loss, demonstrating the model's capacity to quickly learn the system's dynamics. The gradual plateau observed indicates that the model reaches a stable state, maintaining a very low loss value. This performance highlights the effectiveness of the Leaky ReLU function in capturing the nuances of the diffusion equation.

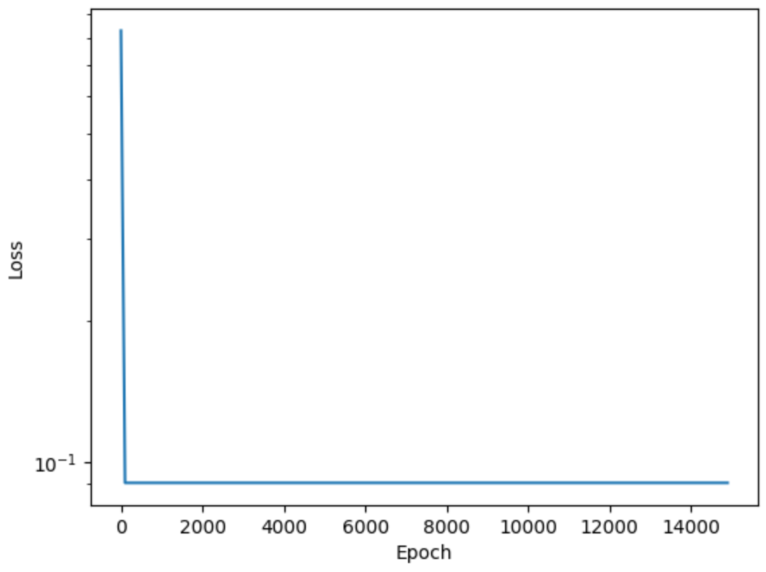

Figure 8. Training loss over epochs using the Sigmoid activation function on a network with 5 layers with 32 nodes in each, a learning rate of 10^{-3}, and $\lambda = 10^{-4}$. The loss quickly drops and then remains nearly constant, indicating early convergence.

The Sigmoid activation function, as shown in Figure (8), demonstrates that the loss quickly drops and remains nearly constant, indicating early convergence. This early plateau suggests that the model may reach a suboptimal minimum faster, which limits its performance compared to other activation functions.

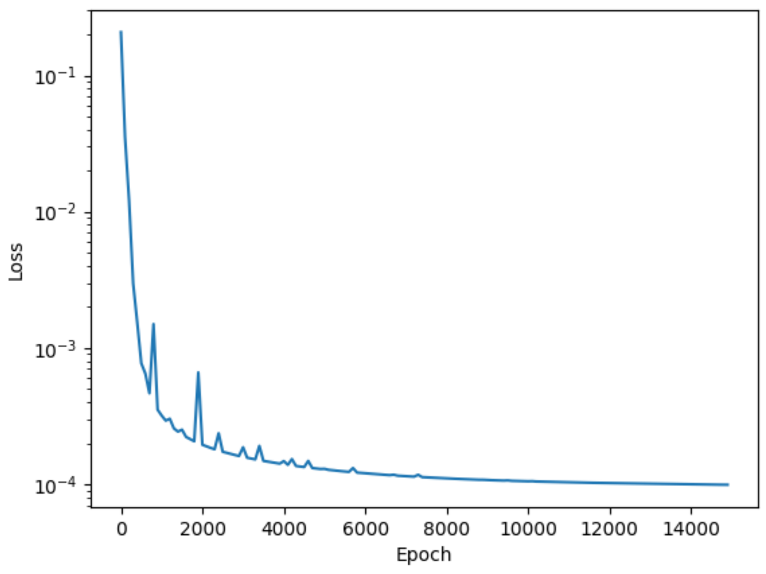

Figure 9. Training loss over epochs using the Tanh activation function on a network with 5 layers with 32 nodes in each, a learning rate of 10^{-3}, and $\lambda = 10^{-4}$. The loss decreases significantly in the initial epochs and gradually plateaus, indicating convergence with occasional spikes.

At last, the Tanh activation function, illustrated in Figure (9), shows a significant initial loss reduction followed by a plateau with occasional spikes. These spikes indicate sensitivity to certain training dynamics, potentially due to the nature of the Tanh function's gradients. Despite these fluctuations, the model still converges effectively, proving the robustness of PINNs in learning complex physical phenomena.

Figures (6), (7), (8) and (9) displayed in the training loss graphs for different activation functions provide significant insights into the performance of Physics-Informed Neural Networks (PINNs) in solving the diffusion equation. The SiLU activation function, as shown in Figure (6), demonstrates a rapid decrease in loss initially and converges after approximately 8000 epochs with a loss just above 10^{-4}. This indicates that the model efficiently learns the underlying physics of the system, as corroborated by the satisfactory predictions observed in Figures (10), (11), and (12). Similarly, the Leaky ReLU function achieves the lowest loss, with a significant initial loss reduction that gradually plateaus, reflecting the model's effective convergence and accurate prediction capabilities.

The Sigmoid and Tanh activation functions also exhibit distinct training behaviors. Figure (8) shows that the loss for the Sigmoid function quickly drops and remains nearly constant, indicating early convergence. This early plateau suggests that the model may reach a suboptimal minimum faster, which might limit its performance compared to other activation functions. On the other hand, the Tanh activation function, illustrated in Figure (9), shows a significant initial loss reduction followed by a plateau with occasional spikes. These spikes indicate sensitivity to certain training dynamics, potentially due to the nature of the Tanh function's gradients. Despite these fluctuations, the model still converges effectively, proving the robustness of PINNs in learning complex physical phenomena.

The Diffusion models prediction

Observing Figures 10, 11, and 12, it is evident that the SiLU activation function is well-suited for solving the diffusion equation. The RMSE development over time indicates that the model had an easier time predicting the initial condition than the boundary condition and the diffusion itself. Nevertheless, this network architecture captures the physical phenomenon we are examining with acceptable accuracy, with a slight discrepancy in the magnitude of the amplitude.

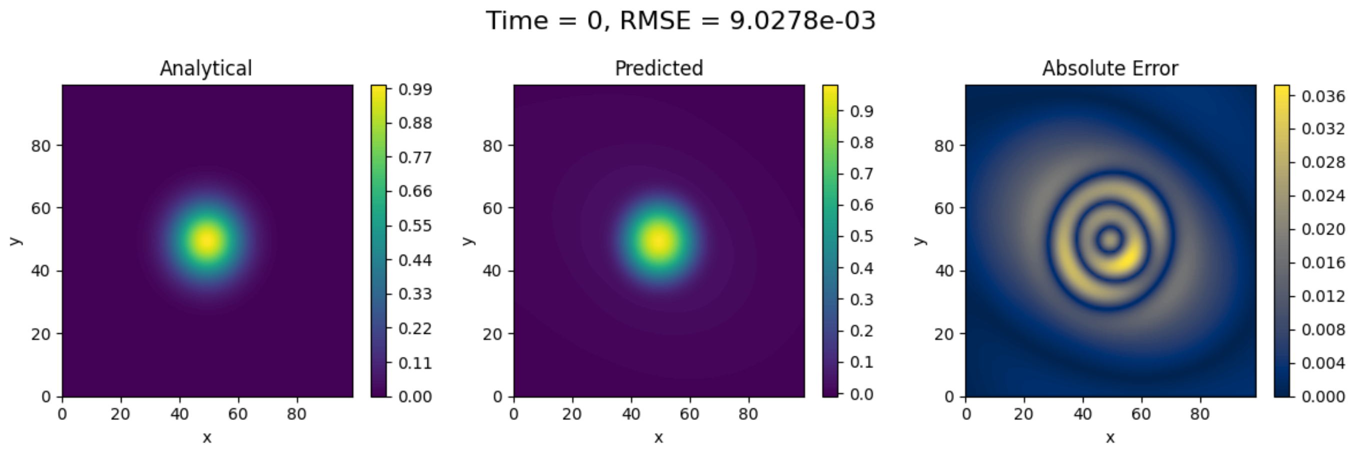

Figure 10. Comparison of analytical and predicted solutions at $t = 0$ using the SiLU activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $9.0278 \cdot 10^{-3}$.

