SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Computational Science and Engineering

# Using neural networks with domain decomposition to solve partial differential equations

Aditya Kishor Phopale


SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

TECHNICAL UNIVERSITY OF MUNICH


Master's Thesis in Computational Science and Engineering

# Using neural networks with domain decomposition to solve partial differential equations


| | |
|---|---|
| Author: | Aditya Kishor Phopale |
| Supervisor: | Prof. Dr. Felix Dietrich |
| Advisor: | Dr. Dirk Hartmann |
| Submission Date: | May 1, 2024 |


I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching bei München, May 1, 2024 Aditya Kishor Phopale

# Acknowledgments

Firstly, I would like to express my sincere thanks to Prof. Dr. Felix Dietrich and Dr. Dirk Hartmann (Siemens), who gave me the opportunity to work on this thesis project. I am indebted to them for sharing expertise, valuable guidance, and encouragement extended to me.

Next, I would like to express special thanks to Dr. Amit Chakraborty (Siemens) and Msc. Chinmay Datar for sharing their experience and being available for any queries or discussions throughout the project.

I would like to thank my parents for their support throughout my life and for giving me the confidence to pass any obstacles.

Lastly, I also would like to thank one and all who directly or indirectly, have lent their hand in this venture.

# Abstract

Solving Partial Differential Equations (PDEs) is fundamental to understanding various physical phenomena across various disciplines. Traditional solvers in industry often face challenges such as computational complexity, mesh generation, and convergence issues. In recent years, neural networks have emerged as a promising alternative for tackling PDEs due to their ability to approximate complex functions and learn underlying patterns from data. This thesis presents novel advancements in solving PDEs using neural networks, coupled with innovative approaches in network initialization and domain decomposition techniques. Firstly, a neural network-based solver is introduced to solve PDEs efficiently over square domains. Employing a neural network architecture with a single hidden layer, emphasis is placed on training only the final layer to directly solve PDEs without the need for explicit data. This methodology handles non-time-dependent PDEs with boundary conditions, including linear and non-linear equations. Secondly, the thesis explores domain decomposition strategies combined with neural networks for tackling complex PDE problems. By partitioning the domain into subdomains, each has its neural network approximator. The coupling of neural networks at shared boundary points ensures solution continuity over the entire domain and accuracy in the overall solution. Furthermore, we dive into the "Sampling Where It Matters" (SWIM) concept, aimed at enhancing the initialization of neural networks for improved performance in various tasks, including PDE solving. SWIM leverages the gradient of the truth function along with input points to intelligently initialize network parameters, leading to more efficient convergence and enhanced solution accuracy. Through several experiments, the efficacy and versatility of the proposed methodologies are demonstrated, including a comparative study against existing methods such as Physics-Informed Neural Networks (PINNs) and Finite Element Method (FEM). This comparative analysis showcases where the presented method stands against existing methods, providing valuable insights into their performance and applicability.

# Contents

# 1 Introduction

In recent years, the intersection of scientific computing and machine learning has led to remarkable advancements in the field of solving Partial Differential Equations (PDEs) using neural networks. Along with academic interest in this field, there has also been a surge in industrial interest. Advances in hardware, such as GPUs and distributed computing frameworks, have made it feasible to train large-scale neural networks and deploy them in parallel on high-performance computing clusters. This scalability and parallelism enable neural network-based solvers to tackle increasingly complex and computationally demanding problems in industry. Companies that adopt these neural network-based PDE solvers early can gain a competitive edge by accelerating innovation, reducing time-to-market, and improving product performance. By leveraging cutting-edge AI technologies, they can differentiate their products and services in the marketplace and drive business growth.

One can observe a notable shift in research from traditional numerical methods to approaches that directly learn solutions from equations without relying on predefined basis functions or meshes. One such technique is Physics-Informed Neural Network (PINN) [RPK19]. This technique converts solving a PDE into an optimization problem. The solution is learned by minimizing the residual of the governing equations, boundary, and initial conditions by updating the network parameters via the conventional training of neural networks using backpropagation. The network is trained once the loss value has converged or is below a certain predefined threshold value. The solution of the PDE is represented by a neural network using the trained parameters. These methods can also be combined with the availability of data by adding a term in the loss function; this has an additional benefit in that it utilizes a data-driven approach, where solutions are approximated based on patterns learned from data along with explicit physical equations. Many of the fundamental concepts behind PINN were proposed in the 1990s by [LLF98]. In recent years, several effective neural network-based methods for PDEs have surfaced, such as Deep Operator Network (DeepONet) that learn the mapping from any functional parametric dependence to the solution; thus, they learn an entire family of PDEs [Lu+21]. Fourier Neural Operator (FNO) learning operators based on parameterizing the integral kernel in the Fourier space [Li+21], and Deep Galerkin Method (DGM), a meshless deep learning algorithm to solve high dimensional PDEs [SS18].

Neural network-based solvers have some advantages, such as solving PDEs without explicit mesh generation, simplifying the implementation process, and enabling efficient

handling of irregular geometries. This feature streamlines solver development and enhances applicability to diverse problem domains since generating a mesh is one of the most time-consuming step in the simulation pipeline. Additionally, using neural networks eliminates the need for in-depth knowledge of numerical discretization techniques and stability considerations, making it accessible to a broader spectrum of researchers and practitioners with a primary focus on the physics of the problem. Furthermore, once trained, neural network-based solvers offer fast evaluation of solutions, facilitating near-real-time predictions crucial for problems such as weather simulations. While neural network-based solvers for PDEs offer promising advantages, but they do have some limitations. The first limitation is the solution accuracy [JKK20]. Neural network-based solvers can provide solutions for many PDEs, but they may struggle with achieving the same level of accuracy as traditional numerical methods. Conventional methods like Finite Difference Method (FDM) or Finite Element Method (FEM) are often based on well-established mathematical principles and can achieve highly accurate solutions provided sufficient mesh resolution and computation time. The second limitation is with the increase in number of layers and nodes in the hidden layers, the improvement in the solution accuracy is not always guaranteed. Another limitation is the computational cost. Neural network-based solvers typically require significant computational resources and time for training, particularly for large-scale problems. In contrast, conventional solvers can solve with similar accuracy in a comparatively short time. Because of their limited accuracy and substantial computational demands, the general consensus is that presently, neural network-based solvers struggle to compete with classical numerical methods in effectiveness [HK24].

In the thesis, we focus on the Local Extreme Learning Machine (locELM) method introduced in [DL21] that solves linear/non-linear PDEs with domain decomposition. The idea introduced in this work was to freeze the weights of all the layers to the initial value except the linear layer. This makes the optimization problem simple since only a few parameters need to be trained compared to the conventional training of neural networks. It also combines domain decomposition to obtain the solution of PDE over multiple subdomains, where each subdomain has its neural network to approximate the solution. This requires additional continuity constraints being forced on the common boundaries of the subdomains. It also introduces a block time-marching scheme together with the method mentioned above for long-time simulations of time-dependent linear/non-linear partial differential equations. Later [Che+22] introduces a new approach of domain decomposition that uses partition of unity or overlapping domain decomposition, which results in skipping the forcing of additional continuity constraints on the shared points between the subdomain as the solution over each subdomain smoothly transitions to other subdomains. In the thesis, we mainly look into the initialization of neural networks for each subdomain. We compare random initialization with Sampling Where It Matters (SWIM) initialization [Bol+23]. SWIM initialization uses the input data and truth label to place activation functions between input points where the gradient of the truth label

is high. This helps initialize more activation functions at the location where the true label has a high gradient magnitude.

The thesis is structured as follows: Chapter 2 introduces neural networks and explains the universal approximation theorem of neural networks for function approximation. Different initialization techniques are presented, the SWIM method is introduced. Chapter 3 describes implementation details of the locELM method in solving PDEs. It also contains the results of experiments on linear Poisson PDEs and also presents the results of the extension of the implementation to non-linear PDEs. Finally, we discuss the conclusion and future research directions in Chapter 4.

# 2 Background theory

## 2.1 Introduction to neural networks

Neural networks are a fundamental component of modern machine learning. Inspired by the biological neural networks of the human brain [Ako13], Artificial Neural Networks (ANNs) are computational models composed of interconnected nodes, known as neurons, which work collectively to process and learn from data.

### 2.1.1 The Neuron

At the core of neural networks lies the neuron, the basic computational unit. A neuron receives input signals from other neurons or input data, processes these inputs, and generates an output signal. The key components of a neuron are:

- **Input Weights ($w$):** Each input signal is associated with a weight, which determines its significance in influencing the neuron's output.
- **Activation Function ($\sigma$):** After aggregating the weighted inputs, the neuron applies an activation function to introduce non-linearity and determine the output signal.
- **Bias ($b$):** A bias term is added to the weighted sum before applying the activation function, allowing the neuron to better fit the data.

The output of a neuron $j$ in layer $l$ is computed as follows [Bis06]:

$$\begin{aligned} z_j^l &= \sum_{i=1}^{n^{(l-1)}} w_{ji}^l x_i^{(l-1)} + b_j^l \\ x_j^l &= \sigma(z_j^l) \end{aligned} \tag{2.1}$$

Where:

- $z_j^l$ is the weighted sum of inputs to neuron $j$ in layer $l$.
- $n^{l-1}$ is the number of neurons in the previous layer.
- $w_{ji}^l$ is the weight of the connection between neuron $i$ in layer $l-1$ and neuron $j$ in layer $l$.
- $x_i^{(l-1)}$ is the output of neuron $i$ in layer $l-1$.

- $b_j^l$ is the bias of neuron $j$ in layer $l$.
- $\sigma$ is the activation function.

### 2.1.2 Structure of Neural Networks

Neural networks are organized into layers, with each layer comprising multiple neurons. The three main types of layers in a neural network are:

- **Input Layer:** Receives the initial data input and passes it to the subsequent layers.
- **Hidden Layers:** Intermediate layers between the input and output layers. These layers perform complex transformations on the input data through interconnected neurons.
- **Output Layer:** Produces the final output of the neural network.

The connections between neurons are characterized by weights, which are adjusted during the training process to optimize the network's performance.

## 2.2 Training Neural Networks

Training a neural network involves adjusting its parameters (weights and biases) to minimize the difference between the predicted outputs and the actual targets. This process typically involves two main steps: forward propagation and backpropagation.

### 2.2.1 Forward Propagation

During forward propagation, input data is fed into the neural network, and the activations of neurons are computed layer by layer until the output is generated. The steps involved in forward propagation are as follows:

1. **Input Propagation:** The input data is fed into the input layer.
2. **Hidden Layer Computations:** The weighted inputs are aggregated, biases are added, and activation functions are applied in each hidden layer as given in eq. 2.1.
3. **Output Layer Computations:** The process continues through the hidden layers until the output layer produces the final prediction.

### 2.2.2 Backpropagation

Backpropagation is the process of computing gradients of the loss function with respect to the network's parameters. These gradients are then used to update the parameters through optimization algorithms such as gradient descent [RHW86]. The steps involved in backpropagation are:

1. **Compute Loss:** Calculate the difference between the predicted outputs and the actual targets using a suitable loss function, such as Mean Squared Error (MSE):

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

   Where:

   - $N$ is the number of training examples.
   - $y$ is the vector of actual target values.
   - $\hat{y}$ is the vector of predicted values.

2. **Compute Gradients:** Compute the gradients of the loss function with respect to the parameters of the network using the chain rule.

3. **Update Parameters:** Use an optimization algorithm, such as gradient descent, to update the parameters in the direction that minimizes the loss.

### 2.2.3 Optimization Techniques

Several optimization techniques are employed to enhance the training process and improve the performance of neural networks:

- **Gradient Descent:** A first-order optimization algorithm that updates the parameters in the direction of the negative gradient of the loss function [RHW86]. The weight update rule for gradient descent is given by:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial L}{\partial w_{ij}^{(l)}}$$

  Where:

  – $w_{ij}^{(l)}$ is the weight connecting neuron $i$ in layer $l-1$ to neuron $j$ in layer $l$.
  – $\alpha$ is the learning rate, determining the step size of the update.

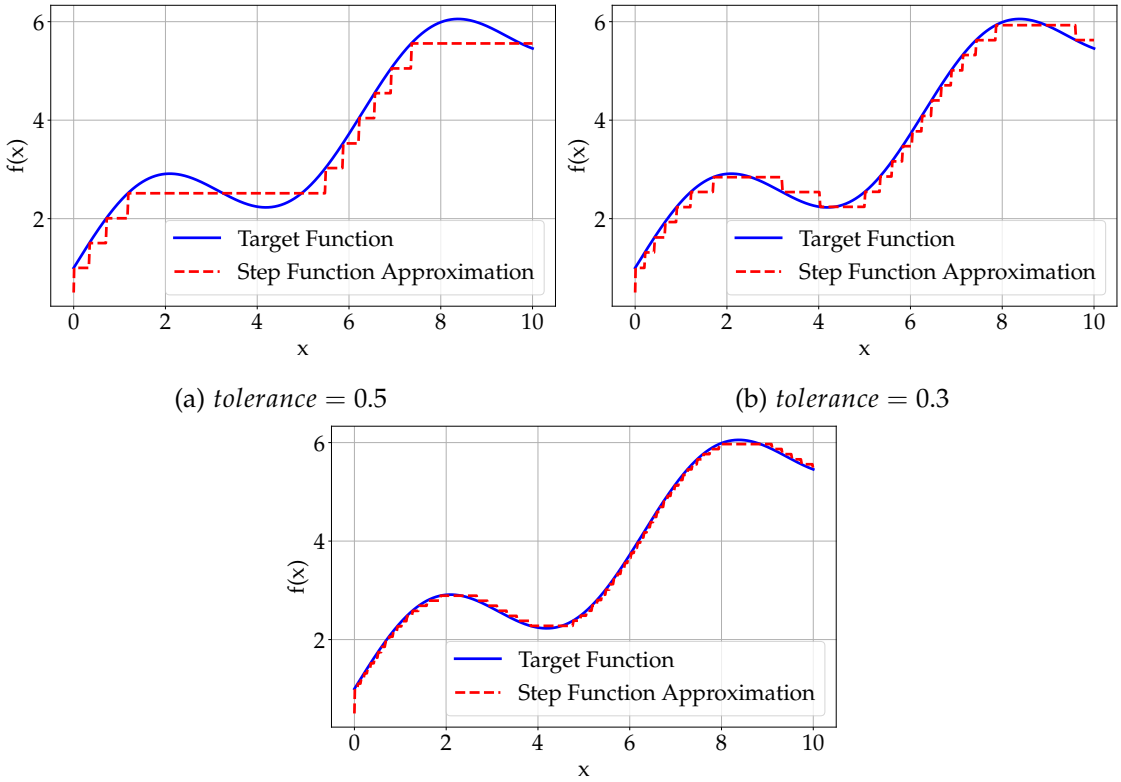
- **Stochastic Gradient Descent (SGD):** A variant of gradient descent that updates the parameters using a random subset of training examples at each iteration, reducing computation time and memory requirements [BCN18].

- **Adam:** An adaptive optimization algorithm that maintains separate learning rates for each parameter and adjusts them based on past gradients, improving convergence speed and performance [KB17].

- **Broyden-Fletcher-Goldfarb-Shanno (BFGS):** This technique aims to iteratively improve upon an initial estimate of the solution by approximating the inverse Hessian matrix [Fle70].

- **Least squares method:** Least-squares estimation provides a means of determining estimates of model parameters that are optimal in minimizing the sum of the squares of the estimation errors [Fad24]. This optimization method is particularly interesting from the point of view of this thesis as we use it to train only the linear layer parameters of the neural network to solve PDEs. The process of constructing the least squares system is mentioned later in the section 3.1.4.

## 2.3 Universal Approximation Theorem of neural networks




(a) *tolerance* = 0.5

(b) *tolerance* = 0.3

(c) *tolerance* = 0.1

Figure 2.1: Function approximation using superposition of step functions for different error tolerances

The Universal Approximation Theorem (UAT) is a foundational result in the theory of neural networks, providing a fundamental understanding of their expressive power and capabilities. Formally stated, the theorem asserts that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function to arbitrary accuracy, given a sufficiently large number of neurons and appropriate activation functions [Cyb89]. This theorem underscores the remarkable versatility of neural networks in modeling complex relationships between inputs and

outputs, making them powerful tools for a wide range of tasks in machine learning.

In fig. 2.1, we observe the function approximation using the superposition of step functions for different error tolerances. These step functions can be thought of as different neurons in the single hidden layer of a neural network, with the activation function set as a step function. As the number of neurons in the hidden layer increases, the neural network can approximate the target function more accurately, highlighting the universal approximation capabilities of neural networks. Some key observations from the fig. 2.1 for a neural network used for function approximation in 1D are:

- The neural network can approximate the target function with increasing accuracy as the number of neurons in the hidden layer increases.
- For every neuron to contribute to the approximation of a function, the shift in the step function corresponding to that neuron is always within the domain.
- More step functions are placed in the region where the function gradient magnitude is high to approximate a function within the given error tolerance.

