Department of Informatics
School of Computation, Information and Technology
Technical University of Munich

Solving Heat Conduction Problems with DeepONets

Alan Xavier

Solving Heat Conduction Problems with DeepONets

Alan Xavier

Department of Informatics
School of Computation, Information and Technology
Technical University of Munich

Solving Heat Conduction Problems with DeepONets

Alan Xavier

Thesis for the attainment of the academic degree

Master of Science (M.Sc.) Computational Science and Engineering (CSE)

at the School of Computation, Information and Technology of the Technical University of Munich.

Examiner:
Dr. Felix Dietrich

Advisor:
Dr.-Ing. Henning Sauerland (Hitachi)

Submitted:
Munich, 31.03.2023

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Munich, 31.03.2023 Alan Xavier

Abstract

Scientific machine learning (SciML) involves the use of machine learning (ML) in solving various problems in computational science and engineering, that can often be described by ordinary and partial differential equations (ODEs and PDEs). One approach is the physics-informed neural networks (PINNs) which embeds physical laws into the loss function of the neural network (NN) to determine the solution of PDEs. This approach has been studied extensively to solve both forward and inverse problems in diverse scientific and engineering applications since its introduction.

However, a new approach that was recently introduced, involves learning operators that map between infinite-dimensional function spaces using NNs. The purpose of this thesis is to study one specific architecture for operator learning called deep operator networks (DeepONets). The focus lies on the use of DeepONet models to solve a common PDE in science and engineering, the heat conduction equation. We employ data-informed and physics-informed DeepONets to solve three different parametric heat conduction problems subjected to different initial and boundary conditions (IBCs) using the Modulus framework developed by NVIDIA. We discuss how the training data-set is generated, and compare the differences in data-set size required by each DeepONet variant. We also provide details of the loss function definitions for each DeepONet, and its implementation. We also show that after sufficient training, each DeepONet is able to predict, with reasonable accuracy, the parametric solutions for each heat conduction problem.

PINNs and classical numerical solvers can only provide solutions for a specific set of input parameters, and any modifications requires either retraining the model or running a new numerical simulation. DeepONets, however, can provide different PDE solution function spaces when trained using parametric functions as inputs. This new approach is rather exciting as it is capable of predicting parametric solutions orders of magnitude faster than traditional solvers and without further training.

Contents

Abstract **ix**

1 Introduction **1**

2 State of the Art **3**
2.1 Theoretical Background 3
2.1.1 Neural Networks 3
2.2 Introducing Deep Operator Networks (DeepONets) 6
2.2.1 Deep Dive into DeepONets 7
2.3 Modulus Framework 12
2.3.1 Modulus Building Blocks 12
2.3.2 Modulus Workflow 13
2.3.3 Activation Functions 13

3 Solving Heat Conduction Problems with DeepONets **15**
3.1 Heat Conduction Equation 15
3.1.1 Initial and Boundary Conditions (IBCs) 15
3.1.2 Types of Boundary Conditions 16
3.1.3 Heat Operator 17
3.1.4 Test Error Analysis 17
3.2 1D Transient Heat Equation 18
3.2.1 Problem Definition 18
3.2.2 Data-set Generation 18
3.2.3 Loss Function 20
3.2.4 Training 22
3.2.5 Results 23
3.2.6 Data Quality 25
3.2.7 Activation Functions (AFs) 25
3.2.8 Summary of Findings 28
3.2.9 Conclusions 28
3.3 2D Steady-State Heat Equation 29
3.3.1 Problem Definition 29
3.3.2 Data-set Generation 30
3.3.3 Loss Function 32
3.3.4 Training 37
3.3.5 Results 39
3.3.6 Summary of Findings 43
3.3.7 Conclusions 43
3.4 2D Axisymmetric Transient Heat Equation 44
3.4.1 Problem Definition 44
3.4.2 Data-set Generation 45
3.4.3 Non-dimensionalizing the Axisymmetric Heat Equation 47
3.4.4 Loss Function 48
3.4.5 Training 52
3.4.6 Results 54

3.4.7 Summary of Findings . . . 62
3.4.8 Conclusions . . . 62

4 Conclusion **63**

Bibliography **65**

A Appendix **69**
A.1 Activation Functions (AFs) . . . 69
A.1.1 Activation Function Graphs . . . 69
A.1.2 Activation Function Results . . . 72
A.2 2D Steady-State Heat Equation Results . . . 76
A.3 Heat Equation 2D Transient Axisymmetric Results . . . 78

1 Introduction

Artificial neural networks (ANNs) are known to be highly efficient approximators of continuous functions, i.e. functions with no sudden changes in values such as discontinuities, holes or jumps. Recent advances in machine learning (ML), particularly in science and engineering, has garnered a lot of popularity among researchers lately driven by its success in approximating highly complex and nonlinear functions. One advancement came in 2018, when Raissi et al. [18] introduced the concept of physics-informed neural networks (PINNs) for solving ordinary differential equations (ODEs) and partial differential equations (PDEs) using neural networks (NNs).