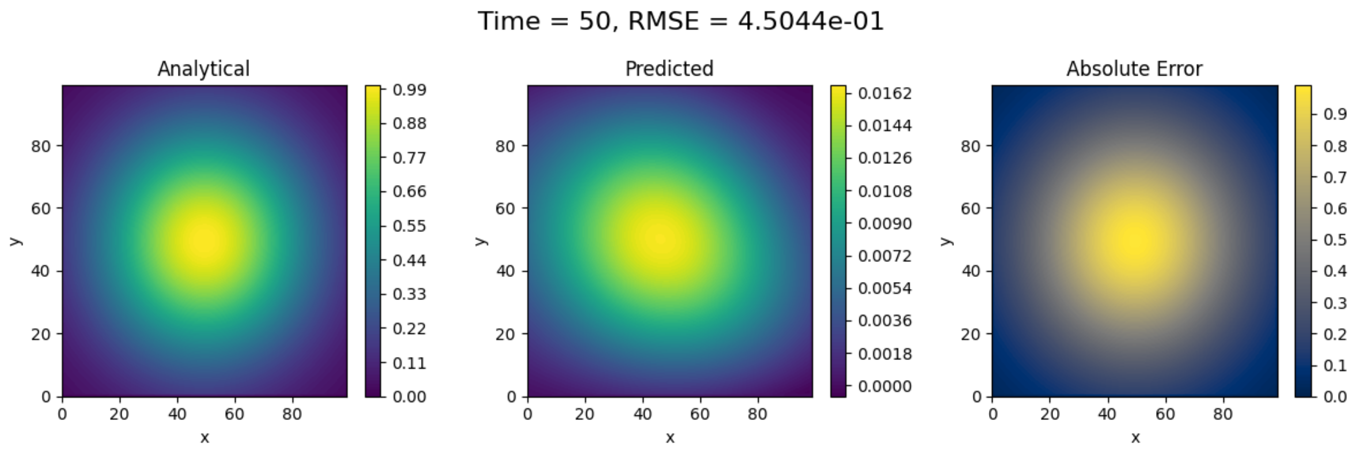

Figure 11. Comparison of analytical and predicted solutions at $t = 50$ using the SiLU activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $4.5044 \cdot 10^{-1}$.

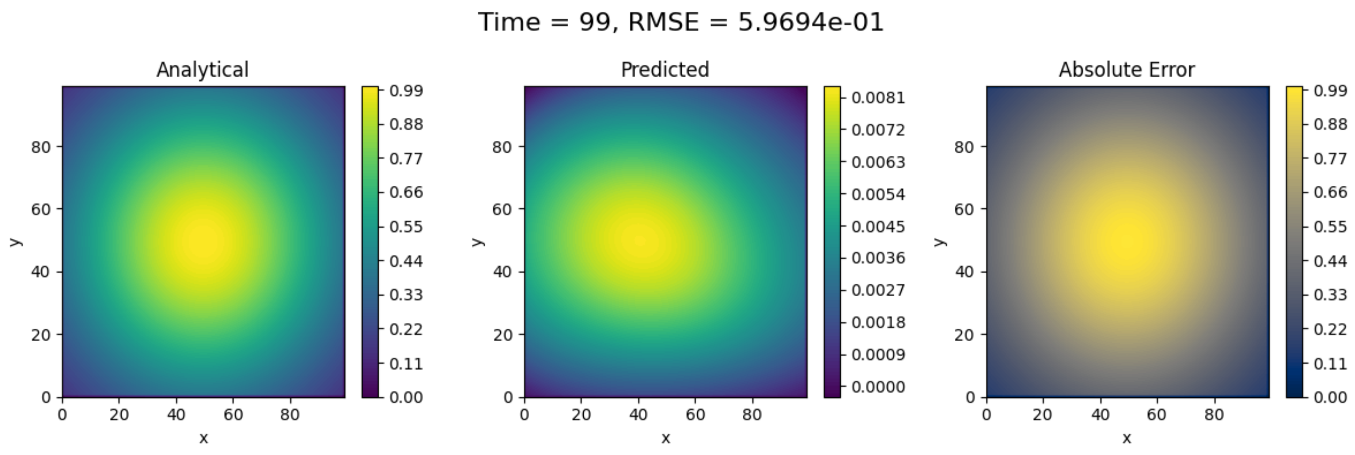

Figure 12. Comparison of analytical and predicted solutions at $t = 99$ using the SiLU activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $5.9694 \cdot 10^{-1}$.

As the initial conditions predicted, the Leaky ReLu activation function has transformed the Gaussian hill into a parallelogram along the diagonal, and it does not diffuse, as shown in Figures (13), (14), (15). Diagnosing this issue is challenging, as it is likely due to the hyperparameters being out of tune, incorrect weight initialization, an improper learning rate, an incorrect L2 regularization constant or all of the above.

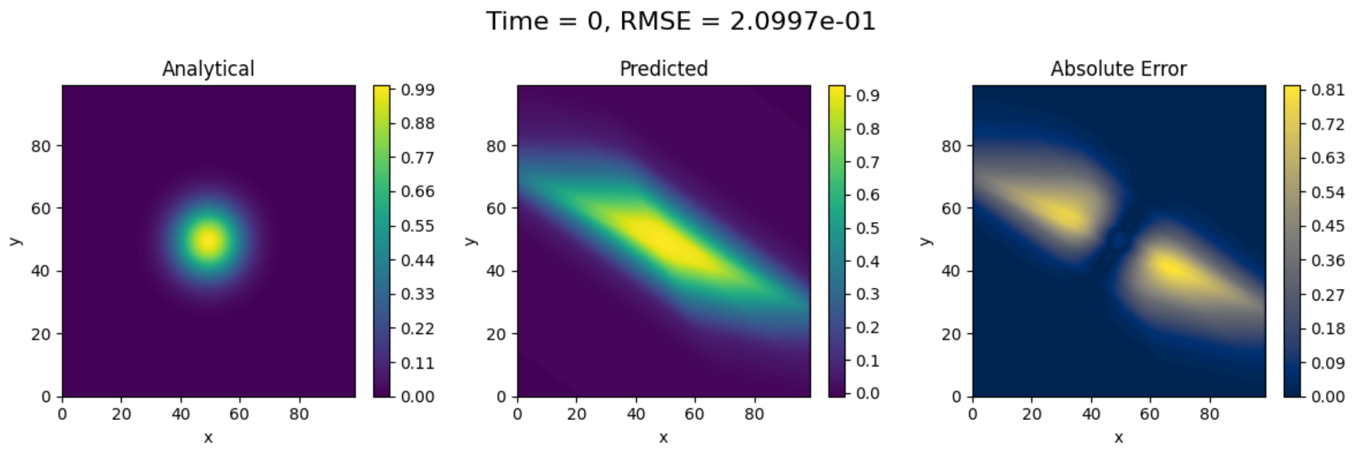

Figure 13. Comparison of analytical and predicted solutions at $t = 0$ using the Leaky ReLU activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $2.0997 \cdot 10^{-1}$.

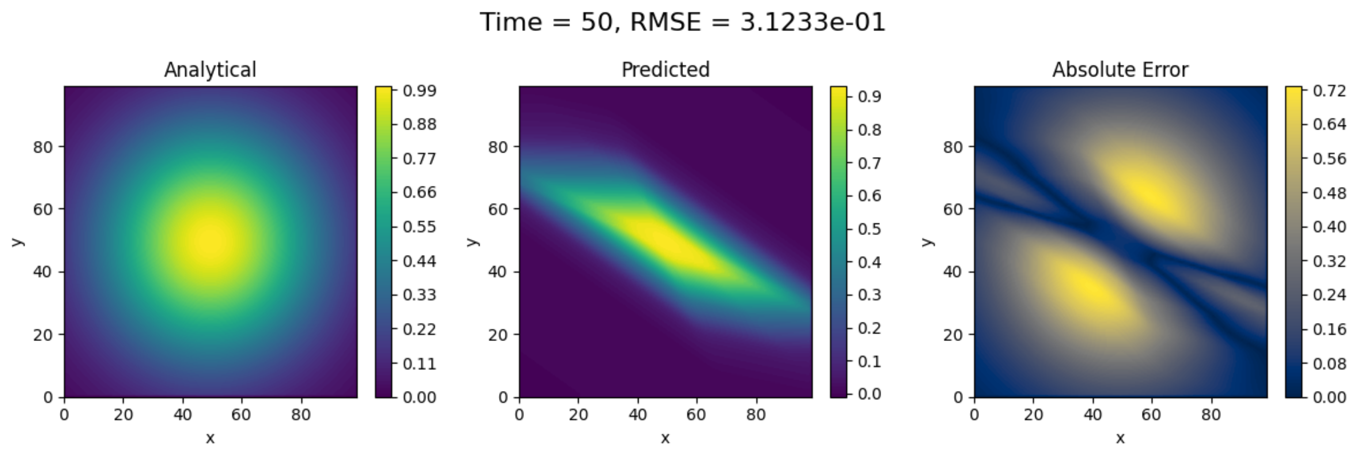

Figure 14. Comparison of analytical and predicted solutions at $t = 50$ using the Leaky ReLU activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $3.1233 \cdot 10^{-1}$.