We will later see how these observations are handled using the SWIM method. In the present work, we obtain the solution by freezing the hidden layer parameters and only optimizing the linear layer parameters so that the neural network's output satisfies the PDE. This strategy offers both advantages and disadvantages. One significant advantage lies in simplifying the optimization problem, as it reduces the number of parameters being updated during training, leading to potentially faster convergence and computationally less intensive optimization processes. However, this approach also carries certain disadvantages, such as a reduction in the expressive power of the neural network. By freezing the hidden layer parameters, the network loses the ability to adapt and learn from the data at intermediate stages, potentially limiting its capacity to capture intricate patterns and representations within the dataset. In the next section, we will see how we can improve the initialization of the neural network using the SWIM method.

## 2.4 Sampling neural network parameters

Initialization of the network parameters is a crucial step in setting up for effective learning. Essentially, it involves assigning initial values to the neural network's parameters (like weights and biases). The initialization of neural networks becomes extremely important since freezing the hidden layer parameters decreases the expressive power of neural networks. While random initialization was commonly used, more sophisticated methods have emerged to ensure smoother and faster learning. For example, techniques like Xavier initialization [GB10] and He initialization [He+15] have become popular choices. Xavier initialization aims to keep the variance of activations and gradients consistent across layers, while He initialization is tailored for networks using Rectified Linear Units (ReLUs) as activation functions. These methods help address challenges

like vanishing or exploding gradients, ensuring the network can learn effectively. Choosing the proper initialization method can significantly impact the network's performance during training and its ability to generalize to new data.

The thesis predominantly investigates the SWIM method, a neural network approach where each hidden layer's weight and bias pairs are entirely determined by two points from the input space, with only the final layer being trained using the least squares method. The outline of the method is that it strategically places the activation function between input points with a high gradient magnitude of the truth label between them based on a probability distribution. The process involves sampling a pair of input point combinations from a uniform distribution, followed by selecting the pairs based on a conditional probability distribution, which depends on the gradient of the true label. This technique aims to enhance the network's ability to capture crucial features and relationships within the data by targeting such pairs. This approach links the initialization process with the network's capacity to efficiently learn from the training data, potentially leading to improved performance and generalization capabilities. The conditional probability distribution is defined as follows:

$$p^{(l)}(x_0^{(1)}, x_0^{(2)} | \{\mathbf{W}_j, b_j\}_{j=1}^{l-1}) \propto \begin{cases} \frac{\|f(x_0^{(2)}) - f(x_0^{(1)})\|_y}{\|x_{l-1}^{(2)} - x_{l-1}^{(1)}\|_{x_{l-1}}} & \text{if } x_{l-1}^{(1)} \neq x_{l-1}^{(2)} \\ 0, & \text{otherwise,} \end{cases} \tag{2.2}$$

where, $x_0^{(1)}$ and $x_0^{(2)}$ are the pair of input points sampled from the input space, $x_{l-1}^{(1)}$ and $x_{l-1}^{(2)}$ are the hidden layer prediction of previous layer for the pair of input points $x_0^{(1)}$ and $x_0^{(2)}$, $\mathbf{W}_j$ and $b_j$ are the weights and biases of the $j^{th}$ neuron in the hidden layer. $f(x_0^{(2)})$ and $f(x_0^{(1)})$ are the true label for the pair of input points $x_0^{(1)}$ and $x_0^{(2)}$ respectively. These weights and biases in the SWIM method are given as follows

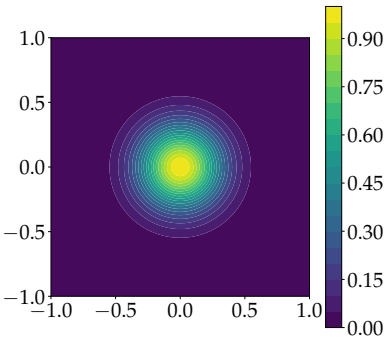
$$\mathbf{W}_{l,i} = s_1 \frac{x_{l-1,i}^{(2)} - x_{l-1,i}^{(1)}}{\|x_{l-1,i}^{(2)} - x_{l-1,i}^{(1)}\|^2}, \quad b_{l,i} = \langle \mathbf{W}_{l,i}, x_{l-1,i}^{(1)} \rangle + s_2, \tag{2.3}$$

where, $s_1$ and $s_2$ are scalar constants, $x_{l-1,i}^{(1)}$ and $x_{l-1,i}^{(2)}$ are hidden layer prediction of previous layer for the $i^{th}$ pair of input points $x_{0,i}^{(1)}$ and $x_{0,i}^{(2)}$ whereas the output layer weights are set by minimizing the loss function. The scalars $s_1$ and $s_2$ influence the placement of the activation function between the input points. Currently, SWIM method works for ReLU and tanh activation functions as follows:

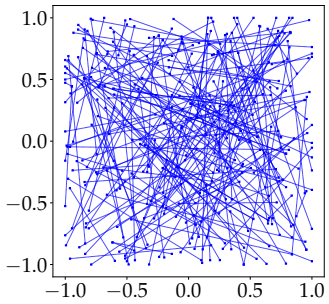
- For ReLU activation, $s_1$ is set to 1 and $s_2$ to 0, mapping $x^{(1)}$ and $x^{(2)}$ to 0 and 1 respectively in the output of the activation function.
- For tanh activation, $s_1$ is set as $s_1 = 2s_2$ and $s_2$ is set as $s_2 = \ln(3)/2$, mapping $x^{(1)}$ and $x^{(2)}$ to $-0.5$ and 0.5 respectively in the output of the activation function.

The SWIM method also has an option to select the pairs of input points uniformly and not based on the gradient of the true label. This option is useful when no true label

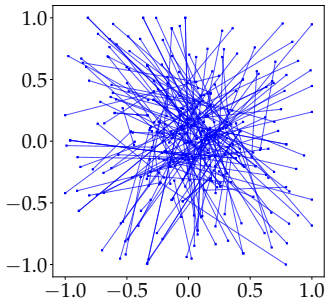
is available and the network has to be trained using the SWIM method. In the context of using neural networks for solving PDEs, the uniform sampling of input points can be beneficial since we do not have the solution of the PDE before solving it. Another option SWIM method provides is to prune duplicate activation functions. Since the pair of input points are sampled based on a conditional probability distribution, there is a possibility that the same pair of input points is sampled multiple times. These duplicate activation function do no contribute to the approximation of the function and can be pruned. In all the experiments we present in section 3.2, we always use the pruning option to remove the duplicate and improve the computational time as it results in less linear layer parameters to solve.



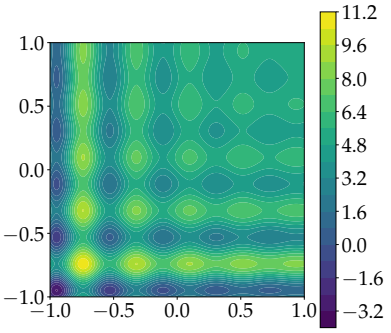

(a) Gaussian function contour plot



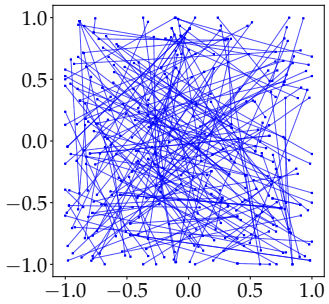

(b) Uniform sampling of pair of input points for Gaussian function



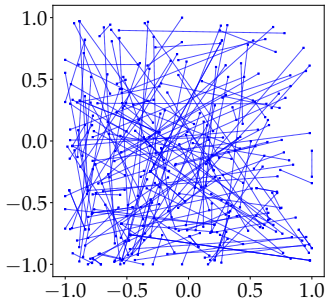

(c) SWIM sampling of pair of input points for Gaussian function




(d) Decaying sine function contour plot




(e) Uniform sampling of pair of input points for Decaying sine function




(f) SWIM sampling of pair of input points for Decaying sine function

Figure 2.2: Function to be approximated and 200 pair of input points sampled using SWIM method

In the discussion about the basis functions in section 2.3, we mentioned the importance of placing activation functions such that the shift of the basis functions are within the domain, and more activation functions are placed in the region of the domain where the gradient of the true function is high. In the fig. 2.2, we can observe the pair of

input points selected for placing the activation function for two different functions. The shift will always be inside the domain since the input points are used to place the activation function. From fig. 2.2c and 2.2f we can also observe that with the SWIM method, more pairs of input points are sampled in the region where the gradient of the function being approximated is high compared to the uniform sampling of input points given in fig. 2.2b and 2.2e. The advantage of using a uniformly sampled pair of input points over random initialization is that the input points are still used to place the activation function, and the shift of the activation function is within the domain. The only downside compared to the SWIM method is that since we do not have the solution, the activation functions are placed uniformly instead of placing more activation functions in the region where the gradient of the true function is high.

# 3 Solving partial differential equations using sampled networks

## 3.1 Implementation details

In this section, we will go through the process to obtain a solution for a linear/non-linear PDE using neural networks with domain decomposition.

Consider the following problem

$$\begin{aligned} \mathcal{L}u(x) + \mathcal{N}u(x) &= f(x) \quad x \in \Omega, \\ \mathcal{B}u(x) &= g(x) \quad x \in \partial\Omega, \end{aligned} \tag{3.1}$$

where $x \in \Omega \subset \mathbb{R}^{d_x}$ and $d_x \subset \mathbb{N}^+$ is the dimension of $x$. $u$ is the solution of the PDE and let the dimension of the solution be $d_u \subset \mathbb{N}^+$. $f(x)$ and $g(x)$ are known functions. $\mathcal{L}$, $\mathcal{N}$ and $\mathcal{B}$ are the linear, non-linear and boundary operator of the PDE applied on the solution $u$.

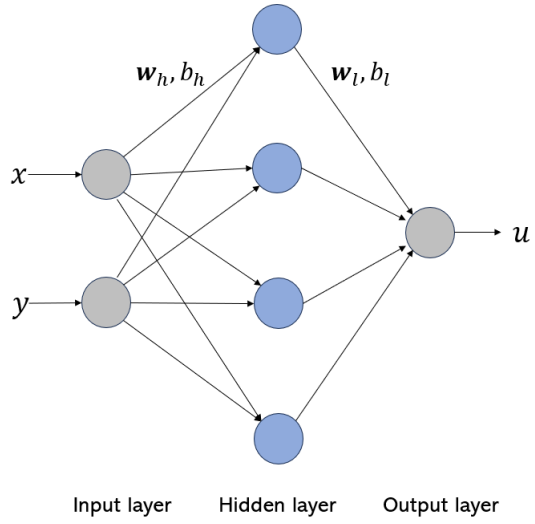



Figure 3.1: Neural network with $2D$ input, $1D$ output and 1 hidden layer

Let us start with an assumption that the solution of the PDE given in eq. 3.1 can be represented by a neural network with one hidden layer, as given in Fig. 3.1. The output

of the neural network is given as

$$u(x) = \sum_{i=0}^{N_u} w_{li}\sigma_i(x) + b_l = \boldsymbol{\sigma(x)} \cdot \boldsymbol{W_l} + b_l, \tag{3.2}$$

$$\boldsymbol{\sigma(x)} = (\sigma_1(x), \sigma_2(x), \ldots, \sigma_{N_u}(x)), \tag{3.3}$$

where $N_u$ is the number of neurons in the hidden layer and $\sigma(x)$ is some activation function $\sigma$ applied on the output from the hidden layer of the neural network and $\boldsymbol{W_l}$ and $b_l$ are the parameters of the linear layer. The concept introduced in [DL21] to obtain the solution of a PDE is that the hidden layer parameters are not trained in the solution process, which means they are fixed to the initial value with which they are initialized to and only linear layer parameters are updated. These parameters can be initialized randomly based on the distribution of choice or set intelligently using the SWIM method; more about the initialization of network parameters is discussed in section 3.1.3. Since the network output satisfies the PDE (initial assumption), we can substitute the output with eq. 3.1 to get the following equations

$$\begin{aligned} \mathcal{L}(\boldsymbol{\sigma(x)} \cdot \boldsymbol{W_l} + b_l) + \mathcal{N}(\boldsymbol{\sigma(x)} \cdot \boldsymbol{W_l} + b_l) = f(x) \quad x \in \Omega, \\ \mathcal{B}(\boldsymbol{\sigma(x)} \cdot \boldsymbol{W_l} + b_l) = g(x) \quad x \in \partial\Omega. \end{aligned} \tag{3.4}$$

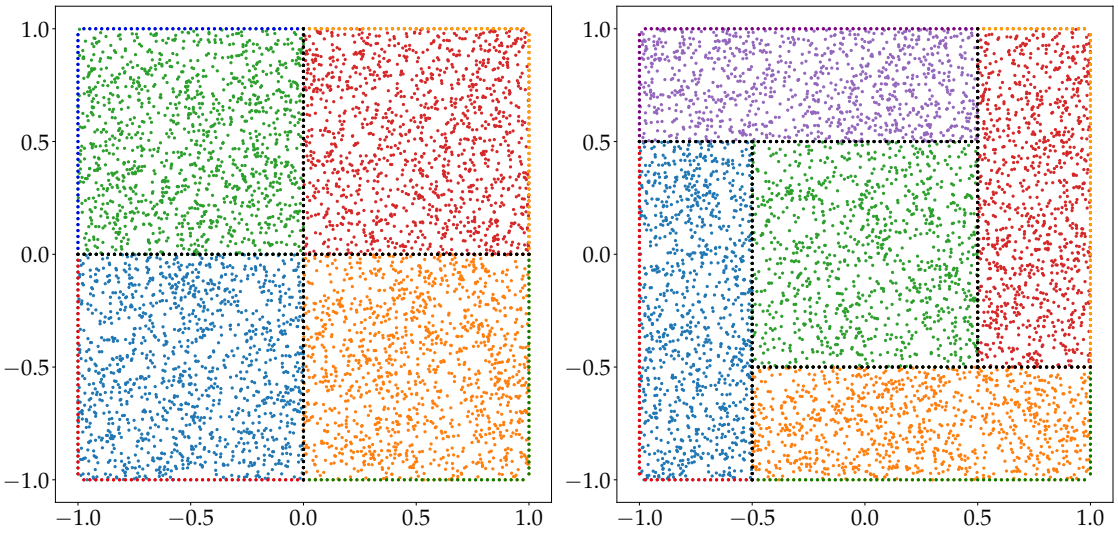
In the next sections, we will look into additional conditions imposed due to the introduction of domain decomposition, preparation of input to the neural network, initializing the neural network parameters using the SWIM method, and finally, the loss function and optimization process to obtain the solution of the PDE.

### 3.1.1 Domain decomposition

Domain decomposition proves to be a helpful strategy in the scientific realm for solving PDEs in a computationally efficient manner. This method involves partitioning the overall computational domain into smaller, more manageable subdomains, treating each as an independent unit. It involves dividing a complex computational domain into smaller subdomains, each of which can be solved independently or semi-independently. The primary goal is to distribute the computational workload across multiple processors or compute nodes, leveraging parallelism to accelerate the solution process.

One notable aspect of domain decomposition, particularly in neural networks, is the ability to approximate relatively simple solutions within each subdomain. By decomposing the problem into smaller units, the inherent complexities of PDEs in each subdomain become more manageable for neural networks. The localized nature of subdomain problems allows for utilizing neural network architectures to efficiently capture and approximate the underlying solution within each distinct region [Hei+21]. Modifications have been proposed in PINN that combine domain decomposition with

it called Extended Physics-Informed Neural Network (XPINN) [DE20]. The continuity conditions are enforced by adding a loss term that satisfies the solution continuity at the shared boundaries of adjacent subdomains. The main downside of this approach is that the overall solution can still contain discontinuities as the continuity constraints are weakly satisfied in the loss term. An improvement was proposed by [MMN21] in which overlapping subdomains resulted in strict enforcing of continuity conditions on the subdomain interface. A similar trend is observed in [DL21] and [Che+22], where the former introduced the locELM method that uses continuity conditions on the interface of subdomains. In contrast, the latter introduced partition of unity or overlapping subdomains to enforce continuity of the solution across subdomains.