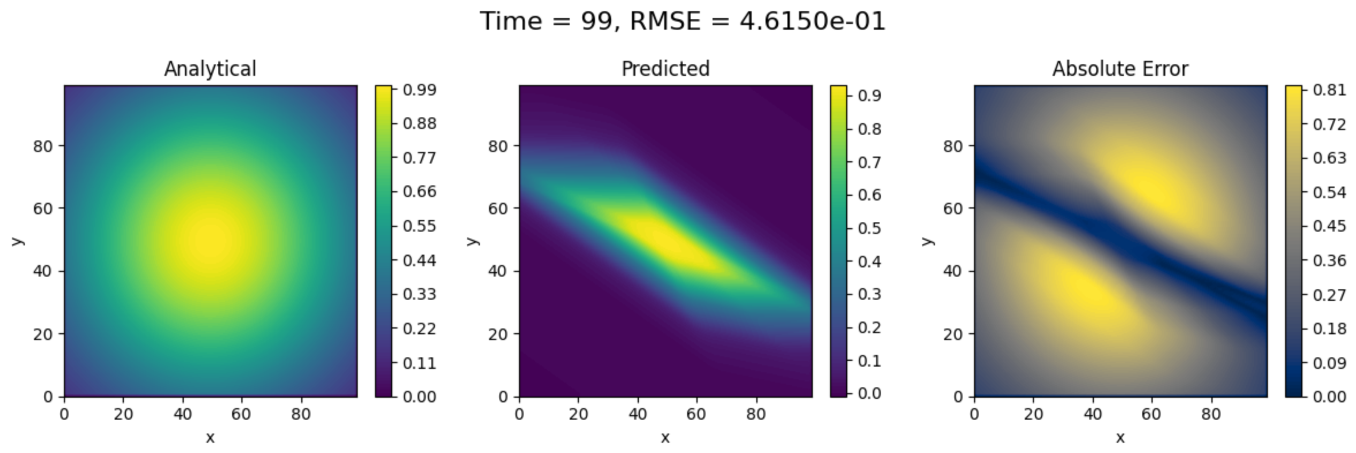

Figure 15. Comparison of analytical and predicted solutions at $t = 99$ using the Leaky ReLU activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $4.6150 \cdot 10^{-1}$.

It is not surprising that the sigmoid function was unable to capture the complexity of the diffusion equation, given its reliance on smooth gradient changes, which is the sigmoid function's primary weakness, as discussed in Section (5 i 2). This weakness results in the predictions shown in Figures (16), (17), and (18), where the boundary conditions are predicted accurately, but the entire domain remains homogeneous. Here, we observe that the absolute error aligns with the analytical values, as expected.

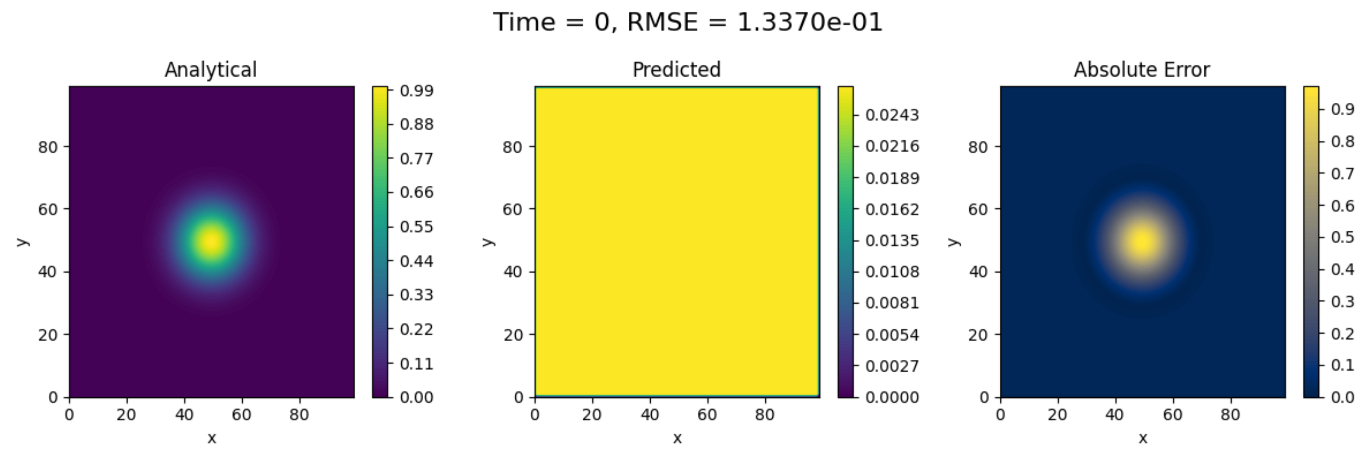

Figure 16. Comparison of analytical and predicted solutions at $t = 0$ using the Sigmoid activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $1.3370 \cdot 10^{-1}$.

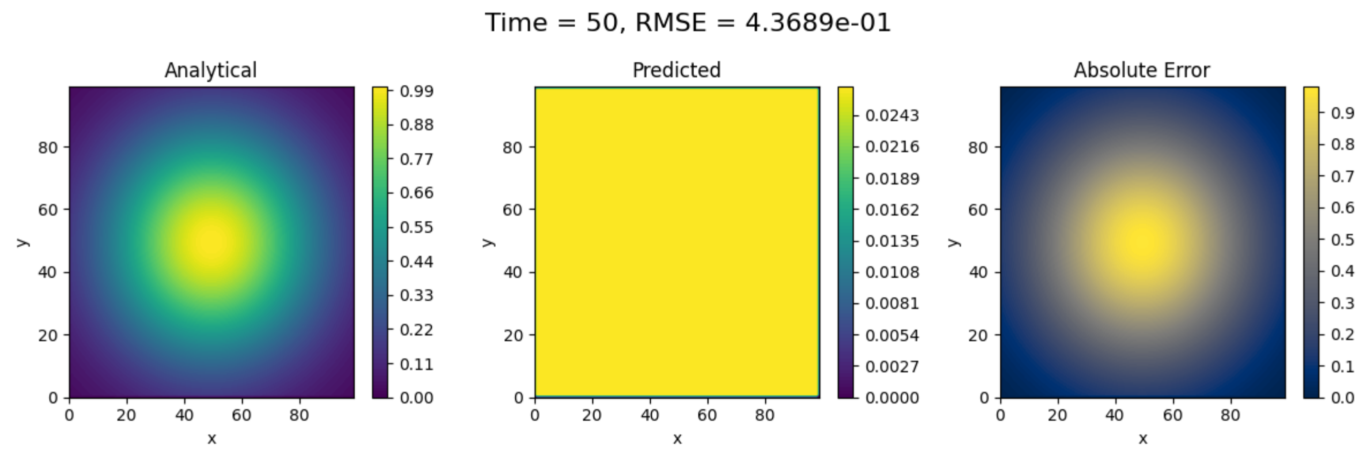

Figure 17. Comparison of analytical and predicted solutions at $t = 50$ using the Sigmoid activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $4.3689 \cdot 10^{-1}$.

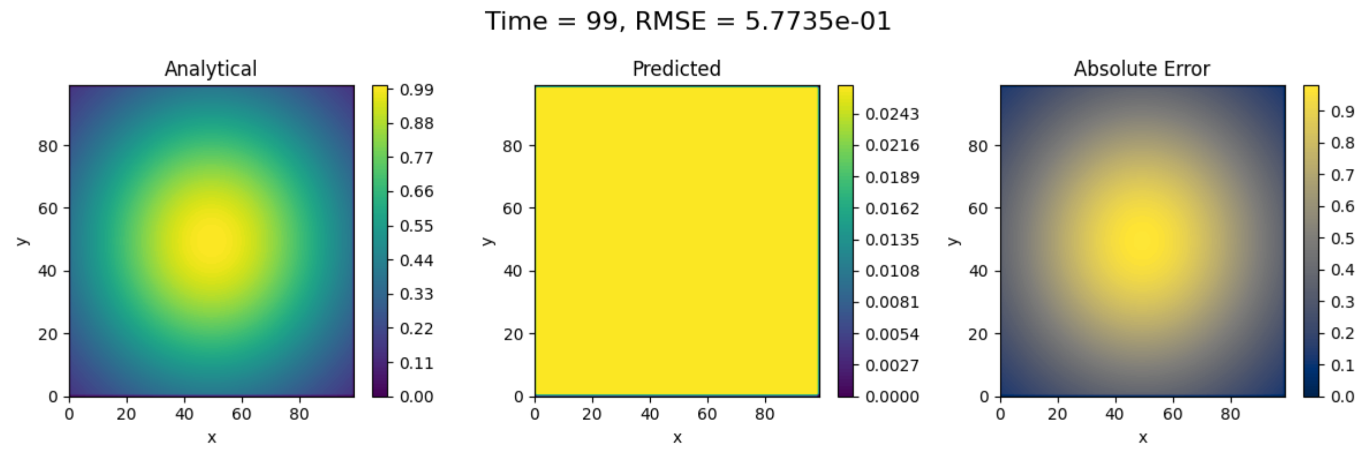

Figure 18. Comparison of analytical and predicted solutions at $t = 99$ using the Sigmoid activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $5.7735 \cdot 10^{-1}$.