(a) Grid domain decomposition

(b) Arbitrary domain decomposition

Figure 3.2: Two types of domain decomposition

In the case of domain decomposition, the domain is divided into multiple subdomains. For each subdomain, the solution is obtained by an individual neural network. These different networks' solutions are stitched together to get the solution over the entire domain. To ensure the overall solution continuity over the domain, extra constraints are imposed on the boundaries of the neighboring subdomains. The order of the PDE constraints the derivatives of that order. Hence, the order of the continuity constraints is one order less than the order of the PDE solved in 3.1 to maintain the continuity of the solution as well as the lower derivatives of the solution. This is done by first sampling points on the shared boundary between two adjacent subdomains. These "shared points" are different from the internal and boundary points we sample in the case of no domain decomposition. The extra constraint imposed in addition to the equations given in 3.1 are

$$\mathcal{C}^m u_j(z) - \mathcal{C}^m u_k(z) = 0 \quad z \in \partial\Omega_{shared}, \tag{3.5}$$

where $\mathcal{C}^m$ are the continuity conditions starting from $m = 0$ to one order less than the PDE, $u_j(x)$ and $u_k(x)$ are the solutions of the PDEs in adjacent subdomains $j$ and $k$ on the shared boundary $\partial\Omega_{shared}$. Substituting the output of the neural network given in eq. 3.2 in eq. 3.5 we get,

$$\mathcal{C}^m(\sigma_j(\boldsymbol{z}) \cdot \boldsymbol{W_{lj}} + b_{lj}) - \mathcal{C}^m(\sigma_k(\boldsymbol{z}) \cdot \boldsymbol{W_{lk}} + b_{lk}) = 0 \quad z \in \partial\Omega_{shared}. \tag{3.6}$$

In the thesis, two distinct domain decomposition strategies are implemented in 2D: grid-based and arbitrary decomposition approaches.

In grid-like domain decomposition, the number of partitions in the x and y directions is used as an input in the 2D case, resulting in equal domain splits based on the number of partitions in that direction. In arbitrary domain decomposition, the subdomains are defined by coordinates of the vertices of subdomains. In the current work, the limitation in the implementation with the arbitrary domain decomposition method is that the shape of subdomains can only be a rectangle, and the union of all the subdomains should cover the entire domain. The two types of domain decomposition implementation are given in fig.3.2.

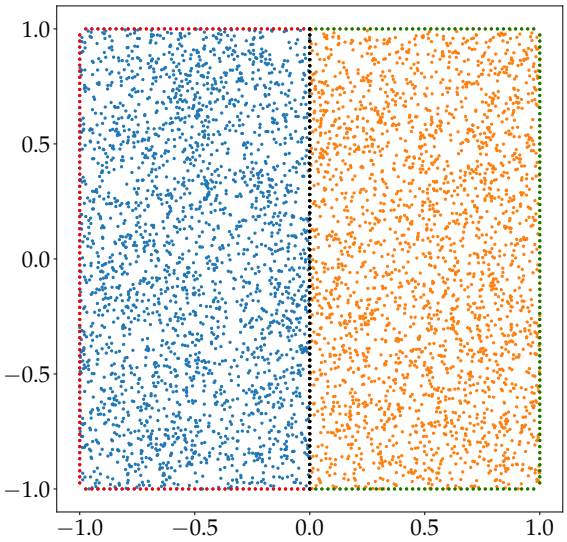



Figure 3.3: Points sampled with domain decomposed into 2 subdomains. For the left subdomain interior points and boundary points are marked in blue and red colors respectively and for the right domain interior and boundary points are marked in orange and green color respectively. The shared points are marked in black color.

### 3.1.2 Input points

The first stage involves systematically sampling points from the interior domain and boundaries. Before using these sampled points as input to the neural network, the coordinates are normalized to $[-1,1]^{\mathcal{R}^d}$. More specifically, the input to the neural network is prepared as follows. Let $N_j^{in}$ be the number of points sampled from the interior of $j^{th}$ subdomain and $N_j^{bc}$ be the number of points sampled from the domain boundary local to the subdomain. For the case with more than one subdomain, we also need to sample points from the shared boundary between adjacent subdomains. Let the number of points sampled from the shared boundary be $N_{jk}^{sh}$ common to the adjacent subdomains $j$ and $k$. An example of points sampling is given in the fig. 3.3.

We set the points for $j^{th}$ subdomain as follows:[1]

$$P_j^{in} = [x_1, x_2, \ldots, x_{N_j^{in}}]^T, \tag{3.7}$$

$$P_j^{bc} = [y_1, y_2, \ldots, y_{N_j^{bc}}]^T, \tag{3.8}$$

$$P_{jk}^{sh} = [z_1, z_2, \ldots, z_{N_{jk}^{sh}}]^T, \tag{3.9}$$

where $x \in [a_j, b_j]^d$ represents points sampled randomly from a uniform distribution from the interior domain with $a_j$ and $b_j$ being the lower and upper bounds of the $j^{th}$ subdomain, $y$ represents points sampled from the domain boundary and $z$ represents points sampled from the shared boundary. Let the center of $j^{th}$ subdomain be $x_n$ and $r_n$ so that the domain's bounds can be represented as $[x_n - r_n, x_n + r_n]^d$. The normalized coordinates then can be given as:

$$(P_j^{in})_{norm} = \frac{1}{r_n}(P_x^{in} - x_n), \tag{3.10}$$

$$(P_j^{bc})_{norm} = \frac{1}{r_n}(P_j^{bc} - x_n), \tag{3.11}$$

$$(P_{jk}^{sh})_{norm} = \frac{1}{r_n}(P_{jk}^{sh} - x_n). \tag{3.12}$$

The normalization procedure maps the coordinates from $[x_n - r_n, x_n + r_n]^d$ to $[-1,1]^d$, which then is used as an input to the neural network.

Normalizing the input can enhance the results along with domain decomposition if random initialization is used for the neural network. Normalizing the inputs for each subdomain, we perform an affine transformation that maps each smaller partition onto the domain $[-1,1]^d$, making it symmetric and better suited for random initialization. In the next section, we discuss the initialization of a neural network using the SWIM method.

[1] Shared points are only sampled if there are more than 1 subdomains

### 3.1.3 Sampling Where It Matters initialization

The SWIM method demonstrates effectiveness when ground truth label data is accessible; however, in our scenario of employing neural networks to solve the PDE, the solution remains unknown, posing a challenge in obtaining prior knowledge of the PDE solution. In this case, when using In the SWIM algorithm, the activation function can be placed uniformly and not based on the gradient of the true label. The advantage this still has over random initialization is that the activation functions are still placed in between the pair of input points, which results in the shift of activation function still within the domain compared to random initialization, where some activation functions whose shift might be far away from the domain of interest and may not help in approximating the solution of the PDE. Another advantage is the effectiveness of SWIM initialization does not depend on the exact position of the domain in the input space, whereas for random initialization, since the weights and biases are sampled from a normal and uniform distribution respectively, it might have a bias of performing better on certain domain bounds.

For a particular case of PDEs where the right-hand side of the PDE gives a good hint of the solution of PDE; for example, in the Poisson equation, we use the right-hand side function to place the activation functions within the domain. An obvious advantage it presents compared to uniformly placing activation functions is if the right-hand side function of the PDE gives a good hint of the solution of PDE; the SWIM method will place more activation functions in the region where the solution also might have a high gradient magnitude.

### 3.1.4 Loss function and optimization

In neural networks, the loss function is a critical metric for quantifying the disparity between model predictions and actual target values. The loss function describes the model's performance by assigning a penalty based on the deviation between predicted and true outcomes. Common loss functions include mean squared error (MSE) for regression tasks and categorical cross-entropy for classification problems. During training, the model iteratively adjusts its parameters to minimize this loss, effectively enhancing its predictive accuracy and generalization capabilities.

The loss function in problems where neural networks are used to obtain the solution of PDE is generated by evaluating the PDE, boundary conditions, and shared conditions

on collocation points. The loss function is given as

$$Loss = \sum_{j=1}^{N_{sub}} (\sum_{i=1}^{N_j^{in}} \lambda_{in} |\mathcal{L}u_j(x_i) + \mathcal{N}u_j(x_i) - f_j(x)|^2 + \sum_{i=1}^{N_j^{bc}} \lambda_b |\mathcal{B}u_j(y_i) - g_j(y_i)|^2 + \sum_{i=1}^{N_{jk}^{sh}} \lambda_{sh} |\mathcal{C}^m u_j(z_i) - \mathcal{C}^m u_k(z_i)|^2). \quad (3.13)$$

In the case of a single domain, the loss function will only have contributions from internal and boundary points. In the above equation, $\lambda_{in}$, $\lambda_b$, and $\lambda_{sh}$ are parameters multiplied to each term of the loss function to balance the contribution from the different type of points. We describe the process to set these parameters later in this section.

Since we do not train the hidden layer parameters but just the linear layer parameters, the complete problem can be defined by a linear or non-linear system of equations for linear or non-linear PDEs, respectively.

From eq. 3.4 and eq. 3.6 we can construct the following system of equations:

$$\mathcal{R} = \begin{bmatrix} \lambda_{in}(\mathcal{L}(\boldsymbol{\sigma_j(x)} \cdot \boldsymbol{W_{lj}} + b_{lj}) + \mathcal{N}(\boldsymbol{\sigma_j(x)} \cdot \boldsymbol{W_{lj}} + b_{lj})), 1 \leq j \leq N_{sub} \\ \lambda_b(\mathcal{B}(\boldsymbol{\sigma_j(y)} \cdot \boldsymbol{W_{lj}} + b_{lj})), 1 \leq j \leq N_{sub} \\ \lambda_{sh}(\mathcal{C}^m(\boldsymbol{\sigma_j(z)} \cdot \boldsymbol{W_{lj}} + b_{lj}) - \mathcal{C}^m(\boldsymbol{\sigma_k(z)} \cdot \boldsymbol{W_{lk}} + b_{lk})), 1 \leq j < k \leq N_{sub} \end{bmatrix} - \begin{bmatrix} f(x) \\ g(y) \\ 0 \end{bmatrix}, \quad (3.14)$$

where the least squares problem is to minimize $0.5\|\mathcal{R}\|^2$.

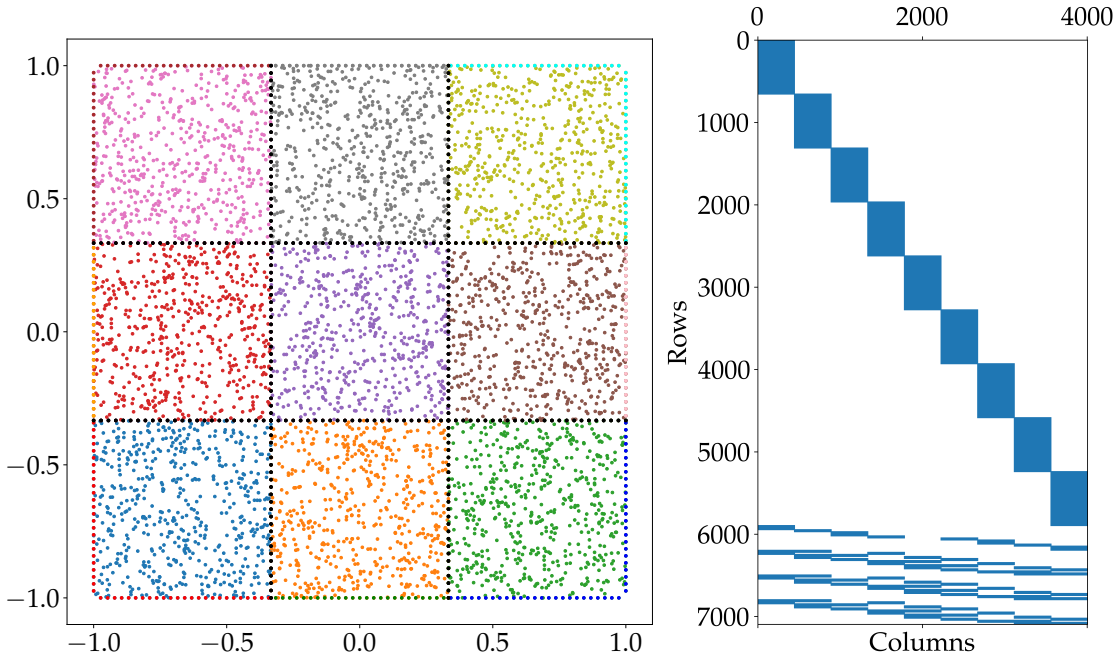
To simplify the equations we assume only 2 subdomains from now. For the case where the pde does not have a non-linear part or $\mathcal{N} = 0$ we can rewrite eq. 3.14 as follows

$$\mathcal{R} = \begin{bmatrix} \lambda_{in}(\mathcal{L}(\boldsymbol{\sigma_1(x)} \cdot \boldsymbol{W_{l1}} + b_{l1})) & 0 \\ 0 & \lambda_{in}(\mathcal{L}(\boldsymbol{\sigma_2(x)} \cdot \boldsymbol{W_{l2}} + b_{l2})) \\ \lambda_b(\mathcal{B}(\boldsymbol{\sigma_1(y)} \cdot \boldsymbol{W_{l1}} + b_{l1})) & 0 \\ 0 & \lambda_b(\mathcal{B}(\boldsymbol{\sigma_2(y)} \cdot \boldsymbol{W_{l2}} + b_{l2})) \\ \lambda_{sh}(\mathcal{C}^m(\boldsymbol{\sigma_1(z)} \cdot \boldsymbol{W_{l1}} + b_{l1})) & -\lambda_{sh}(\mathcal{C}^m(\boldsymbol{\sigma_2(z)} \cdot \boldsymbol{W_{l2}} + b_{l2})) \end{bmatrix} - \begin{bmatrix} f_1(x) \\ f_2(x) \\ g_1(y) \\ g_2(y) \\ 0 \end{bmatrix}. \quad (3.15)$$

Since $\mathcal{L}$, $\mathcal{B}$ and $\mathcal{C}$ are linear operators applied on the solution. The residual equation 3.15 can be written as a linear system of equations as follows

$$\begin{bmatrix} \lambda_{in}\mathcal{L}(\boldsymbol{\sigma_1(x)}) & \lambda_{in}\mathcal{L}(1) & 0 & 0 \\ 0 & 0 & \lambda_{in}\mathcal{L}(\boldsymbol{\sigma_2(x)}) & \lambda_{in}\mathcal{L}(1) \\ \lambda_b\mathcal{B}(\boldsymbol{\sigma_1(y)}) & \lambda_b\mathcal{B}(1) & 0 & 0 \\ 0 & 0 & \lambda_b\mathcal{B}(\boldsymbol{\sigma_2(y)}) & \lambda_b\mathcal{B}(1) \\ \lambda_{sh}\mathcal{C}^m(\boldsymbol{\sigma_1(z)}) & \lambda_{sh}\mathcal{C}^m(1) & -\lambda_{sh}\mathcal{C}^m(\boldsymbol{\sigma_2(z)}) & -\lambda_{sh}\mathcal{C}^m(1) \end{bmatrix} \begin{bmatrix} \boldsymbol{W_{l1}} \\ b_{l1} \\ \boldsymbol{W_{l2}} \\ b_{l2} \end{bmatrix} = \begin{bmatrix} f_1(x) \\ f_2(x) \\ g_1(y) \\ g_2(y) \\ 0 \end{bmatrix}. \quad (3.16)$$

This linear system of equations can be solved for the parameters of the linear layer using SciPy's [Vir+20] linear least squares solver: *scipy.optimize.lsq_linear* [BCL99]. For a linear system of equations, the parameters $\lambda_{in}$ and $\lambda_b$ are set to normalize the maximum of rows of the coefficient matrix to 1. This strategy of balancing loss terms is given in [Che+22], and from our experiments, we can conclude that it is the best strategy among the few we tried. The results for the same can be seen in 3.2.1.2.




(a) Points sampled in a 9 subdomain grid like domain decomposition split

(b) Resulting sparsity pattern of the coefficient matrix of linear system of equations

Figure 3.4: Visualization of points sampled in the case of 9 subdomains and resulting sparsity pattern of coefficient matrix in the linear system of equations.

For more than two subdomains, the system of equations can be written in a similar manner to the one given in eq. 3.16 and can be solved using a linear least squares solver. In fig.3.4, we can observe the points sampled from the domain divided into nine subdomains and the sparsity pattern of the resulting matrix of the coefficient matrix.

For the case where $\mathcal{N} \neq 0$, the system of equations has to be solved using SciPy's non-linear least squares solver. In the current work, these equations are solved using a non-linear least squares solver: *scipy.optimize.least_squares* [Mor78]. The solver computes the Jacobian matrix of the residual with respect to the parameters solved for using a finite difference scheme. The computational speed can be improved significantly by providing the Jacobian matrix. If the linear layer parameters can be expressed as

$\phi = (\boldsymbol{W}_l, b_l)$. The Jacobian matrix is computed by taking the partial derivative of residual with respect to the linear layer parameters

$$\mathcal{J} = \frac{\partial \mathcal{R}}{\partial \phi}. \tag{3.17}$$

### 3.1.5 Summary

In this section, we have discussed the entire process of obtaining the solution for the PDE. We discuss several aspects of the solution-obtaining process, like sampling input points, preparing input for the network, and modifications in case of domain decomposition. Finally, we mention the loss function and optimization process to train the network parameters. In this section, we summarize the entire process. The solution of the PDE is obtained using neural networks with domain decomposition by following these steps:

1. Sample points from the interior, boundary, and shared boundaries between subdomains in the case of domain decomposition and normalize the coordinates to $[-1, 1]^d$. This will be the input to the neural network.
2. Fit the network using SWIM method.
3. Construct the loss function based on the PDE and boundary conditions and in case of domain decomposition add the continuity constraints on the points sampled from the shared boundaries.
4. In case of:
    - Linear PDEs solve the resulting linear system of equations using a linear least squares solver.
    - Non-linear PDEs compute the residual vector and Jacobian of it with respect to linear layer parameters and solve the resulting non-linear system of equations using a non-linear least squares solver.

## 3.2 Experiments

### 3.2.1 Linear Partial Differential Equation

#### 3.2.1.1 Problem Description

The Poisson equation is a fundamental PDE widely employed in various scientific disciplines, particularly physics, engineering, and mathematics. It describes the distribution of a scalar field, such as temperature, pressure, or electrostatic potential, within a given domain. Mathematically, the Poisson equation is expressed as:

$$\Delta u = f(x_1, x_2, \ldots, x_d), \tag{3.18}$$

where $\Delta$ represents the Laplacian operator, $u$ is the scalar field of interest, and $f$ is a given function of independent variables $x_1, x_2, \ldots, x_d$. This equation can be generalized to any dimension $d$. The results in the following subsections are compared for four different solution functions of the Poisson equation. The different Poisson equations and its solution is given as follows for any dimensions $d$:

- Decaying sin

$$\Delta u = \sum_{i=1}^{d} -e^{-1.5x_i}\left(222.75\sin(15x_i) + 45\cos(15x_i)\right) \tag{3.19}$$
$$u = \sum_{i=1}^{d} \sin(15x_i)e^{-1.5x_i}$$

- Sum of sin functions with two frequencies

$$\Delta u = \sum_{i=1}^{d} -4\sin(2x_i) - 100\sin(10x_i) \tag{3.20}$$
$$u = \sum_{i=1}^{d} \sin(2x_i) + \sin(10x_i)$$

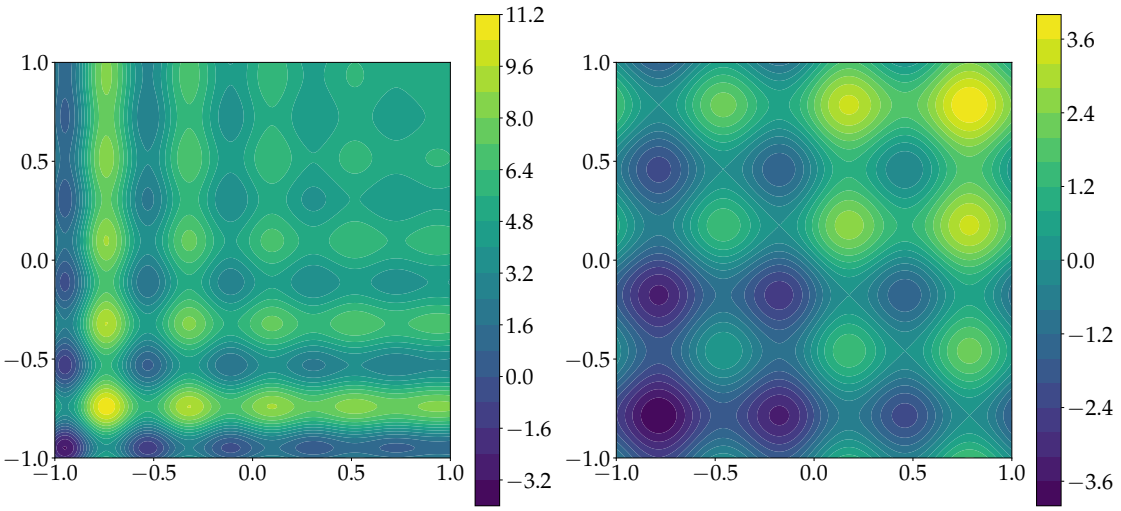
- Gaussian function

$$\Delta u = \sum_{i=1}^{d} (400x_i^2 - 20)\exp\left(-10\sum_{i=1}^{d} x_i^2\right) \tag{3.21}$$
$$u = \exp\left(-10\sum_{i=1}^{d} x_i^2\right)$$

- Polynomial (5th order)

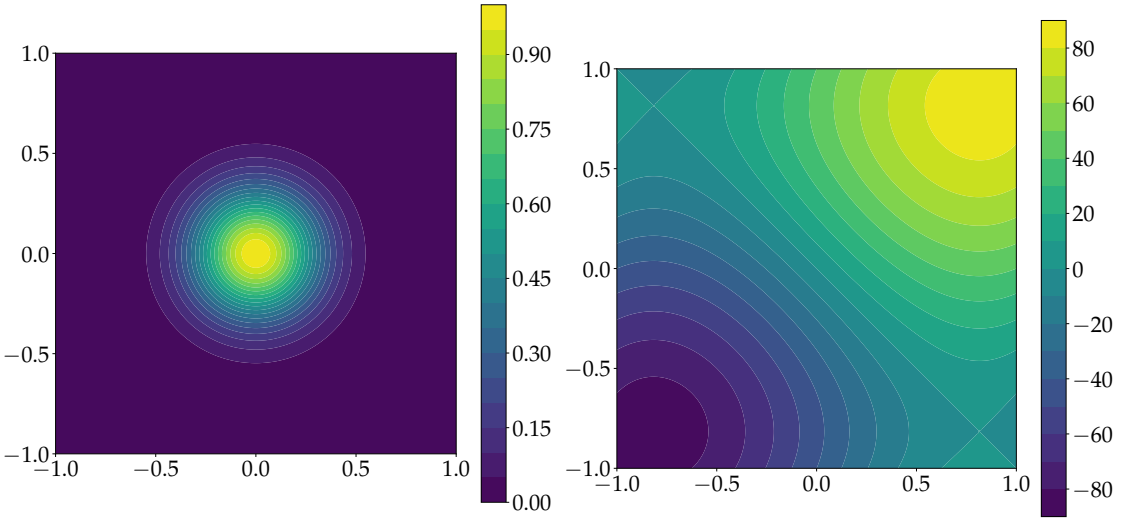
$$\Delta u = 80(\sum_{i=1}^{d} x_i^3) - 270(\sum_{i=1}^{d} x_i) \tag{3.22}$$
$$u = 4(\sum_{i=1}^{d} x_i^5) - 45(\sum_{i=1}^{d} x_i^3) + 81(\sum_{i=1}^{d} x_i)$$

The boundary condition for the above-mentioned equations is given by evaluating the true solution at the boundary points. In this section, for all the experiments we use the right-hand side of the Poisson equation to initialize the network parameters and always prune the duplicate activation functions in the SWIM method.




(a) Decaying sin

(b) Sum of sin functions with two frequencies




(c) Gaussian function

(d) $5^{th}$ order polynomial

Figure 3.5: Solution of PDEs used for experiments over a 2D domain $[-1, 1]^2$

#### 3.2.1.2 Balancing PDE and boundary conditions losses

In this experiment, we present a comparative analysis of the results obtained by balancing losses from internal and boundary points across four different functions, employing various loss-balancing techniques. The primary objective of this experiment was to

mitigate errors in the solution by effectively distributing the loss between internal and boundary points. Notably, all experiments were conducted for a single domain, i.e., no domain decomposition, ensuring a consistent evaluation for all the strategies. The strategies we compare are:

- Default least squares problem.

$$\begin{aligned} \lambda_{in} &= 1, \\ \lambda_b &= 1. \end{aligned} \tag{3.23}$$

- Normalize the coefficient matrix rows by maximum of the rows.

$$\begin{aligned} \lambda_{in} &= \frac{1}{\max\limits_{1 \leq l \leq N_l} (\mathcal{L}(\sigma_l(x_i)), \mathcal{L}(1))}, \\ \lambda_b &= \frac{1}{\max\limits_{1 \leq l \leq N_l} (\mathcal{B}(\sigma_l(y_i)), \mathcal{B}(1))}, \end{aligned} \tag{3.24}$$

  here $[\mathcal{L}(\sigma_l(x_i)), \mathcal{L}(1)]$ represents a row of the coefficient matrix for the internal points $x_i$ and $[\mathcal{B}(\sigma_l(y_i)), \mathcal{B}(1)]$ represents a row for the boundary points $y_i$, these equations for the row are also given in eq. 3.15. $\lambda_{in}$ is computed by taking the maximum of each row of PDE conditions part of the coefficient matrix and is a vector of the dimension $N_{in} \times 1$. Similarly, $\lambda_{bc}$ is computed by taking the maximum of each row of boundary conditions part of the coefficient matrix and is a vector of the dimension $N_{bc} \times 1$.

- Normalize the internal points block and the boundary points block of the matrix by the maximum of blocks respectively

$$\begin{aligned} \lambda_{in} &= \frac{1}{\max\limits_{1 \leq i \leq N_{in}} \max\limits_{1 \leq l \leq N_l} (\mathcal{L}(\sigma_l(x_i)), \mathcal{L}(1))}, \\ \lambda_b &= \frac{1}{\max\limits_{1 \leq i \leq N_{bc}} \max\limits_{1 \leq l \leq N_l} (\mathcal{B}(\sigma_l(x_i)), \mathcal{B}(1))}. \end{aligned} \tag{3.25}$$

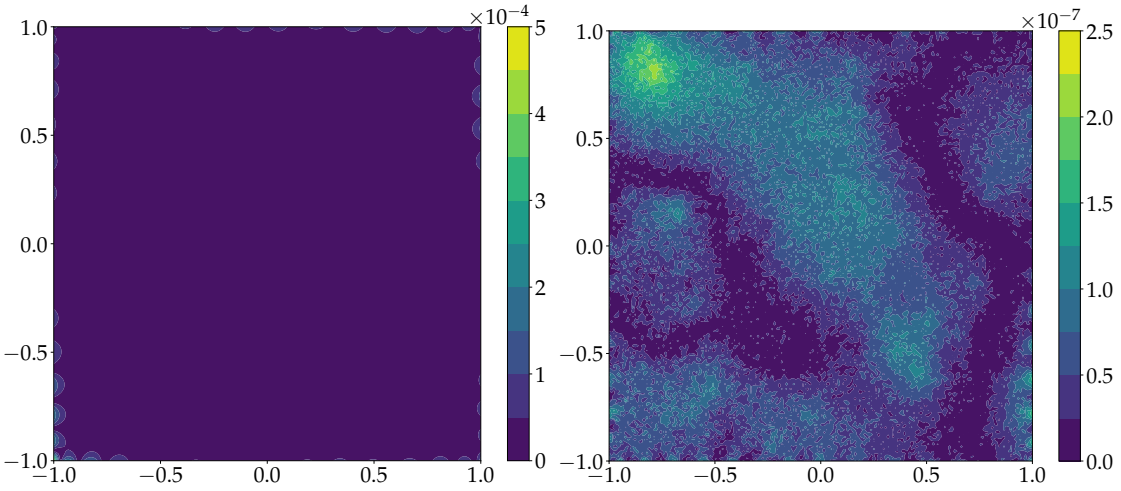
- Scale the PDE and boundary conditions by inverse of points sampled from interior and boundary respectively

$$\begin{aligned} \lambda_{in} &= \frac{100}{N_{in}}, \\ \lambda_b &= \frac{100}{N_{bc}}, \end{aligned} \tag{3.26}$$

  in this strategy $\lambda_{in}$ and $\lambda_b$ are simply computed by scaling the rows by inverse of the number of points sampled in the interior and boundary of the domain respectively.

For this experiment, we investigated a Poisson problem defined over a 2D square domain spanning from [-1, 1] in both dimensions. The various right-hand side functions for the Poisson equation are detailed in section 3.2.1.1. To address boundary conditions, we utilize the solution of the PDE at the boundary points. To investigate the impact of balancing losses from the PDE and boundary conditions on accuracy, we maintain a constant layer width of 6000, and the number of boundary points sampled is also fixed at 500 while varying the number of points sampled from the domain. The results we present are averages obtained from five different random states used for sampling points from the domain. This approach allows us to assess how different configurations of sampled points influence the model's accuracy. For all the next experiments we compute the error on test points sampled from the domain linearly that are different from the points sampled for solving/training.

The results of implementing the strategies above are presented in figs. 3.6, 3.7. From this data, it is evident that normalizing the coefficient matrix rows by the maximum value within each row and scaling the PDE and boundary losses by the inverse of the points sampled from the interior and boundary, respectively, yielded comparable outcomes. Both approaches demonstrated an improvement in the overall accuracy when compared to the default solution. This finding suggests that these normalization and scaling techniques play a crucial role in enhancing the solution's accuracy. Further, comparing the absolute error contours in fig. 3.6 for the default and normalizing rows strategy, it is clear that the error concentrated at the boundaries for the default strategy is observed to be reduced and distributed over the domain for normalizing rows strategy.

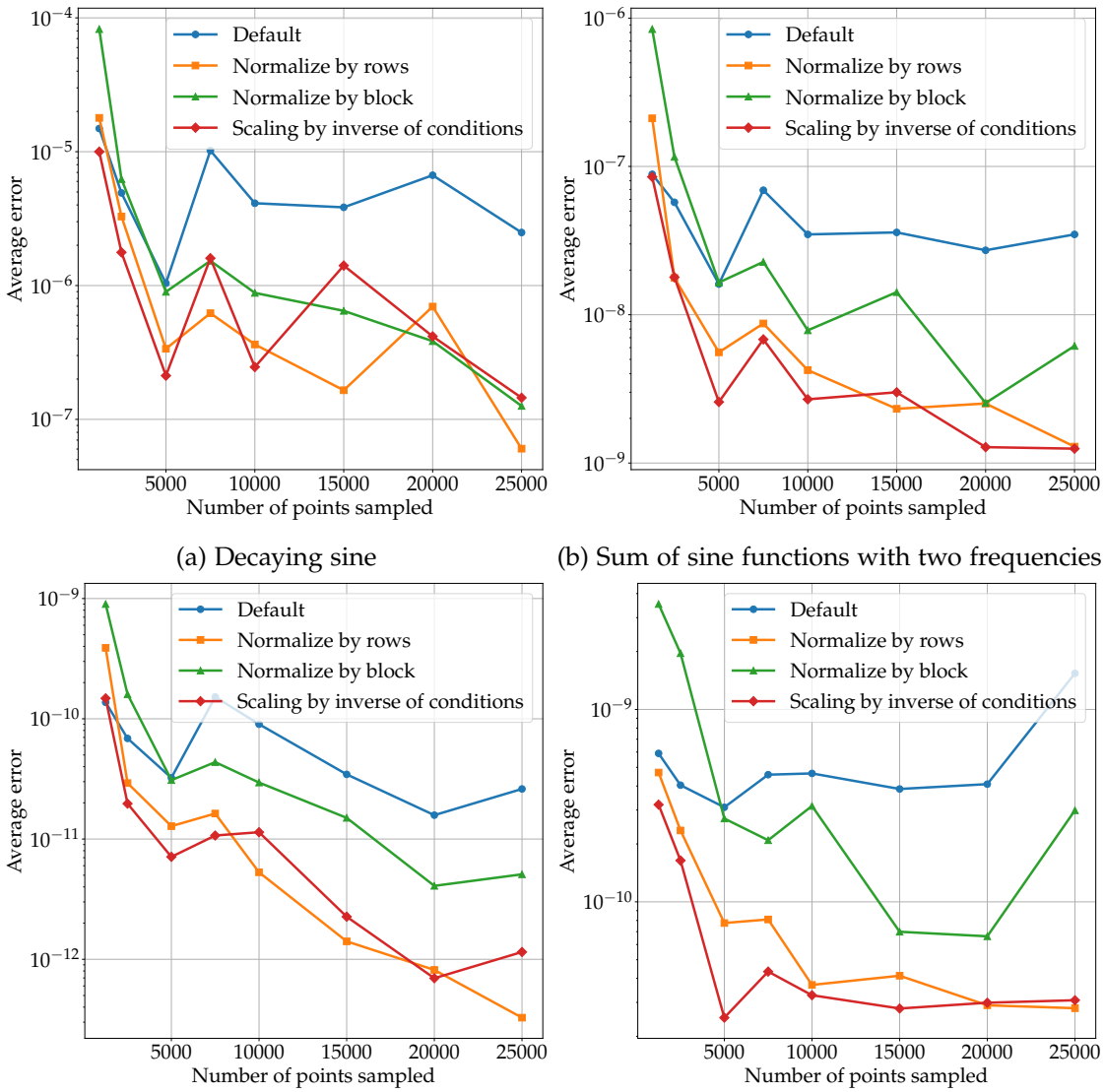



(a) Default least squares problem (b) Normalize rows of coefficient matrix

Figure 3.6: Error contour plots for decaying sine problem

**Conclusion 1** *In [Che+22], the authors suggest normalizing the rows of the coefficient matrix*

*by the maximum value within each row if the constants in the PDE exhibit disparate magnitudes and the results for our experiments align with their conclusion. In all the following experiments, we always normalize the coefficient matrix rows by the maximum value within each row.*