Finally, as predicted by Figure (5), we observe that the Tanh activation function warps the Gaussian hill along the diagonal of the domain. Despite this warping, Tanh demonstrates high accuracy in modeling the physical phenomenon, capturing the diffusion of the warped Gaussian hill better than expected from the initial condition.

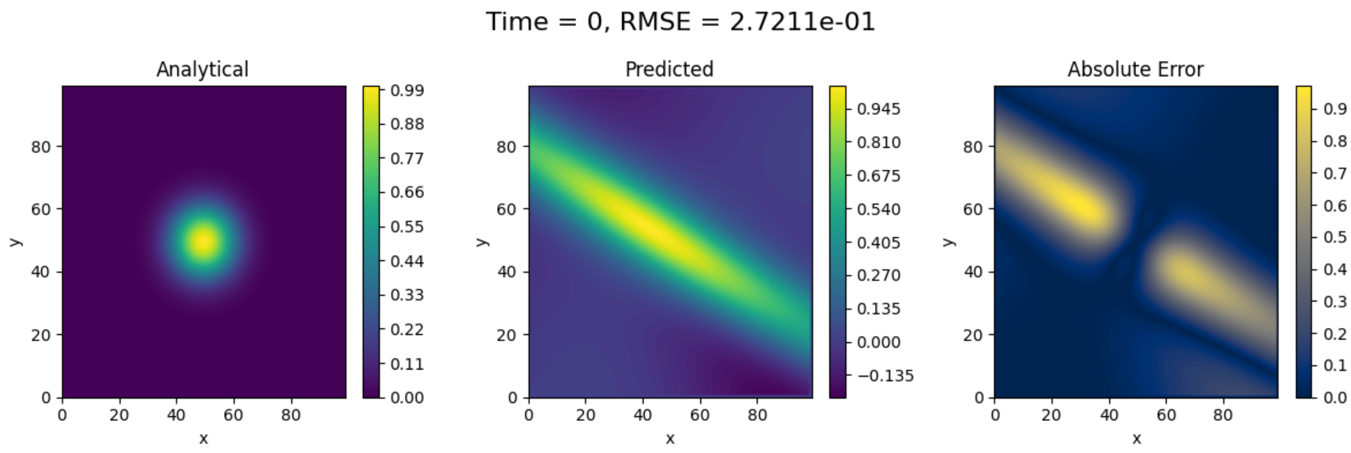

Figure 19. Comparison of analytical and predicted solutions at $t = 0$ using the Tanh activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $2.7211 \cdot 10^{-1}$.

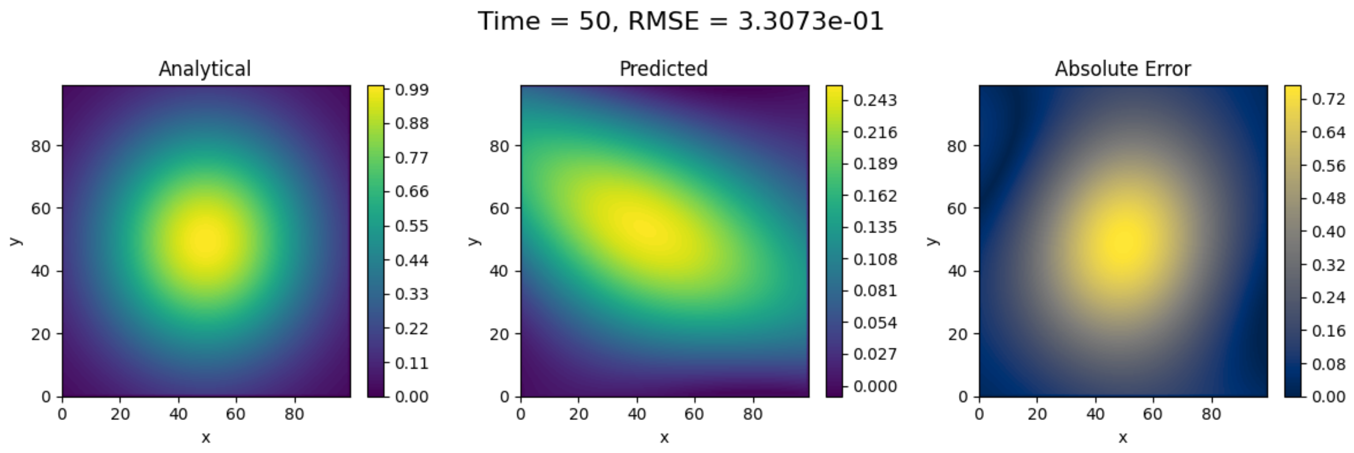

Figure 20. Comparison of analytical and predicted solutions at $t = 50$ using the Tanh activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $3. \cdot 10^{-1}$.

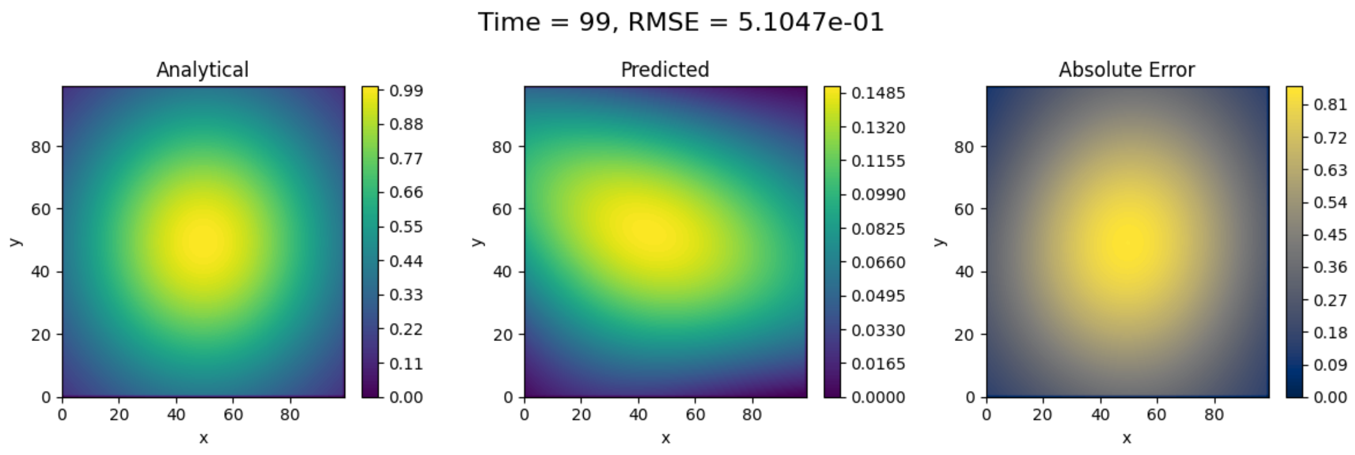

Figure 21. Comparison of analytical and predicted solutions at $t = 99$ using the Tanh activation function. The absolute error plot (right) shows the difference between the analytical and predicted solutions, with an RMSE of $5.1047 \cdot 10^{-1}$.

3. *Results from The Navier-Stokes model*

We start by introducing the network architecture of the Navier-Stokes model, shown in table (II).

PDE:	**Navier-Stokes**
Parameter	**Value**
Activation function	SiLU, Leaky ReLU, Sigmoid and Tanh
Epochs	15000
Model	3 input, 5 layers, 60 neurons per layer
Optimizer	Adam with learning rate 10^{-3} and weight decay 10^{-4}
Scheduler	Exponential with $\gamma = 0.96$

Table II. Model parameters for the Navier-Stokes (PDE) used in this study. The table lists the key parameters including activation functions, training epochs, model architecture, optimizer settings, and scheduler details, and L2 regularization constant $\lambda = 10^{-4}$.

In Figure (22), we observe that the loss converges early to suboptimal minima. This issue may stem from several factors: the network might not be deep enough to capture the complexity of the Navier-Stokes equations, the learning rate might be too high, or the L2 regularization constant might be too low. Additionally, we implemented batch sizing in an attempt to speed up the training process, which introduced another layer of complexity. If the batch size is too small, it introduces noise into the loss, leading to instability; if it is too large, it can cause early convergence.