(a) Decaying sine

(b) Sum of sine functions with two frequencies

(c) Gaussian function

(d) $5^{th}$ order polynomial

Figure 3.7: Results for balancing PDE and boundary conditions losses fixing layer width constant at 6000, boundary points sampled at 500 and increasing number of internal points sampled from the domain

#### 3.2.1.3 Ratio of layer width to internal points

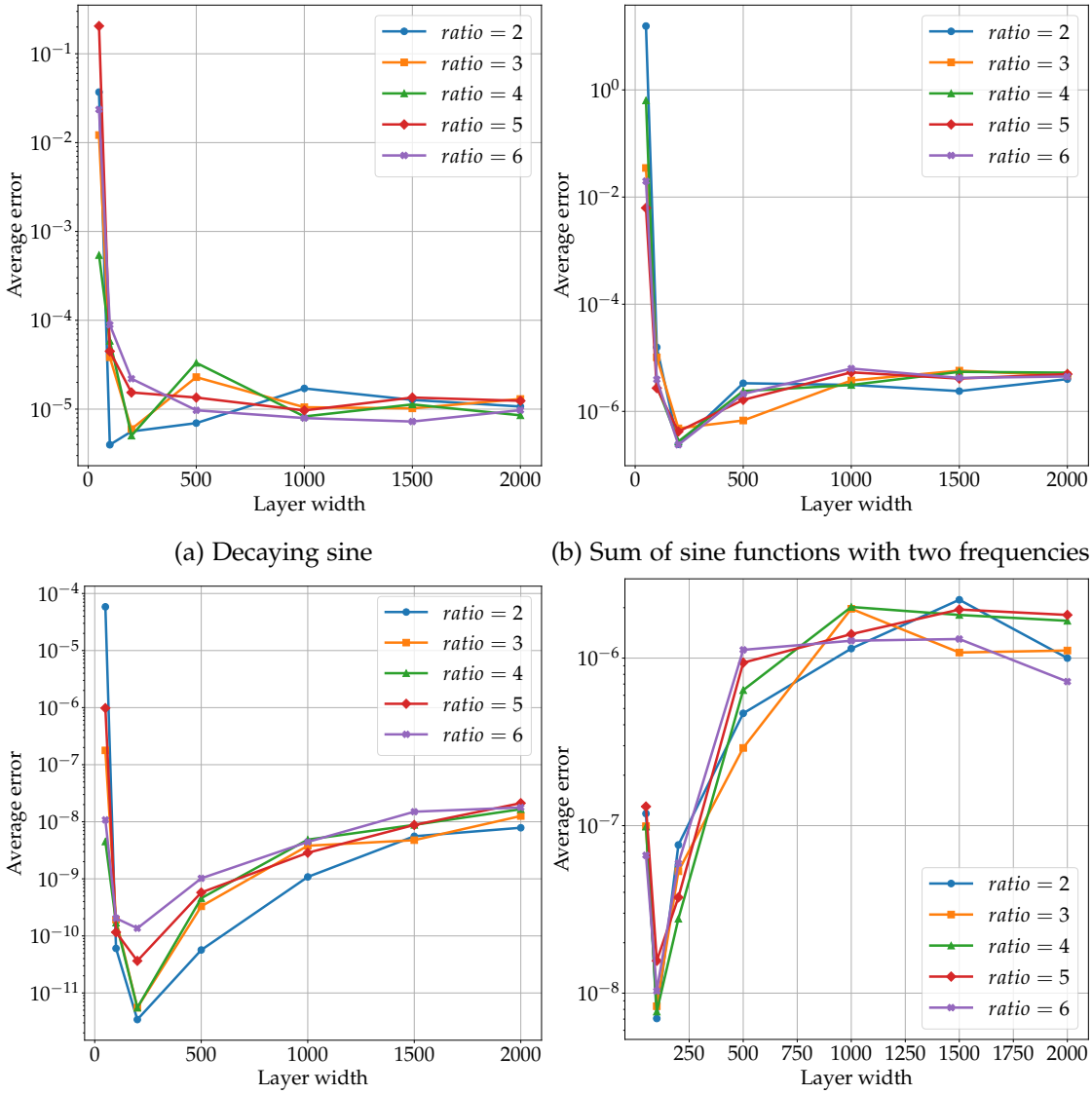
In this section, we run experiments to figure out the best balance between how wide the hidden layer of the neural network should be and how many internal points should be sampled from the domain. We are particularly interested in this because we use the SWIM method to initialize the network weights, which uses a pair of input points to place the activation function for each neuron in the hidden layer. By trying out various ratios of internal points sampled to layer width, we aim to pinpoint the setups that make the network work best. This is important because it helps to understand how to design neural networks that can capture complex patterns effectively while keeping computations manageable.

In this experiment, we investigate various ratios between the number of points sampled and the layer width. For each ratio, we increment the layer width proportionally with the number of points sampled, with the number of points sampled becoming the ratio times the layer width. The boundary points sampled are constant throughout the experiment set at 500. Since only internal points are used to place the activation functions, we fix the number of points sampled from the boundary. The results we present are averages over five different random states used for sampling points from the domain. We compute the average error for different Poisson problems detailed in the section.3.2.1.1. The results are presented for 1D and 2D Poisson problems. By experimenting with these different ratios, we aim to enhance our understanding of how the relationship between point sampling density and layer width impacts the accuracy of our results in solving the Poisson problem.

From the results given in fig.3.8, the network seems to overfit the functions after layer width of 200 and the accuracy for all the ratios is also similar. In fig.3.9, overfitting is not observed and the plots follow the expected trend of increasing ratio resulting in improvement in accuracy, although the improvement is marginal. Following conclusion can be made from this experiment.

**Conclusion 2** *For 1D experiments in fig.3.8 the results suggest that for layer width of 200 for all the functions have the best accuracy for all ratios of internal point to layer width. After layer width of 200 neural network overfits the function and increasing it further results in increasing on test points.*

*For 2D experiments in fig.3.9 the results were as expected with increase in layer width and number of points sampled the accuracy decreases exponentially. In the plots for ratio set at 6, the accuracy was observed to be marginally better than other ratios we tried. Although increasing ratio improved the accuracy marginally the size of system of linear equation increases with the ratio resulting in the time taken to solve to increase with it. For further experiments in 2D we go ahead with ratio set at 5 balancing the accuracy and time taken to obtain the solution.*



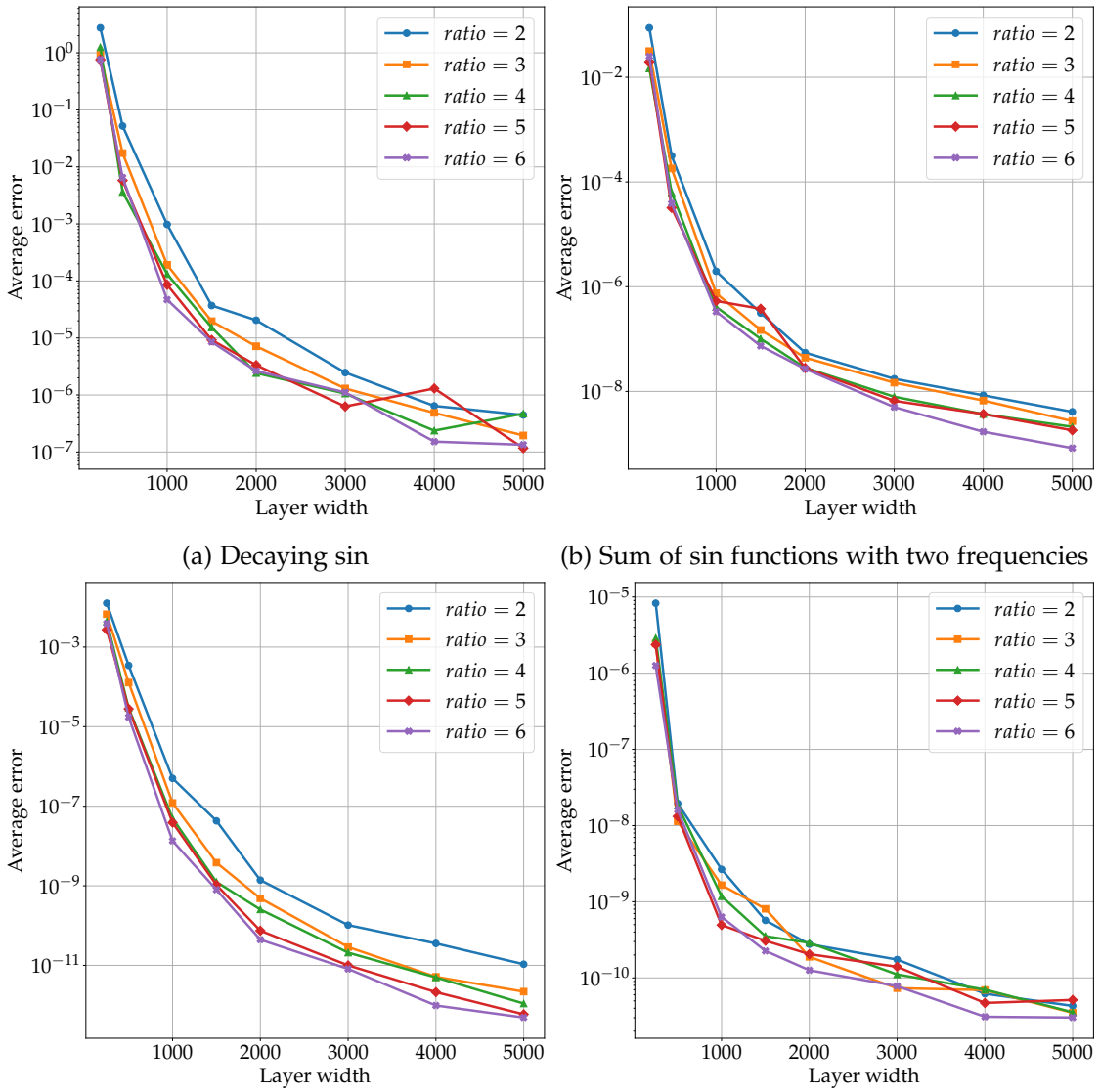

(a) Decaying sine

(b) Sum of sine functions with two frequencies

(c) Gaussian function

(d) $5^{th}$ order polynomial

Figure 3.8: Layer width vs average error plot of five different ratios of internal point to layer width for four functions in $1D$




(a) Decaying sin

(b) Sum of sin functions with two frequencies

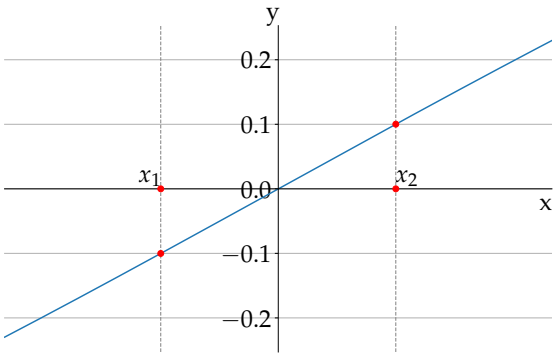
(c) Gaussian function

(d) $5^{th}$ order polynomial

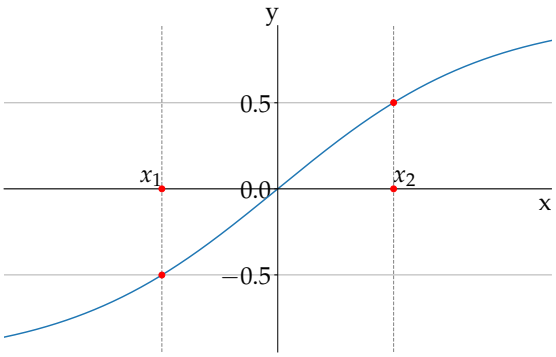
Figure 3.9: Layer width vs average error plot of five different ratios of internal point to layer width for four functions in $2D$

#### 3.2.1.4 Input points mapping in SWIM method

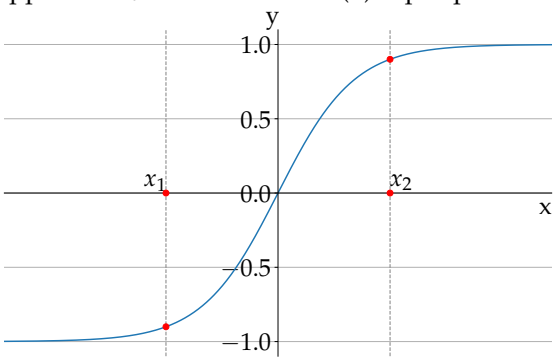
In this experiment, we understand the relation between the hidden layer's weight and the solution's accuracy with the SWIM method. In the SWIM method, the activation function is placed in such a way that the tanh function is exactly between the two input points, and the input points map to $+/-0.5$ in the activation function output. In this experiment, we vary the mapping of the input points from $+/-0.1$ to $+/-0.9$. This affects the steepness of activation functions used as a basis function to approximate the solution. The idea is to approximate a function with high gradients; a steeper activation function will be better, whereas to approximate a smoother function, a less steep activation function should work better. We test if the default $+/-0.5$ mapping in the SWIM method is ideal for the task of approximating the solution of PDEs. In fig.3.10, we can see the different values tested for the activation function output for the pair of input points.




(a) Input points mapped to $+/-0.1$




(b) Input points mapped to $+/-0.5$




(c) Input points mapped to $+/-0.9$

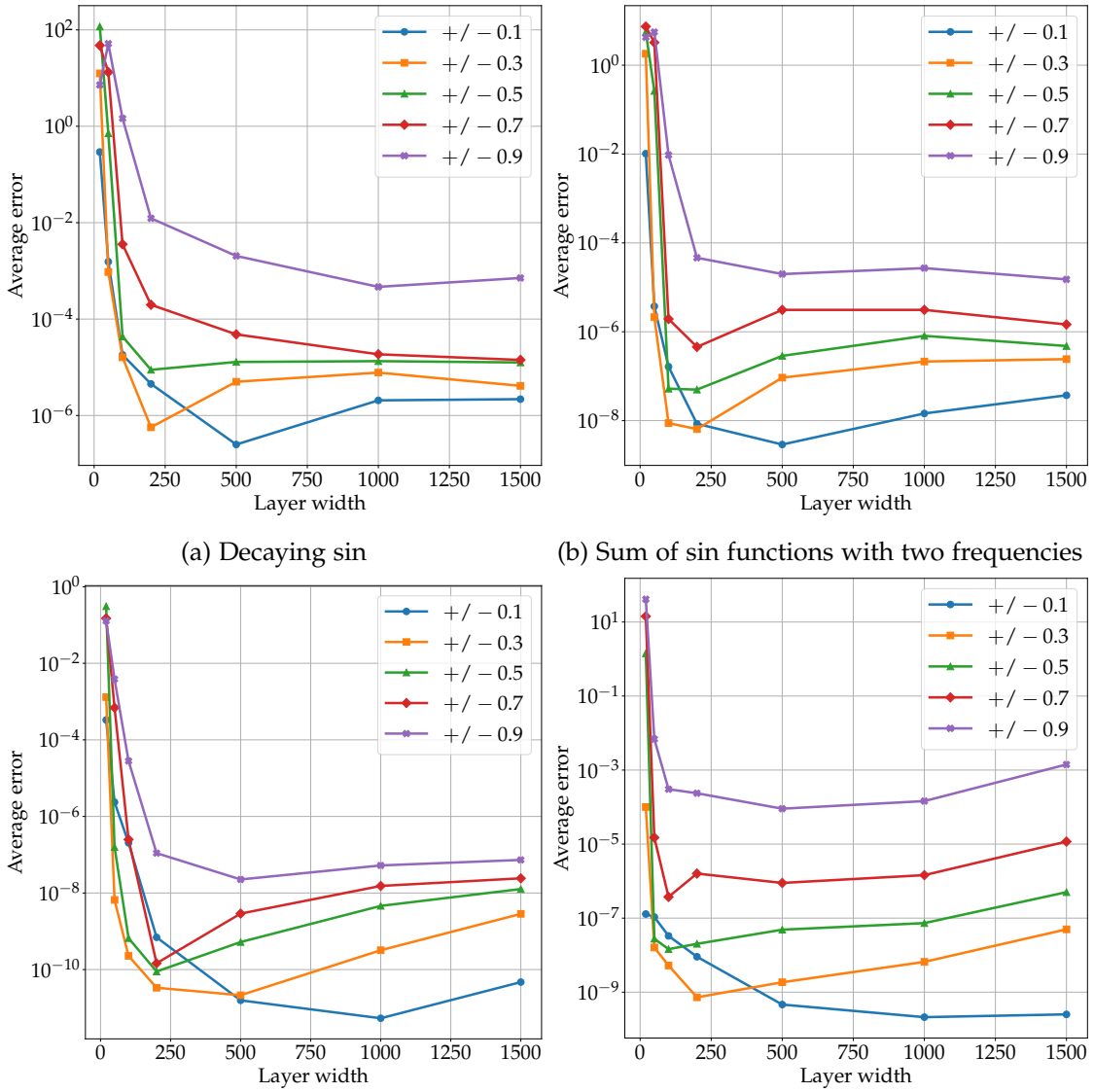
Figure 3.10: Different steepness of tanh activation functions due to input points mapped to different values

For the experiments, we used a domain from $[-1,1]^d$ and 4 Poisson PDEs introduced in 3.2.1.1. For the 1D case, the layer width is increased from 50 to 1500, and for the 2D case, the layer width is increased from 1000 to 5000 with the number of internal

points fixed at five times the layer width for both cases. The number of boundary points sampled in the 2D experiments were fixed at 500. The results we present are averages over five different random states used for sampling points from the domain.

From fig.3.11, we can observe that for smooth functions like Polynomial, a clear improvement in accuracy can be seen with less steep activation function whereas, for other functions in 1D, a marginal improvement is observed for the same.

From fig.3.12, we can observe that the default steepness of the activation function works best for all the true solutions of the PDE tested.



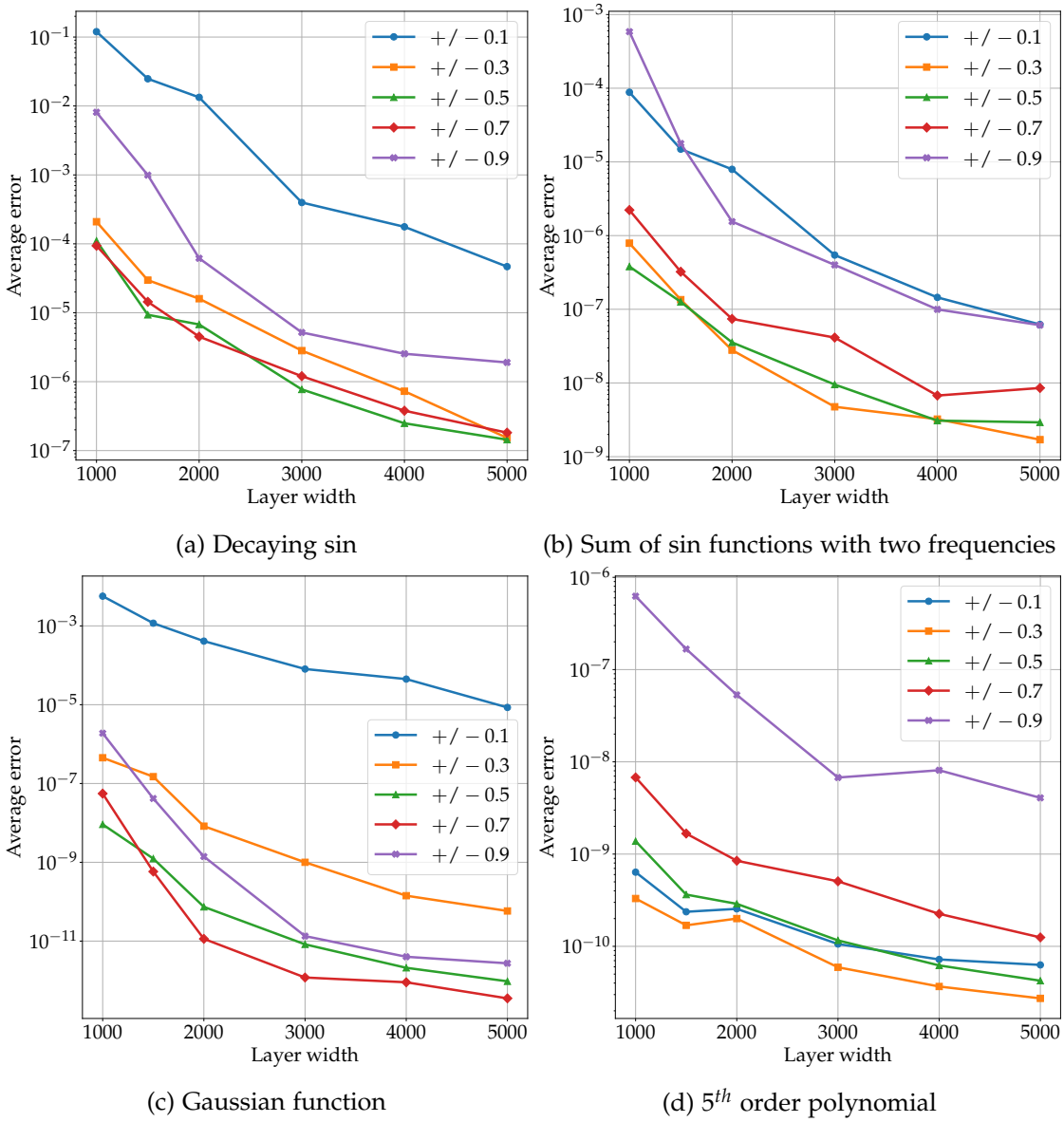

(a) Decaying sin

(b) Sum of sin functions with two frequencies

(c) Gaussian function

(d) $5^{th}$ order polynomial

Figure 3.11: Layer width vs average error plot of 5 steepness value of the tanh activation function for 4 1D Poisson problems




(a) Decaying sin

(b) Sum of sin functions with two frequencies

(c) Gaussian function

(d) $5^{th}$ order polynomial

Figure 3.12: Layer width vs average error plot of 5 steepness value of the tanh activation function for 4 2D Poisson problems

**Conclusion 3** *For 1D experiments the results given in fig.3.11 suggest that less steep activation functions work well for all the functions approximated.*

*For 2D experiments the results given in fig.3.12 suggest that the default steepness of activation function performs well compared other steepness tested.*

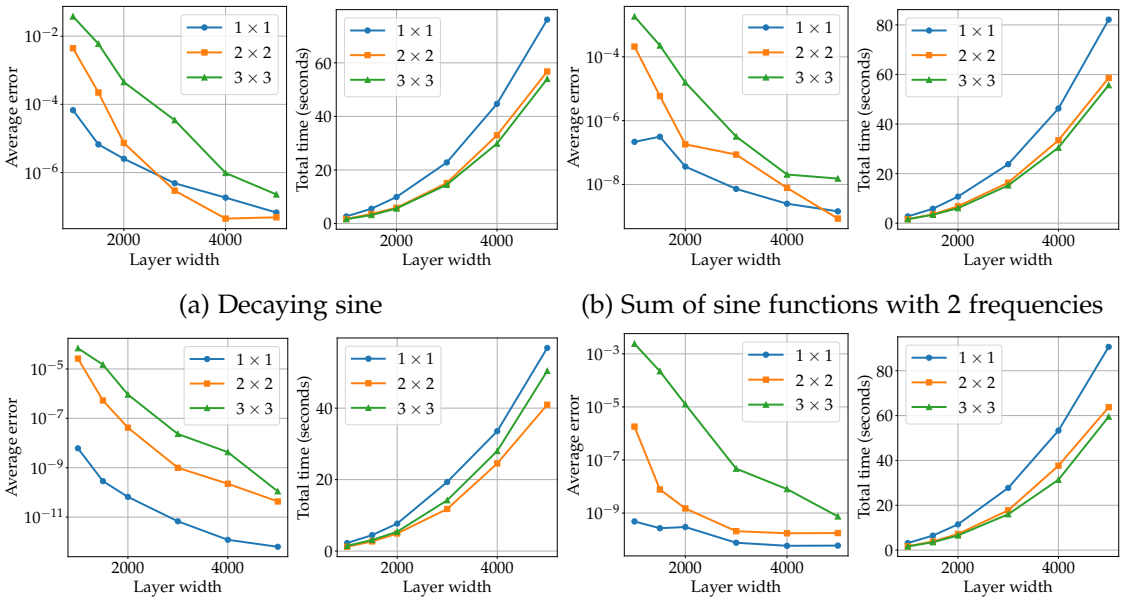
*For further 1D and 2D experiments the output of activation function at the pair of input points is set to be +/-0.1 and +/-0.5 respectively.*

#### 3.2.1.5 Domain decomposition

In this experiment, we present the results of using domain decomposition on accuracy and time taken to obtain the solution of the Poisson equation. The domain size is fixed to $[-1, 1]^2$ for the 2D case, and three grid domain decomposition configurations are compared, i.e., one subdomain ($1 \times 1$), four subdomains ($2 \times 2$) and nine subdomains ($3 \times 3$). The PDEs solved are the same as in previous sections and are given in 3.2.1.1. The layer width is increased from 1000 to 5000, and the number of points sampled is fixed to 5 times the layer width. Number of boundary points sampled for all the cases is fixed at 500 points. For the comparison between 1 domain case and domain decomposition, the total layer width and number of points sampled from the domain are kept constant. This means increasing the number of subdomains results in decreasing the number of points sampled and layer width per subdomain.

Some of the key observations from the results presented in fig.3.13 are for functions that do not have a lot of local variations, for example, Polynomial and Gaussian functions using only one subdomain gives the best accuracy for all the layer width tested, whereas for functions with local variations increasing the number of subdomains resulted in the accuracy being similar to 1 subdomain case for higher layer width, but with domain decomposition, the time taken to solve the system is less compared to 1 domain solution. This is due to the SWIM method's quick sampling of the weights for each local neural network. Also, constructing a linear system of equations is done comparatively faster for increasing the number of subdomains. This can be observed in fig. 3.14 where the breakdown of the total time is presented into different components of the solution-obtaining procedure. The time required to solve the linear system of equations is consistent for all the domain decomposition configurations as the size of the least squares system being solved is marginally increased with an increasing number of subdomains as additional continuity constraints have to be satisfied.

An essential consideration in our experiments is choosing the dense least squares solver over the sparse least squares solver. As outlined in section 3.1, we discussed the sparsity of the coefficient matrix resulting from domain decomposition. Utilizing the sparse least squares solver led to degraded accuracy and increased computational time. This can be attributed to the resulting system's ill-conditioning, a challenge not effectively addressed by the sparse solver [GD21]. Moving forward, exploring preconditioners tailored for handling sparse ill-conditioned systems of linear equations present a promising avenue for future research and optimization.




(a) Decaying sine

(b) Sum of sine functions with 2 frequencies

(c) Gaussian

(d) $5^{th}$ order polynomial

Figure 3.13: Layer width vs average error and layer width vs time taken to solve for 4 different functions with 3 domain decomposition configuration

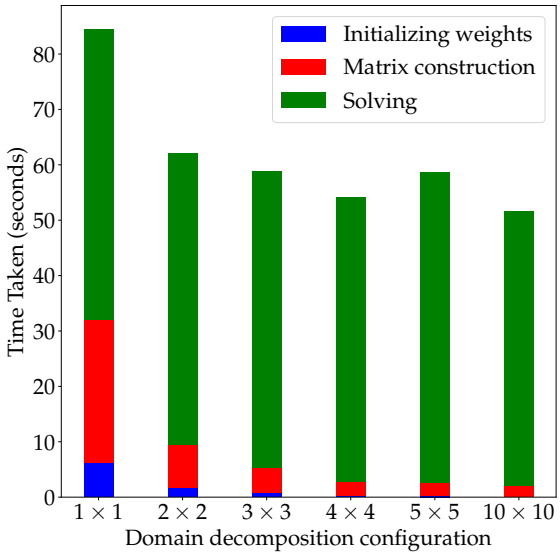



Figure 3.14: Distribution of time taken to obtain solution of PDE with different domain decomposition configurations for $5^{th}$ order polynomial function with layer width set at 5000 and number of points sampled from interior and boundary set at 25000 and 500 respectively.

**Conclusion 4** *From fig. 3.13 and fig. 3.14 we can conclude that fixing the total number of nodes in the hidden layer and points sampled from the domain with domain decomposition, the accuracy remains The same is true for larger layer widths, whereas the time taken to obtain the solution is reduced for a* $2 \times 2$ *split. Further increasing the number of subdomains reduce the accuracy, whereas the time taken to solve is comparable with a* $2 \times 2$ *split.*

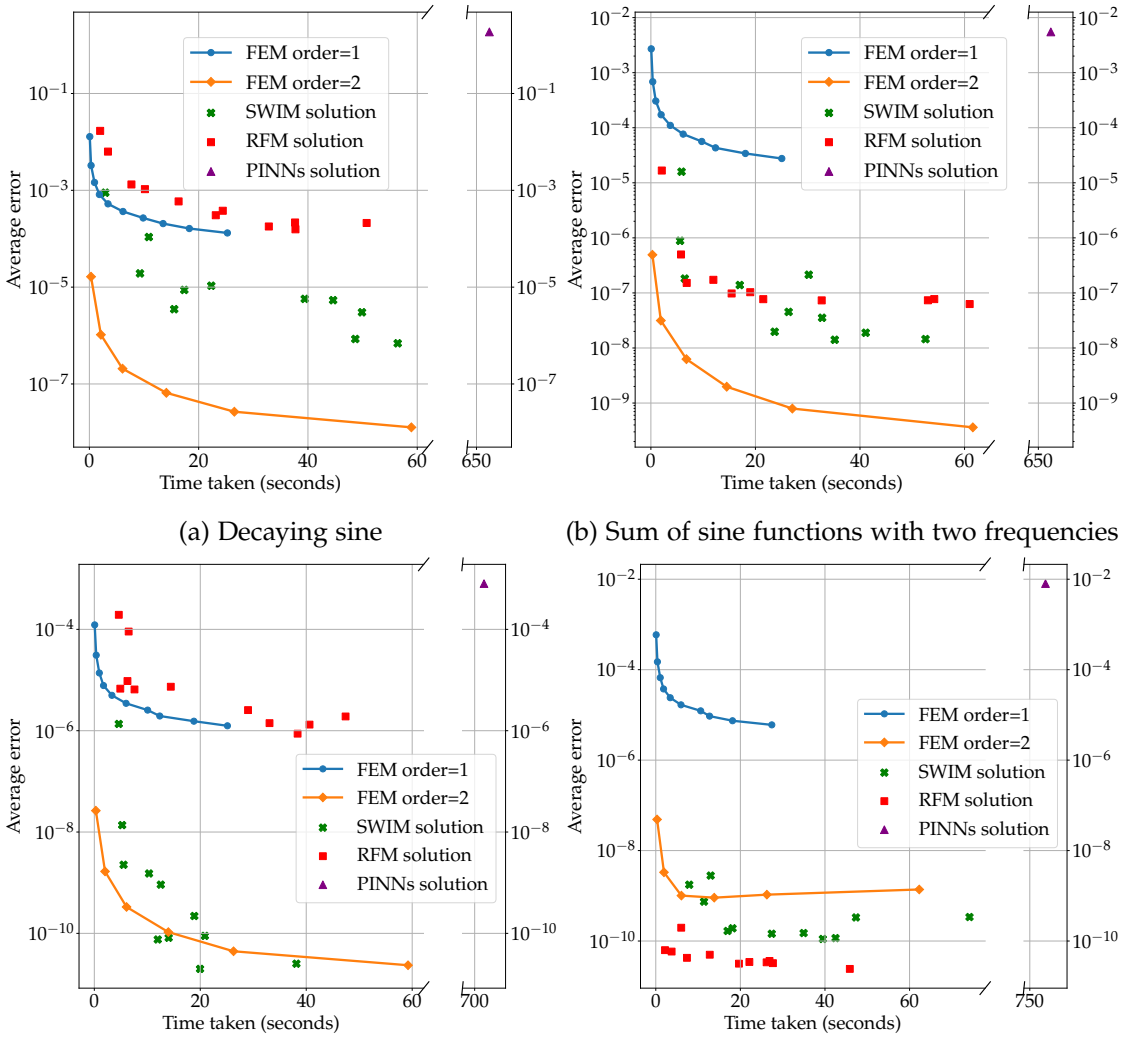
#### 3.2.1.6 Comparison with conventional methods

In this section, we compare the SWIM method's solution with FEM and PINN. The domain size is fixed to $[-1, 1]^2$ for the 2D case. For FEM results, we used FEniCS, a python-based FEM library [Bar+23]. First and second-order basis functions are employed, with the number of nodes varying from $100^2$ to $1000^2$. Meanwhile, for the SWIM method, the number of points sampled from the domain and boundary is fixed at 10000 and 300 respectively, with the layer width increased from 500 to 3000. PINN utilized a fixed number of points sampled from the domain and boundary at 10000 and 300 respectively, with a layer width of 500 chosen for comparison. The parameters are chosen such that the time it takes to obtain the solution for FEM and SWIM methods is comparable, and approximately the same number of trainable parameters are present in the PINN and SWIM method with largest layer width. This comparative analysis sheds light on the performance differences across the methods, highlighting how variations in basis functions, number of points sampled, and layer width influence solution accuracy and computational efficiency.