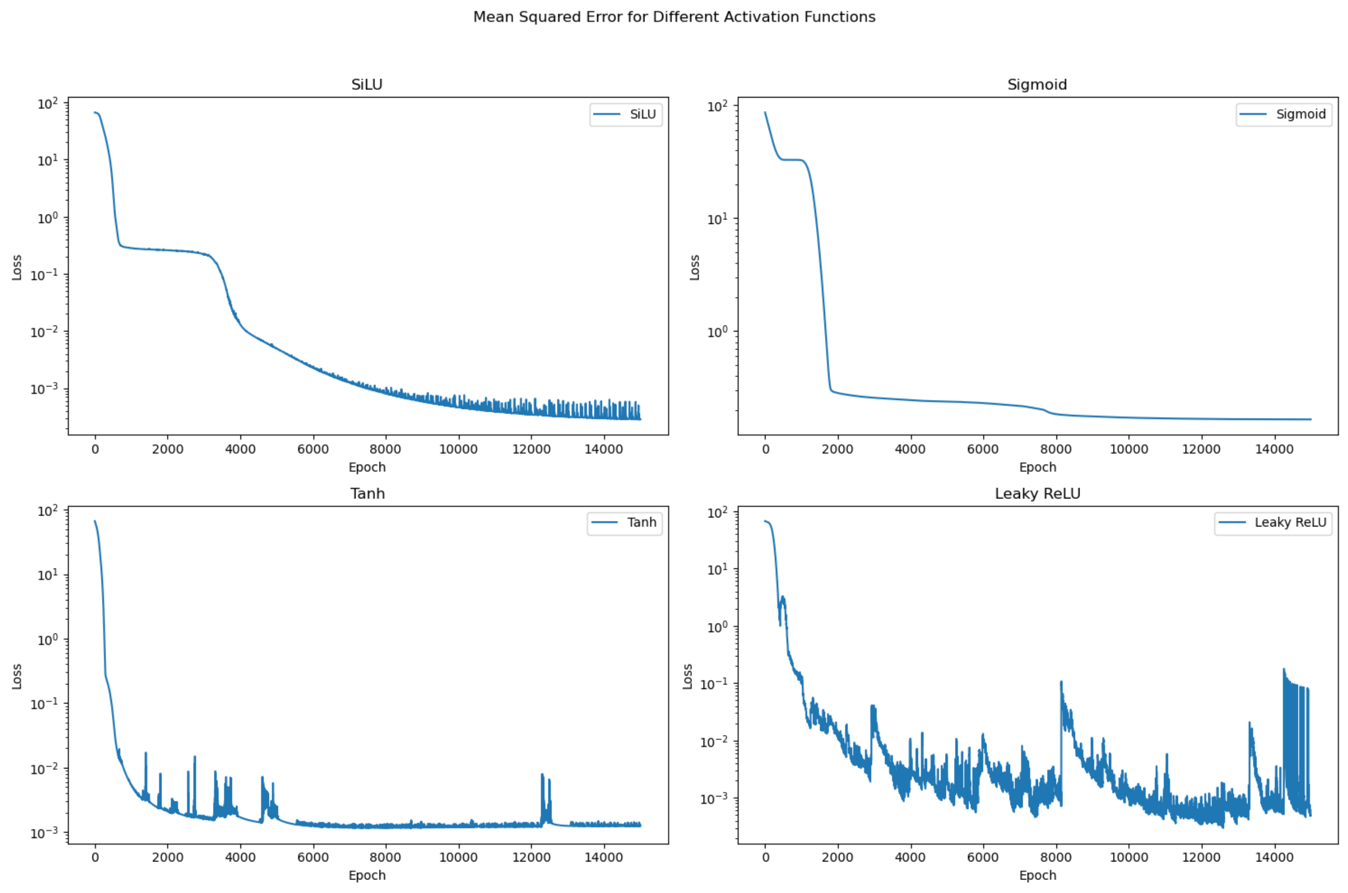

Figure 22. Mean Squared Error for Different Activation Functions: The loss convergence is shown for Leaky ReLU, Tanh, Sigmoid, and SiLU activation functions over 15,000 epochs. Each subplot demonstrates how the loss decreases and stabilizes except Leaky RElU which displays a great amount of noise. This indicates how each activation function performs in minimizing the error during training.

4. The Navier-Stokes models prediction

Observing Figures 23, 24, and 25, it is evident that the SiLU activation function effectively captures the physical properties of the system compared to the other activation functions examined in this study. This effectiveness is partly due to SiLU's ability to capture gradient changes well. Although the morphing of the original water beam does not perfectly match the system, this discrepancy may result from suboptimal hyperparameter tuning.

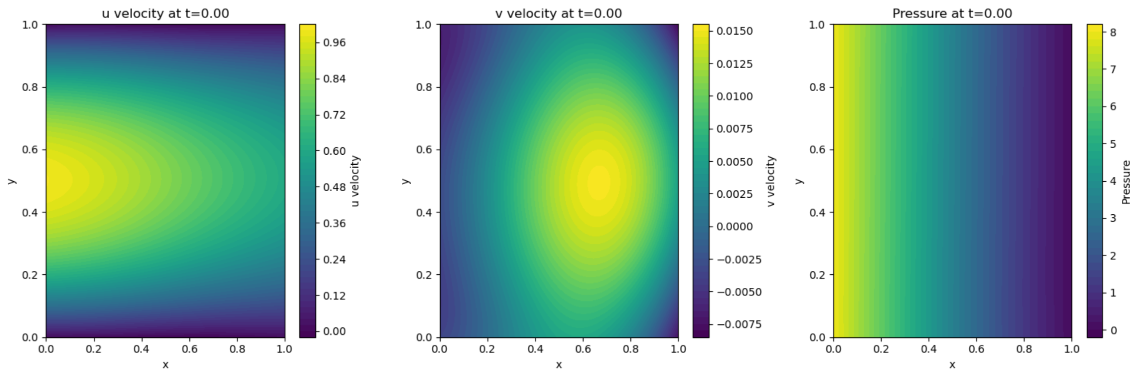

Figure 23. Solution of the Navier-Stokes equations with SiLU activation function at $t = 0.00$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

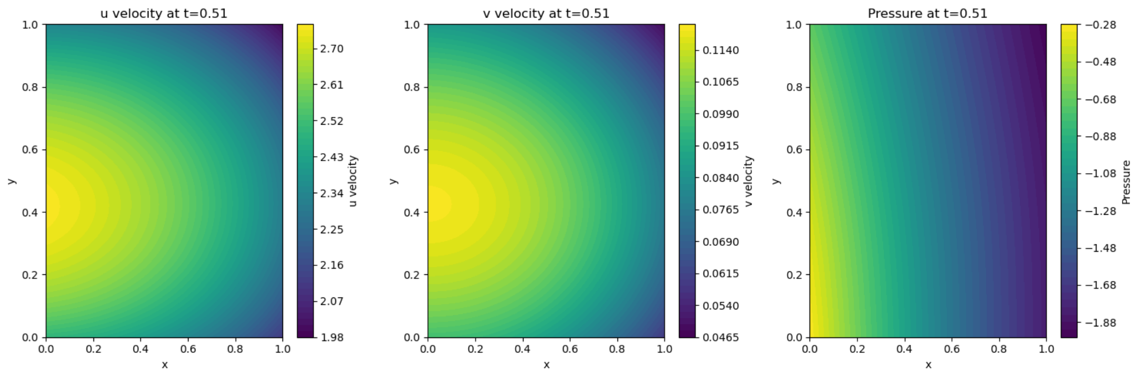

Figure 24. Solution of the Navier-Stokes equations with SiLU activation function at $t = 0.51$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

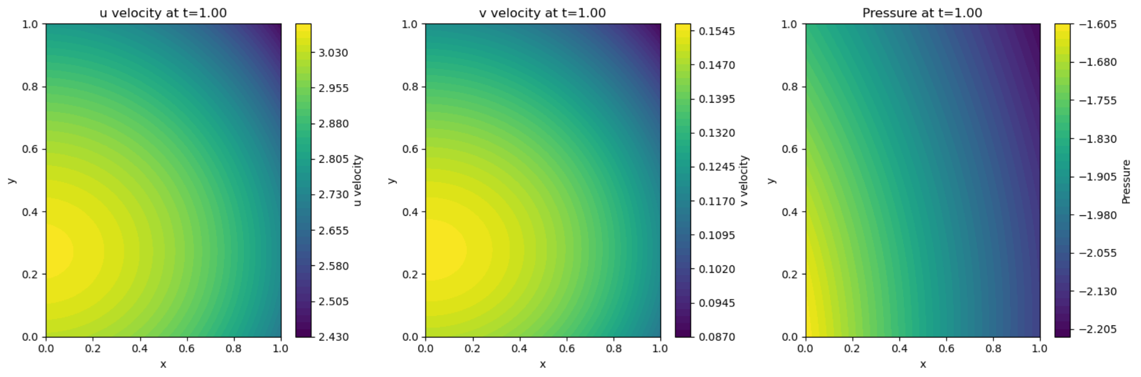

Figure 25. Solution of the Navier-Stokes equations with SiLU activation function at $t = 1.00$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

Observing Figures 26, 27, and 28, it is clear that Leaky ReLU captures the physical properties of the Navier-Stokes equations about as well as SiLU. However, it shows stronger morphing, as also seen in Figure 3 for the diffusion equation. Leaky ReLU appears to predict the pressure gradient much better than the other activation functions.

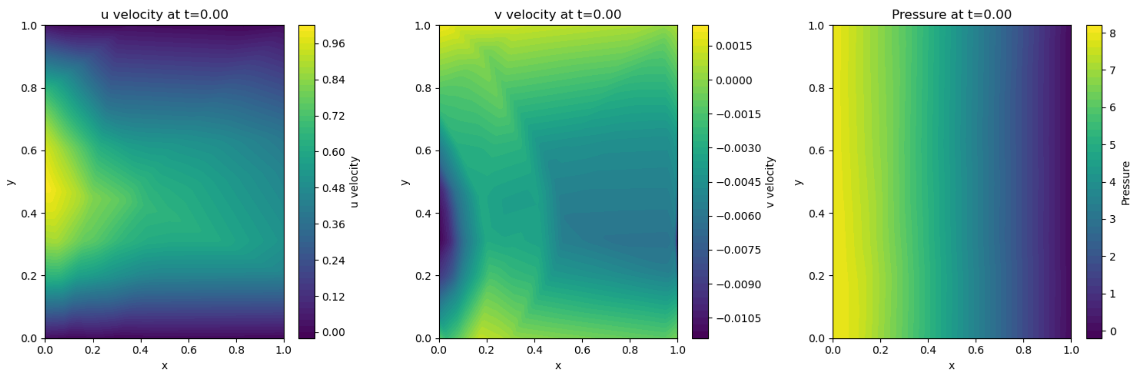

Figure 26. Solution of the Navier-Stokes equations with Leaky ReLU activation function at $t = 0.00$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

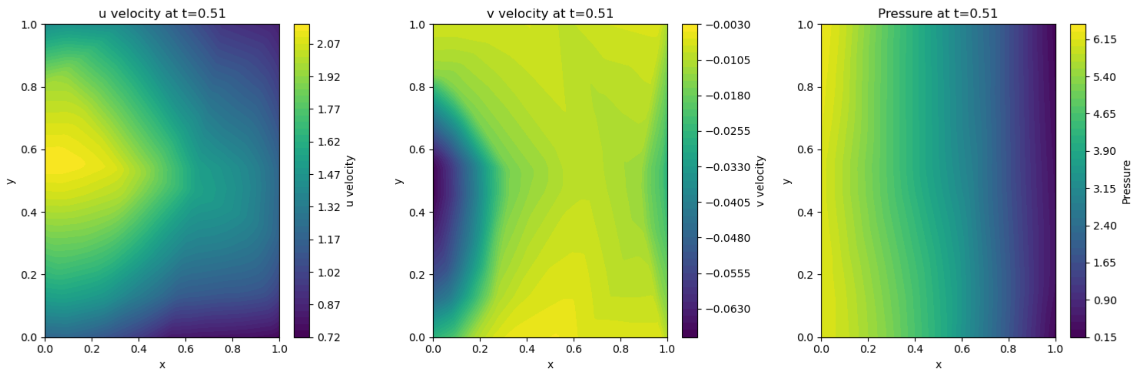

Figure 27. Solution of the Navier-Stokes equations with Leaky ReLU activation function at $t = 0.51$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

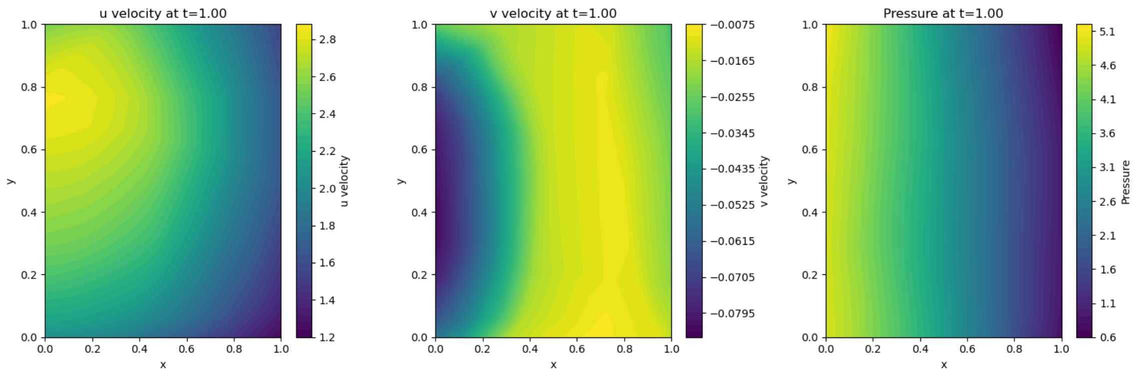

Figure 28. Solution of the Navier-Stokes equations with Leaky ReLU activation function at $t = 1.00$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

One might intuitively think that the sigmoid function is well-suited for achieving a highly contrasted representation of the system's physical phenomena. However, this is not the case, as it completely fails to learn the system's physical properties. The issue likely arises from its binary nature, which tends to produce values close to either 0 or 1. Looking at Figures (29), (30), and (31), it is clear that sigmoid is the worst performing activation function for problems examined in this study.

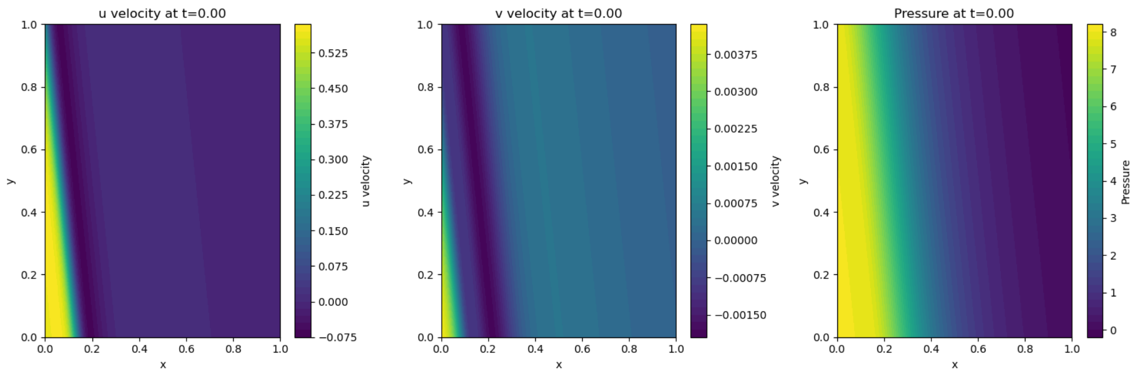

Figure 29. Solution of the Navier-Stokes equations with Sigmoid activation function at $t = 0.00$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

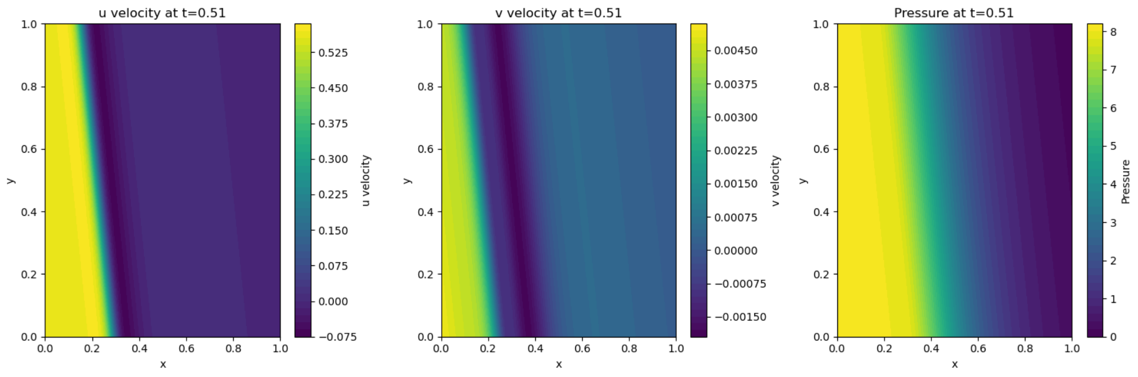

Figure 30. Solution of the Navier-Stokes equations with Sigmoid activation function at $t = 0.51$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