The results are presented in fig. 3.15 in which the accuracy or the average error in the solution is plotted against the computational time required to obtain the solution of the PDE. The results suggest that the SWIM method performs well compared to FEM, whereas PINN struggle to approximate the solution with comparable accuracy and time taken to solve the PDE. The general trend observed is that the accuracy of the SWIM method lies between the accuracy of FEM of first-order and second-order basis functions. An interesting observation for random initialized neural networks is that it outperforms all methods for functions without local variations, e.g., $5^{th}$ order polynomial function.

We can also observe for the SWIM and random initialized neural network the general trend that increasing layer width results in increasing computational time and improvement in accuracy is not followed. This shows that increasing just one parameter out of the layer width and number of interior points does not guarantee improvement in the accuracy or computational time.

**Conclusion 5** *For simple PDEs like the Poisson equation, the accuracy of the SWIM method lies in between the accuracy of FEM with first and second-order basis functions. SWIM method for all the cases outperforms random initialization or for simple functions like the* $5^{th}$ *order polynomial, both the methods perform equally well. PINN struggle to compete with other methods, suggesting the presented method is a good alternative to solving PDEs using neural networks.*




(a) Decaying sine

(b) Sum of sine functions with two frequencies

(c) Gaussian function

(d) $5^{th}$ order polynomial

Figure 3.15: Average error vs time taken to solve for FEM, PINN, SWIM and random initialized neural network solution

#### 3.2.1.7 3D linear PDEs

In this section, we present the results for the method extended to 3D linear Poisson PDEs. The experimental setup is same as other previous experiments were we solve the Poisson preoblems given in section 3.2.1.1. The domain is extended to $[-1, 1]^3$ for the 3D case. The boundary conditions are set to be the solution of the PDE at the boundary points. The results given in table 3.1 suggest that the method works well for the 3D case too and can be extended to higher dimensions as well. The advantage of this method lies in that fact that the ease with which the lower dimensional implementation can be extended to higher dimensions. The changes required to extend the method to higher dimensions are the architecture of neural network and computation of extra derivative term in the loss function. Comparing with conventional methods which suffer from the curse of dimensionality, the neural network based methods are more scalable and can be a good alternate to solving PDEs in higher dimensions.

Table 3.1: Results for 3D linear Poisson PDEs for 4 functions

| **Functions** | **Points sampled** | | **Layer Width** | **Error** | |
|---|---|---|---|---|---|
| | Interior domain | Boundary | | Max error | Avg error |
| Decaying sine | 32768 | $32^2 \cdot 6$ | 8000 | 3.15e-03 | 3.20e-04 |
| Sine 2 freq | 32768 | $32^2 \cdot 6$ | 8000 | 6.61e-06 | 6.68e-07 |
| Gaussian | 32768 | $32^2 \cdot 6$ | 8000 | 6.52e-06 | 1.29e-07 |
| Polynomial | 32768 | $32^2 \cdot 6$ | 8000 | 1.32e-08 | 4.32e-10 |

#### 3.2.1.8 Analysis of linear layer parameters

In this experiment, we observe the relation between the linear layer parameters, specifically the weights of the linear layer and the distance between the input points. The experiment is done for the 2D case for a square $[-1, 1]^2$ domain. Results for the problems from section 3.2.1.1 are presented in figs. 3.16, 3.17, 3.18, and 3.19. The linear layer parameters are arranged in decreasing magnitude in the first plot for all the functions. The distance between the input points is presented in the second plot, corresponding to the sorted order of the magnitude of linear layer parameters in the first plot. This provides some insights into the relation between the magnitude or the importance of the activation function and the distance between the input points chosen to place the activation function. The results suggest a clear relation between the input points and the magnitude of linear layer parameters for all the functions. For the pair of points that are very close to each other, the magnitude of the corresponding linear layer parameter is very low. A cut-off for linear layer parameter magnitude is set as $10^{-5}$ and corresponding distance between the input points is marked. This shows that for distance between input point less than the cut-off distance, the corresponding activation function does not contribute in approximating the solution of the PDE.

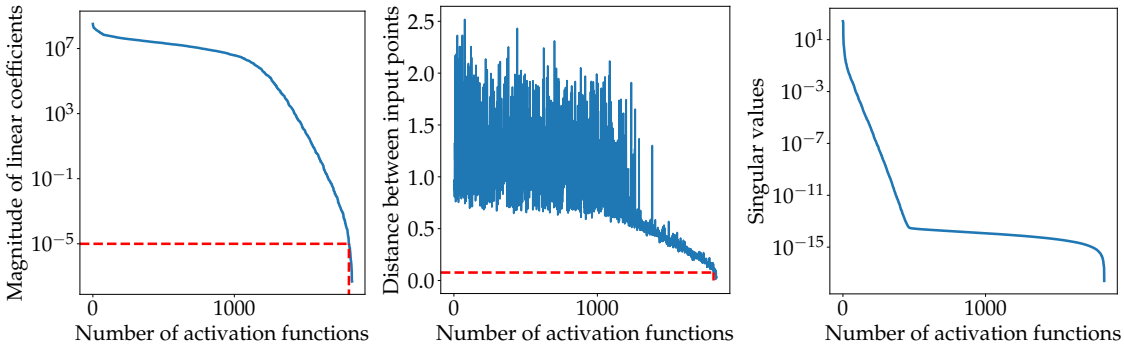
In the third plot, we present the singular values of the covariance operator of the basis functions. In [MQS23], a similar experiment is done to improve basis functions into

orthonormal basis functions for DeepONets based on singular values of the covariance matrix. The covariance operator is given as follows

$$\mathcal{C} = \sum_{k=1}^{N} \tau_k \otimes \tau_k = \sum_{k=1}^{N} \tau_k \langle \tau_k, \cdot \rangle , \tag{3.27}$$

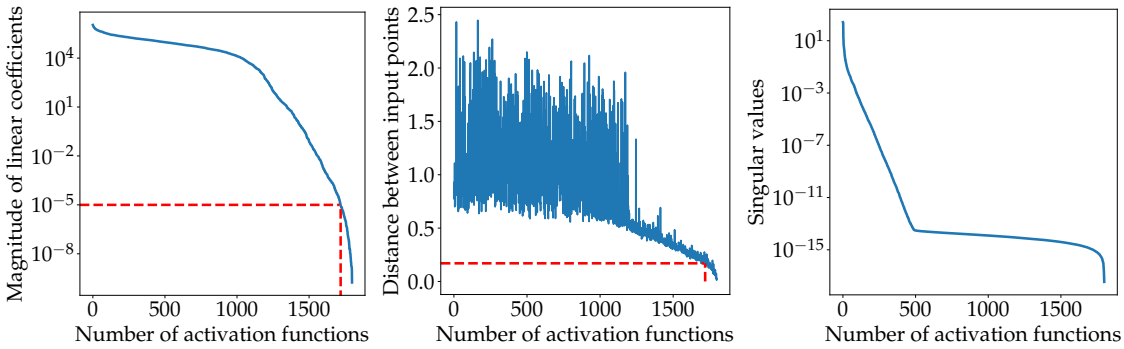
where $\tau_k$ are the predictions of the basis functions for a point $k$ and $\otimes$ denotes the outer product. $N$ is the total number of points sampled from the domain.

The singular value plots also suggest that approximately 500 basis functions are enough to approximate the function space. This is an important observation as the number of basis functions could be reduced, effectively improving the computational time with minimal effect on the accuracy of the solution.



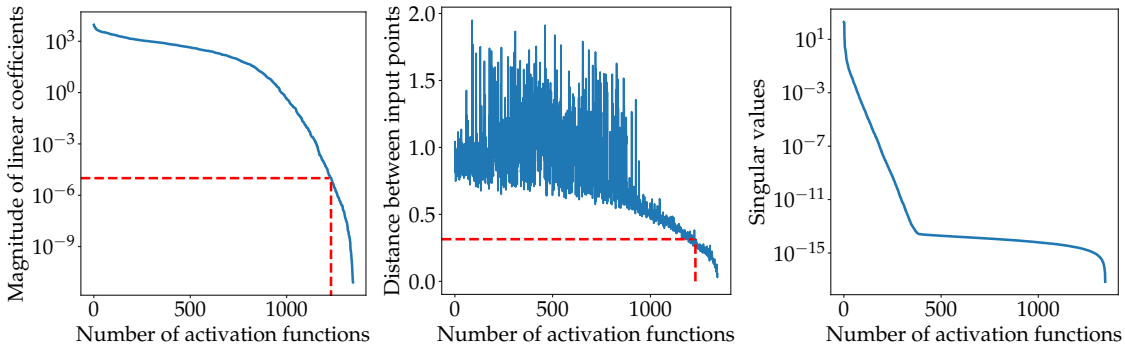

(a) Magnitude of linear layer coefficients (b) Distance between input points (c) Singular values of covariance matrix of basis functions

Figure 3.16: Decaying sine function



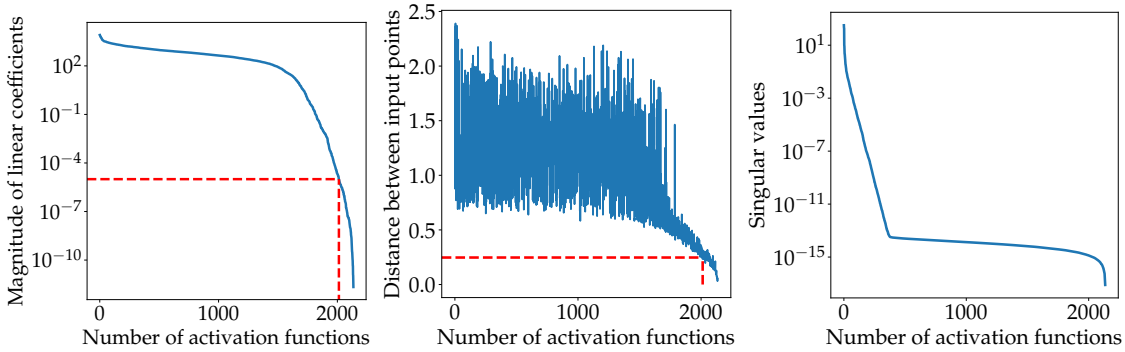

(a) Magnitude of linear layer coefficients (b) Distance between input points (c) Singular values of covariance matrix of basis functions

Figure 3.17: Sum of sine functions with 2 frequencies




(a) Magnitude of linear layer coefficients (b) Distance between input points (c) Singular values of covariance matrix of basis functions

Figure 3.18: Gaussian function




(a) Magnitude of linear layer coefficients (b) Distance between input points (c) Singular values of covariance matrix of basis functions

Figure 3.19: $5^{th}$ order polynomial function

**Conclusion 6** *These experiments suggest that some basis functions do not contribute to approximating the solution for the PDE. These basis functions can be discarded, resulting in fewer parameters to solve. Thus reducing the computational time taken to solve the linear system of equations.*

### 3.2.2 Non-linear Partial Differential Equation

In the previous section, we extensively experiment with the linear Poisson equation. This section extends our implementation and shows that it can also solve non-linear partial differential equations.

#### 3.2.2.1 Non-linear Poisson equation

The non-linear Poisson equation extends the fundamental concept of the traditional Poisson equation by introducing non-linear terms. Widely employed across scientific disciplines such as physics, chemistry, and biology, this equation serves to describe the distribution of diffusing quantities exhibiting non-linear behaviors within a given spatial domain.

Mathematically, the non-linear Poisson equation can be represented as

$$\nabla \cdot (g(u)\nabla u) = F(x_1, x_2, \ldots, x_d). \tag{3.28}$$

In this formulation, $g(u) = 1 + u$, $\nabla$ denotes the gradient operator, $u$ is the scalar field of interest, and $F$ denotes a function of independent variables $x_1, x_2, \ldots, x_d$. This equation can be easily extended to any dimension $d$.

In this section we present the results for the following non-linear partial differential equation in 2D

$$\nabla \cdot ((1 + u)\nabla u) = 0.5 * \exp(-0.5 * (x + y)) + \exp(-(x + y)). \tag{3.29}$$

The solution of the eq.3.29 is given as

$$u = \exp(-0.5 * (x + y)). \tag{3.30}$$

The eq.3.29 is solved over a 2D domain $[-1, 1]^2$ and the boundary conditions are enforced by evaluating the solution of the PDE given in 3.30 at the boundary points.

In this section, we compare the solution accuracy and time taken to solve the non-linear Poisson equation by SWIM method, FEM and PINN. The domain size is fixed to $[-1, 1]^2$ for the 2D case. For FEM, first-order basis functions are used to solve the non-linear Poisson equation, and the number of nodes is increased from $100^2$ to $1000^2$. For the SWIM method, the number of points sampled from the domain and boundary is fixed at 1000 and 300 respectively, with the layer width increasing from 100 to 700. For PINN, we use 1000 and 300 points sampled from the domain and boundary respectively, and the layer width is set at 200 so that the number of trainable parameters is approximately equal for the SWIM method with the largest layer width. We present the results in fig. 3.20 with the pair of input points sampled uniformly as the right-hand side function does not give us a good idea about the solution of the PDE, and duplicate activation functions are pruned in the SWIM method. The results suggest that for the simple problem we tried the SWIM method performs well compared to FEM, whereas PINN struggle to approximate the solution with comparable accuracy and time taken to solve the non-linear PDE is also high.

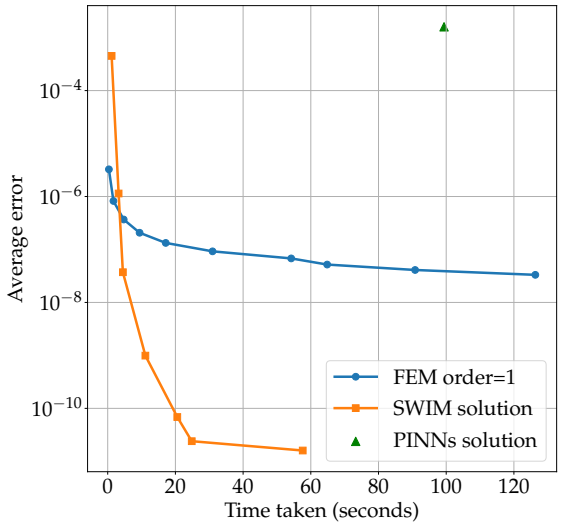



Figure 3.20: Accuracy vs time taken to solve for FEM, PINN and SWIM initialized neural network solution

#### 3.2.2.2 Navier Stokes - Steady state lid driven cavity

In the previous section we present the results for a scalar non-linear PDE. Now, we delve into the solution of the Navier-Stokes equation in two dimensions, approximated using neural networks. Given that the solution for the Navier-Stokes problem comprises a vector belonging to $\mathbb{R}^3$, representing the $u$, $v$, and $p$ variables in 2D, and we need to solve three equations in total—continuity, x-momentum, and y-momentum—the neural network architecture is designed accordingly. It consists of two input neurons, a hidden layer, and three output neurons, facilitating the representation of the final solution of the equation.

The solution procedure resembles the method described earlier in section 3.1. Initially, we compute a residual vector for all the equations available to us, including the boundary conditions. This vector depends on the linear layer parameters corresponding to the three output variables. Subsequently, we compute the Jacobian of the residual vector with respect to all trainable parameters.

Finally, we employ a non-linear least squares approach to minimize the residual, effectively refining the solution until convergence is achieved. This iterative process ensures that the neural network effectively captures the dynamics of the Navier-Stokes equation, yielding accurate and reliable results for fluid flow simulations.