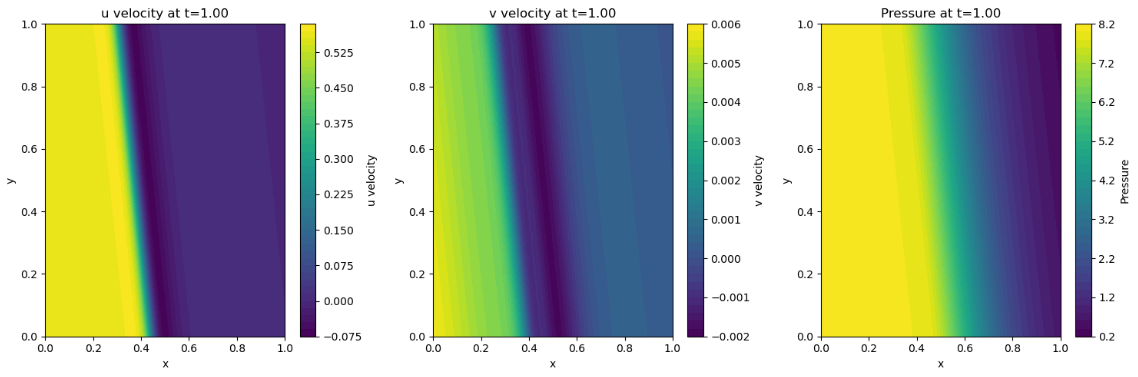

Figure 31. Solution of the Navier-Stokes equations with Sigmoid activation function at $t = 1.00$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

Finally, we observe figures (32), (33), and (34) of the Tanh activation function, which shows great stability over time but lacks the correct geometry of the flow. This stability is evident in figure (22), as the loss curve converges with only a little noise at the end. The warping of the stream shape may stem from training on too large of a batch size, an excessively small learning rate slowing down the convergence process, and potentially landing it in a local minimum, or suboptimal hyperparameters, which will also reduce the quality of the model's output.

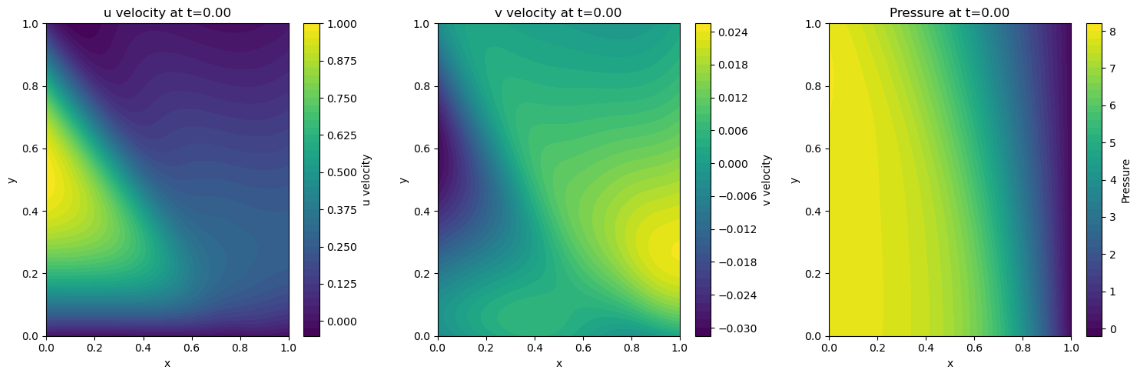

Figure 32. Solution of the Navier-Stokes equations with Tanh activation function at $t = 0.00$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

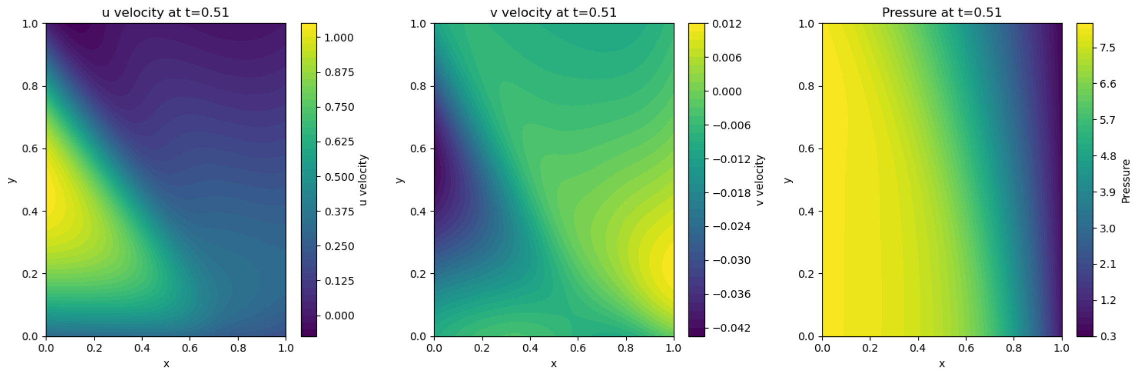

Figure 33. Solution of the Navier-Stokes equations with Tanh activation function at $t = 0.51$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

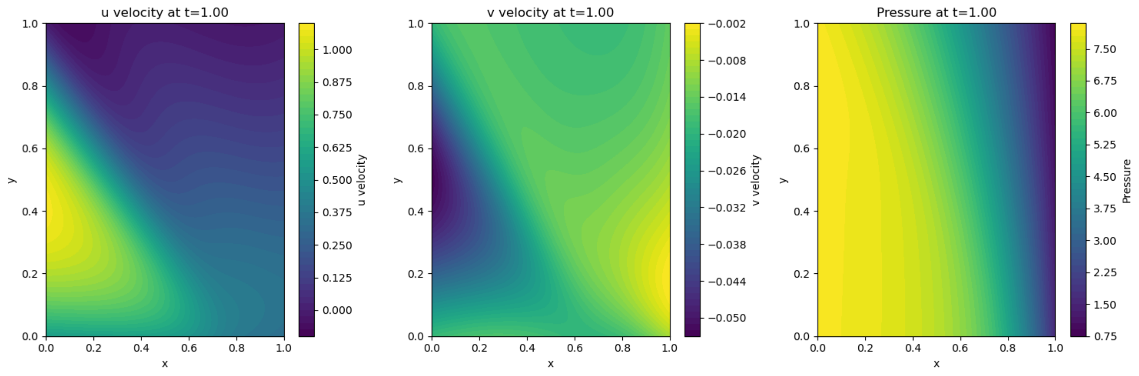

Figure 34. Solution of the Navier-Stokes equations with Tanh activation function at $t = 1.00$: (Left) u velocity, (Middle) v velocity, (Right) pressure.

a. Run Time. It is important to note that all simulations were run on a single CPU with 8 cores. Therefore, one can expect significantly reduced computation time if these simulations were run on GPUs. The easiest way to achieve this is by using CUDA[9] if you have Nvidia graphics cards. From Tables (V) and (VI), it is clear that batch sizing helps improve the computation speed for the Navier-Stokes model. However, the added complexity led to a degradation in the model output, indicating that a more careful approach is necessary.

Activation function - Diffusion equation	**Runtime**	**epochs**
SiLU	46 minutes 11.54 seconds	15000
Leaky ReLU,	14 minutes 10.84 seconds	15000
Sigmoid	29 minutes 5.71 seconds	15000
Tanh	27 minutes 48.16 seconds	15000

Table III. Runtime comparison for different activation functions when solving the diffusion equation.

Activation function - Navier-Stokes equation	**Runtime**	**epochs**
SiLU	16 minutes 4.78 seconds	15000
Leaky ReLU,	10 minutes 53.80 seconds	15000
Sigmoid	14 minutes 31.74 seconds	15000
Tanh	14 minutes 24.56 seconds	15000

Table IV. Runtime comparison for different activation functions when solving the Navier-Stokes equation.

ii. Finite Element Method (FEM)

1. Results

From Figures (35) and (36) we observe the efficiency and accuracy of the rigorous mathematical system FEM. It captures the physical properties with great precision in a short amount of time and maintains a negligible error over time. For a visual representation of the time evolution of the gaussin hill diffusing, see the accompanying GIF here, and for the figure of the Poiseuille flow see figure (36) The accuracy of the model's output is validated in the Tables (V) and (VI). As we can see, FEM uses almost no time to achieve acceptable precision.

Diffusion FEM Error	**Runtime**
Max Error: $1.8372 \cdot 10^{-2}$	12.73 seconds
L2 Error: $5.3404 \cdot 10^{-1}$	———

Table V. Runtime comparison for different activation functions when solving the Diffusion equation with FEM.

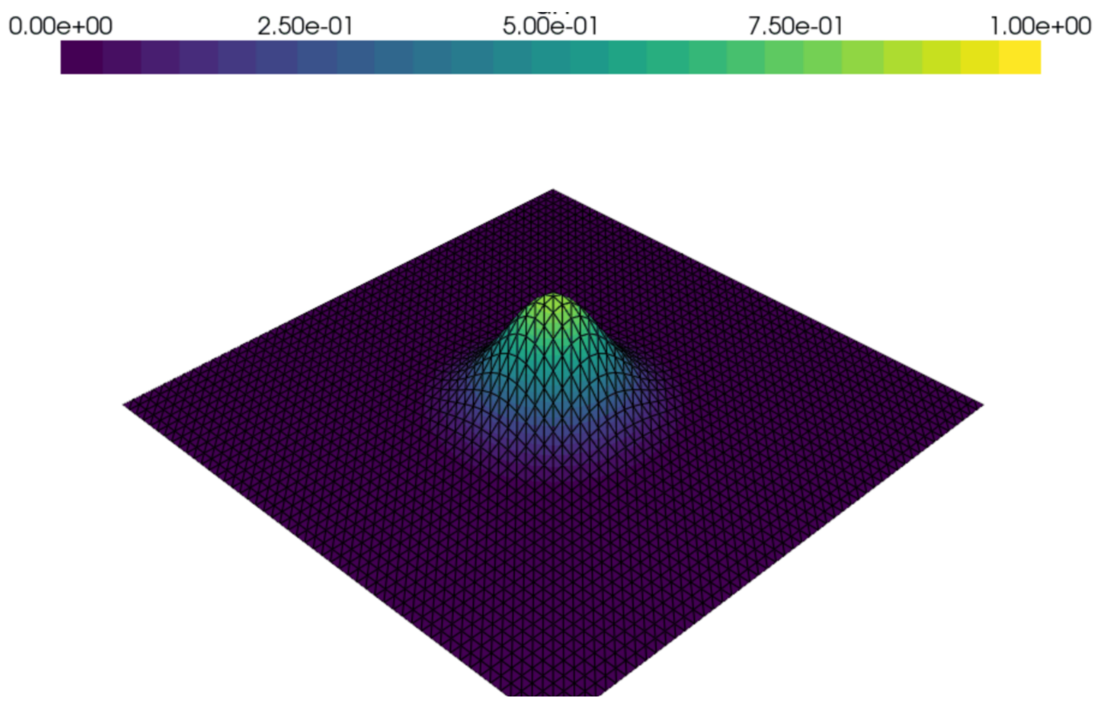

Figure 35. Three-dimensional surface plot illustrating the diffusion of a Gaussian distribution over a structured mesh grid. The plot shows the diffusion characteristics of the Gaussian, with the peak centered at the origin and gradually diminishing towards the edges.

Navier-Stokes FEM Error	**Runtime**
Max Error: $1.05 \cdot 10^{-5}$	10.00 seconds
L2-error: $3.31 \cdot 10^{-6}$	——

Table VI. Runtime comparison for different activation functions when solving the Navier-Stokes equation.

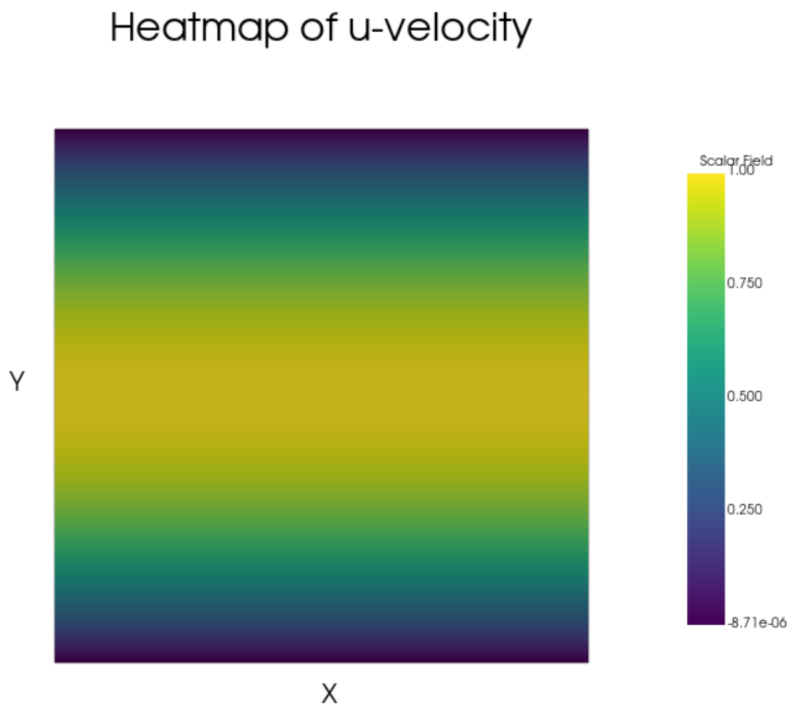

Figure 36. Heatmap of the u-velocity component obtained from the finite element method (FEM) simulation of the Navier-Stokes equations. The color bar on the right represents the scalar field of the velocity magnitude, indicating regions of different flow speeds.

6. Conclusion

This study aimed to compare the Finite Element Method (FEM) and Physics-Informed Neural Networks (PINNs) for solving the two-dimensional Navier-Stokes and Diffusion equations. Our rigorous analysis examined the impact of hyperparameters such as network architecture, regularization, training iterations, and activation functions.

For the Diffusion equation, the SiLU activation function emerged as the most effective, providing accurate initial condition predictions and demonstrating rapid loss convergence. The SiLU function achieved a minimum loss of $1.0 \cdot 10^{-4}$ after approximately 8000 epochs, as shown in Figure (6). The Leaky ReLU function achieved the lowest loss for the Diffusion model of $1.0 \cdot 10^{-6}$ but struggled with accurately capturing the Gaussian hill, as discussed in Section 5 i 2 and shown in Figure (3). In contrast, the Sigmoid functions showed significant discrepancies, with the Sigmoid failing to capture gradient changes shown in Figure (4). Tanh warped the Gaussian hill along the diagonal in Figure (5) but captured the physical properties of the system well.

The Navier-Stokes equation's increased complexity highlighted the limitations of our current network architecture and hyperparameter settings. The SiLU function performed reasonably well, with a minimum MSE of $5.0 \cdot 10^{-4}$, as shown in Figure (22). However, the loss convergence indicated the need for deeper networks or adjusted learning rates and regularization constants. Runtime analysis suggested the potential benefits of GPU acceleration, with CPU runtimes for the Navier-Stokes simulations being 56 minutes for SiLU and 39 minutes for Leaky ReLU, indicating substantial room for improvement in computation speeds Table (VI).

Comparatively, the FEM demonstrated superior accuracy and computational efficiency for both equations. For the Diffusion equation, FEM achieved a maximum error of $1.8372 \cdot 10^{-2}$ and an L2 error of $5.3404 \cdot 10^{-1}$ with a runtime of just 12.73 seconds Table (V). For the Navier-Stokes equation, FEM achieved a maximum error of $1.05 \cdot 10^{-5}$ and an L2 error of $3.31 \cdot 10^{-6}$ with a runtime of 10.00 seconds, as shown in Table (VI). This underscores FEM's robustness and reliability for solving complex PDEs.

To conclude, selecting appropriate activation functions and tuning hyperparameters in PINNs is crucial for achieving accurate and stable predictions. While PINNs offer a promising approach, further research and optimization are necessary to match the performance and efficiency of traditional methods like FEM. Future work should focus on exploring advanced network architectures, adaptive learning strategies, and leveraging GPU acceleration to improve PINNs' applicability and scalability for solving complex physical systems.In fairness, it should be noted that PINNs excel in solving high-dimensional problems, an area where FEM often struggles. Therefore, selecting problems more suited to this strength in future research would provide a more realistic benchmark for comparing PINNs to FEM

Acknowledgments

I would like to thank Jørgen S. Dokken for providing essential code and guidance for the Navier-Stokes channel flow and the diffusion of a Gaussian curve, as found in the tutorial on dolfinx-tutorial. His contribution was invaluable to the success of this project.

[1] Automatic differentiation using torch. https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html. Accessed: 2023-12-10.

[2] Jørgen S. Dokken. The fenicsx tutorial, 2023. Accessed: 2024-06-10.

[3] J.G. Garnier and A. Quetelet. *Correspondance mathématique et physique*. Number v. 10. M.Hayez, imprimeur, 1838.

[4] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

[5] Zheyuan Hu, Khemraj Shukla, George Em Karniadakis, and Kenji Kawaguchi. Tackling the curse of dimensionality with physics-informed neural networks, 2023.

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

[7] Yann LeCun, D Touresky, G Hinton, and T Sejnowski. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, pages 21–28. San Mateo, CA, USA, 1988.

[8] Siddhartha Mishra and Roberto Molinaro. Estimates on the generalization error of physics-informed neural networks for approximating a class of inverse problems for PDEs. *IMA Journal of Numerical Analysis*, 42(2):981–1022, 06 2021.

[9] NVIDIA Corporation. Cuda toolkit, 2024. Accessed: 2024-06-13.

[10] Murat Sazli. A brief review of feed-forward neural networks. *Communications Faculty Of Science University of Ankara*, 50:11–17, 01 2006.