The continuity equation is given as follows in $2D$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \tag{3.31}$$

The 2$D$ incompressible steady state Navier-Stokes equation is given as follows

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{\partial P}{\partial x} + \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \tag{3.32}$$

$$u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{\partial P}{\partial y} + \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right), \tag{3.33}$$

where the non-dimensional parameter Reynolds number ($Re$) is given as

$$Re = \frac{UL}{\nu}, \tag{3.34}$$

where $U$ is the characteristic velocity, $L$ is the characteristic length and $\nu$ is the kinematic viscosity.
This section presents the results of a steady state-driven cavity problem with Reynolds number 100. The steady-state lid-driven cavity problem is a classic benchmark scenario in fluid dynamics, extensively studied for its relevance in understanding fluid flow patterns and boundary effects. In this scenario, a fluid-filled cavity is bounded by rigid walls on three sides, with the top lid moving at a constant velocity parallel to the base. The governing equations, typically the Navier-Stokes equations and appropriate boundary conditions describe the fluid's velocity and pressure distribution within the cavity. The solution to this problem provides insights into complex fluid behavior, including forming vortices, boundary layer development, and velocity gradients across the cavity.

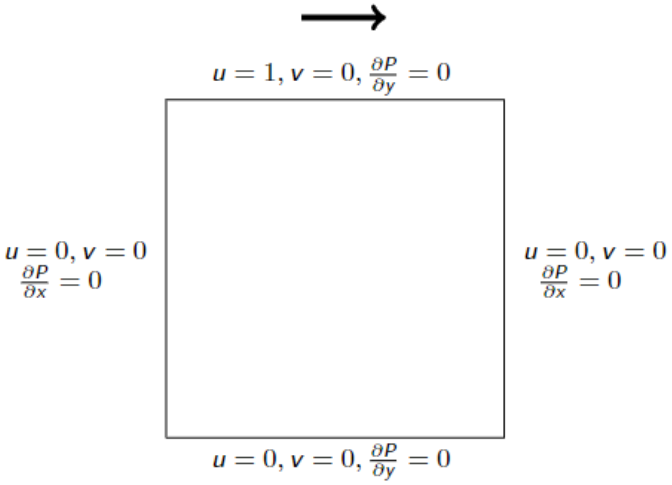
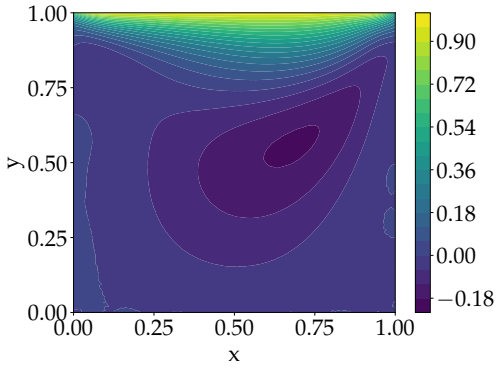



Figure 3.21: Boundary conditions for lid-driven cavity problem

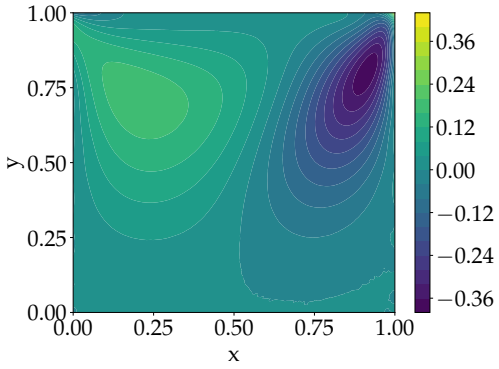
Along with the eq. 3.31, 3.32 and 3.33 we enforce the boundary conditions where the top lid moves at a constant velocity $u = 1$ and the other boundaries are set to no-slip condition. Neumann boundary conditions are applied to pressure on all the boundaries. The boundary conditions are given in fig. 3.21. Since the gradient of pressure is

constrained in the Navier-Stokes equation as well as in the boundary conditions. The pressure is set to 0 in the bottom left corner of the domain. The entire pressure field is computed with respect to this reference pressure.

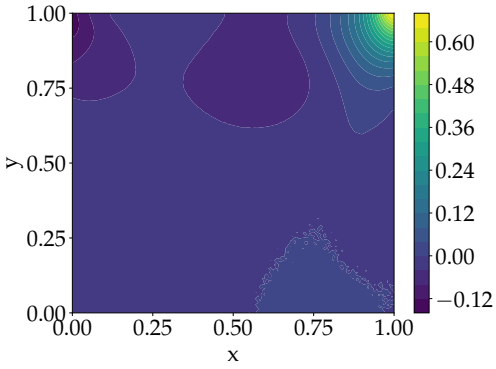
The solution contour plots are presented in fig. 3.22 and a comparison of the center-line u-velocity is given in fig. 3.23. Further increasing the Reynolds number, the solution was not as accurate, suggesting that while neural networks can approximate solutions for complex problems like Navier Stokes, they might not be as effective as traditional computational fluid dynamics methods regarding accuracy and reliability.




(a) u-velocity contour plot




(b) v-velocity contour plot




(c) pressure contour plot

Figure 3.22: Contour plots for lid-driven cavity problem with Reynolds number 100

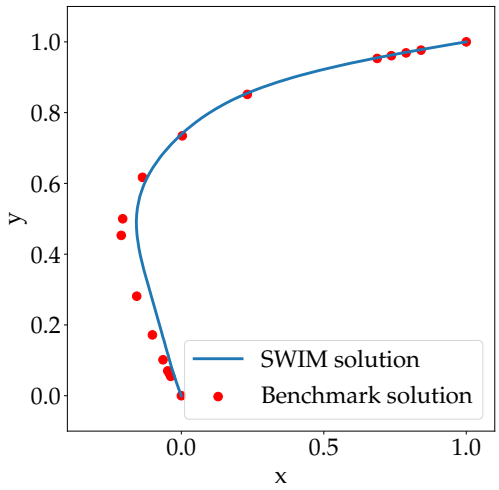



Figure 3.23: Comparison of centerline u-velocity with benchmark solution for lid-driven cavity problem with Reynolds number 100. Benchmark solution is taken from [GGS82]

## 3.3 Discussion

In this thesis, we extensively experiment with the linear Poisson PDE. We start with balancing losses between the PDE and boundary conditions. The results show that normalizing the coefficient matrix of the linear system of equations is the best strategy to balance the two losses. We then experiment with the ratio of interior points to layer width. The SWIM method uses the pair of input points to initialize the weights of the network parameters making it a particularly interesting experiment. We observe a marginal effect on the accuracy with the increasing ratio. The ratio of 5 is selected as a good balance between accuracy and computational time; since the computational time increases with increase in ratio. The next experiment aims to understand the effect of the activation function's steepness on the solution's accuracy. We observe that the steepness of the activation function significantly affects the solution's accuracy. For 1D PDEs, less steep tanh functions are better suited, whereas, for the 2D PDEs, the default steepness of the activation function performs well. In the next experiment, we observe the effect of domain decomposition on the accuracy and computational time needed to obtain the solution. For larger layer widths, the $2 \times 2$ split performs the best with comparable accuracy to a single domain, taking less time to obtain the solution. We then also compare the presented method with existing conventional numerical methods like FEM and PINN. The presented method shows comparable solution accuracy to FEM and is far superior to the PINN accuracy. In the next experiment, we present the results of extending the implementation to 3D, showcasing the scalability of the method. To conclude the experiments with the linear Poisson equation, we present an analysis of the linear layer parameters, the relation of the distance between the input points, and

the magnitude of linear layer parameters can be observed. The results suggest that some basis functions do not contribute to approximating the solution of the PDE, and similar results are also observed in the plot of singular values of the covariance operator of the basis functions. We then extend the implementation to solve the non-linear Poisson equation and compare the accuracy and computational time with FEM. The results show that the presented method also works well for simple non-linear PDEs. Finally, we showcase that this method can also be used to solve the steady-state Navier-Stokes equation for the lid-driven cavity problem. The method works well for the Reynolds number of 100, but increasing the Reynolds number to 500 results in an inaccurate solution. This suggests that the presented method can solve a large variety of simple PDEs, but with the increase in complexity of the problem, the presented method might not be the most efficient way to solve such problems.

# 4 Conclusion and Future Research Outlook

In this thesis, we present a neural network-based PDE solver for square domains. We first present the theory required to understand the concepts of neural networks, the universal approximation theorem of neural networks-the foundational theorem of neural networks, and explain the SWIM method used to train a neural network in chapter 2. Then we present the implementation details of the neural network based PDE solver in section 3.1. In this, we go through the steps required to implement the neural network-based PDE solver. We present the sampling of input points and then prepare the input to the neural network by normalizing the points. We then briefly explain the method to initialize the network in case of different PDEs. Further, we present the modifications required in domain decomposition and end the chapter by describing the training process of the neural network for linear/non-linear PDEs. In section 3.2, we extensively performed experiments with the linear Poisson equation to improve the solution accuracy and understand the SWIM method in detail. In this section, we also present the results for solving non-linear PDEs. We start with the non-linear Poisson equation and then finally obtain the solution of the steady-state Navier-Stokes equation for the lid-driven cavity case.

As part of future research, there are several directions one can take to improve and understand the method better. One possible direction to improve the method with domain decomposition for linear PDEs could be to use a sparse preconditioner for the resulting coefficient matrix. It would enable the method to take advantage of the sparsity of the coefficient matrix and solve the system of linear equations efficiently. This is important as the condition number of the resulting coefficient matrix is high, making it difficult for the sparse solver to work well for such a system of linear equations. Another possible direction related to domain decomposition could be to use arbitrary domain decomposition hierarchically. This can be achieved by solving the PDE with a single domain and then initializing subdomains only in the region where the error in the single-domain solution is high. Then, the PDE can be solved by combining the solution from all the subdomains. Another obvious direction could be to look into time-dependent PDEs. The method can be extended to solve the time-dependent PDEs by either assuming time as another dimension and solving for the solution dependent on space and time or by using conventional time-stepping methods with basis functions updated after regular time intervals using the SWIM method. From the results in section 3.2.1.8, we can also conclude that the improvement or pruning of unnecessary basis functions could also be looked into, improving computational time to solve PDEs.

# Bibliography


[Ako13] D. Akomolafe. "Scholars Research Library Comparative study of biological and artificial neural networks." In: *European Journal of Applied Engineering and Scientific Research* 2 (Jan. 2013), pp. 36–46.

[Bar+23] I. A. Baratta, J. P. Dean, J. S. Dokken, M. Habera, J. S. Hale, C. N. Richardson, M. E. Rognes, M. W. Scroggs, N. Sime, and G. N. Wells. *DOLFINx: The next generation FEniCS problem solving environment*. Dec. 2023. DOI: 10.5281/zenodo.10447666. URL: https://doi.org/10.5281/zenodo.10447666.

[BCL99] M. A. Branch, T. F. Coleman, and Y. Li. "A Subspace, Interior, and Conjugate Gradient Method for Large-Scale Bound-Constrained Minimization Problems." In: *SIAM Journal on Scientific Computing* 21.1 (1999), pp. 1–23. DOI: 10.1137/S1064827595289108. eprint: https://doi.org/10.1137/S1064827595289108. URL: https://doi.org/10.1137/S1064827595289108.

[BCN18] L. Bottou, F. E. Curtis, and J. Nocedal. *Optimization Methods for Large-Scale Machine Learning*. 2018. arXiv: 1606.04838 [stat.ML].

[Bis06] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, Jan. 2006. URL: https://www.microsoft.com/en-us/research/publication/pattern-recognition-machine-learning/.

[Bol+23] E. L. Bolager, I. Burak, C. Datar, Q. Sun, and F. Dietrich. *Sampling weights of deep neural networks*. 2023. arXiv: 2306.16830 [cs.LG].

[Che+22] J. Chen, X. Chi, W. E, and Z. Yang. *Bridging Traditional and Machine Learning-based Algorithms for Solving PDEs: The Random Feature Method*. 2022. arXiv: 2207.13380 [math.NA].

[Cyb89] G. Cybenko. "Approximation by superpositions of a sigmoidal function." In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[DE20] A. D. Jagtap and G. Em Karniadakis. "Extended Physics-Informed Neural Networks (XPINNs): A Generalized Space-Time Domain Decomposition Based Deep Learning Framework for Nonlinear Partial Differential Equations." In: *Communications in Computational Physics* 28.5 (2020), pp. 2002–2041. ISSN: 1991-7120. DOI: https://doi.org/10.4208/cicp.OA-2020-0164. URL: http://global-sci.org/intro/article_detail/cicp/18403.html.

[DL21] S. Dong and Z. Li. "Local extreme learning machines and domain decomposition for solving linear and nonlinear partial differential equations." In: *Computer Methods in Applied Mechanics and Engineering* 387 (Dec. 2021), p. 114129. ISSN: 0045-7825. DOI: 10.1016/j.cma.2021.114129. URL: http://dx.doi.org/10.1016/j.cma.2021.114129.

[Fad24] M. S. Fadali. "Least-Squares Estimation." In: *Introduction to Random Signals, Estimation Theory, and Kalman Filtering*. Singapore: Springer Nature Singapore, 2024, pp. 177–197. ISBN: 978-981-99-8063-5. DOI: 10.1007/978-981-99-8063-5_6. URL: https://doi.org/10.1007/978-981-99-8063-5_6.

[Fle70] R. Fletcher. "A new approach to variable metric algorithms." In: *The Computer Journal* 13.3 (Jan. 1970), pp. 317–322. ISSN: 0010-4620. DOI: 10.1093/comjnl/13.3.317. eprint: https://academic.oup.com/comjnl/article-pdf/13/3/317/988678/130317.pdf. URL: https://doi.org/10.1093/comjnl/13.3.317.

[GB10] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[GD21] A. Gnanasekaran and E. Darve. *Hierarchical Orthogonal Factorization: Sparse Least Squares Problems*. 2021. arXiv: 2102.09878 [math.NA].

[GGS82] U. Ghia, K. Ghia, and C. Shin. "High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method." In: *Journal of Computational Physics* 48.3 (1982), pp. 387–411. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(82)90058-4. URL: https://www.sciencedirect.com/science/article/pii/0021999182900584.

[He+15] K. He, X. Zhang, S. Ren, and J. Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV].

[Hei+21] A. Heinlein, A. Klawonn, M. Lanser, and J. Weber. "Combining machine learning and domain decomposition methods for the solution of partial differential equations—A review." In: *GAMM-Mitteilungen* 44.1 (2021). e202100001 GAMM.202100001, e202100001. DOI: https://doi.org/10.1002/gamm.202100001. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/gamm.202100001. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/gamm.202100001.

[HK24] L. Herrmann and S. Kollmannsberger. "Deep learning in computational mechanics: a review." In: *Computational Mechanics* (Jan. 2024). ISSN: 1432-0924. DOI: 10.1007/s00466-023-02434-4. URL: https://doi.org/10.1007/s00466-023-02434-4.

[JKK20] A. D. Jagtap, E. Kharazmi, and G. E. Karniadakis. "Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems." In: *Computer Methods in Applied Mechanics and Engineering* 365 (2020), p. 113028. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2020.113028. URL: https://www.sciencedirect.com/science/article/pii/S0045782520302127.

[KB17] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[Li+21] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. *Fourier Neural Operator for Parametric Partial Differential Equations*. 2021. arXiv: 2010.08895 [cs.LG].

[LLF98] I. Lagaris, A. Likas, and D. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations." In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 987–1000. DOI: 10.1109/72.712178.

[Lu+21] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis. "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators." In: *Nature Machine Intelligence* 3.3 (Mar. 2021), pp. 218–229. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00302-5. URL: http://dx.doi.org/10.1038/s42256-021-00302-5.

[MMN21] B. Moseley, A. Markham, and T. Nissen-Meyer. *Finite Basis Physics-Informed Neural Networks (FBPINNs): a scalable domain decomposition approach for solving differential equations*. 2021. arXiv: 2107.07871 [physics.comp-ph].

[Mor78] J. J. Moré. "The Levenberg-Marquardt algorithm: Implementation and theory." In: *Numerical Analysis*. Ed. by G. A. Watson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 105–116. ISBN: 978-3-540-35972-2.

[MQS23] B. Meuris, S. Qadeer, and P. Stinis. "Machine-learning-based spectral methods for partial differential equations." In: *Scientific Reports* 13.1 (Jan. 2023). DOI: 10.1038/s41598-022-26602-3. URL: https://doi.org/10.1038/s41598-022-26602-3.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors." In: *Nature* 323 (1986), pp. 533–536. URL: https://api.semanticscholar.org/CorpusID:205001834.

[RPK19] M. Raissi, P. Perdikaris, and G. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations." In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045. URL: https://www.sciencedirect.com/science/article/pii/S0021999118307125.

[SS18] J. Sirignano and K. Spiliopoulos. "DGM: A deep learning algorithm for solving partial differential equations." In: *Journal of Computational Physics* 375 (Dec. 2018), pp. 1339–1364. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2018.08.029. URL: http://dx.doi.org/10.1016/j.jcp.2018.08.029.

[Vir+20] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.