However, NNs can also be used to learn operators instead of functions. In 2020, Lu Lu et al. [17] developed deep operator networks (DeepONets), inspired from the *Universal Approximation Theorem for Operators* [3], capable of learning both linear and nonlinear continuous operators from data. A physics-informed variant, inspired by how the physical laws are embedded into the network's loss function in PINNs, was later introduced by Wang et al. [28]. The main difference between PINNs and DeepONets is that networks trained to learn functions (PINNs) can only be used to map between finite-dimensional vector spaces, whereas those trained to learn operators (DeepONets) can be used to map between infinite-dimensional function spaces [28]. Both concepts leverage the *Universal Approximation Theorem* that states that NNs can approximate any continuous function (PINNs) or operator (DeepONets) to arbitrary accuracy if no constraint is placed on the width and depth of the network's hidden layers [17].

This study will focus on the application of DeepONets, both a data-informed and physics-informed variant, on the heat conduction equation. This equation has significant importance in diverse scientific and mathematical fields. DeepONets is a fairly new ML concept with limited research studies (to the author's knowledge) focused on its application towards solving physics-based problems described by the heat conduction equation, outside of the very simple examples presented in [17] and [28]. The goal of this thesis is to gain a deeper understanding of DeepONets, and its implementation in NVIDIA's Modulus framework [22]. In particular, we aim to understand the approximation capabilities of DeepONets for a range of steady-state and transient heat conduction problems subject to various material conductivities, boundary and initial conditions. Data-informed and physics-informed versions of DeepONets will be implemented, for simulating both data-rich and data-sparse regimes.

This thesis is structured as follows. In section 2.1.1, we introduce the concept of neural networks (NNs) and provide an example on how these networks are trained. We then discuss the concept and architecture of DeepONets, and describe the differences between data-informed (DI-DeepONet) and a physics-informed (PI-DeepONet) versions in section 2.2.1. Section 2.3 briefly describes the Modulus framework, its components and a common workflow used to develop physics-informed networks, as well as a brief overview of the different activation functions (AFs) provided. In section 3.1, we introduce the heat conduction equation, and the types of boundary conditions (BCs) that will be used throughout our study. This is followed by section 3.2, 3.3 and 3.4, each discussing the implementation of DeepONets for solving a transient one-dimensional, steady-state two-dimensional, and a transient axisymmetric heat conduction problem respectively.

2 State of the Art

2.1 Theoretical Background

This section introduces the concept of neural networks, DeepONets and the heat equation. It is intended to aid the reader in understanding the concept of DeepONets, its subsequent implementation and analysis, and interpretation of the results.

2.1.1 Neural Networks

Architecture

Neural networks (NNs) consist of a bunch of neurons working together and performing some mathematical transformation of its inputs, in order to solve complex problems. These networks are extensively used in deep learning and machine learning as a part of artificial intelligence. ANNs consist of an input layer, one or more hidden layers, and an output layer as shown in figure 2.1. The network consists of nodes arranged into parallel layers, where each node is connected to nodes in an adjacent layer, with this connection having an associated weight $w_{ij}^{l} \in R$.

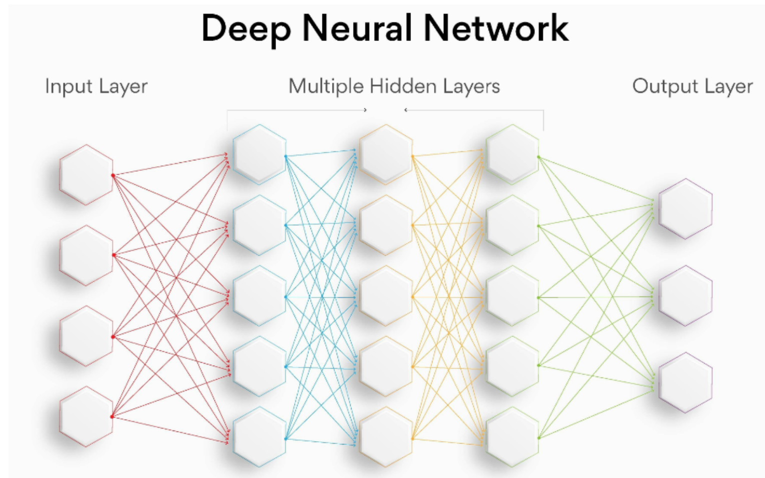


Figure 2.1 An illustration of a Feed Forward Neural Network [9].

Each node then performs a weighted sum of its inputs and produces an output depending on its associated activation function σ as shown in equation 2.1, where N_l represents the number of nodes in layer l, $\{x_i\}_i^{N_{l-1}}$ is the input from node i in the previous layer, N_{l-1}, and b_l is the bias associated with layer l. The bias is introduced to each node to offset the activation function in order to produce the desired output. The final prediction of the network is referred to by $\hat{y}$. This mathematical procedure is often referred to as forward propagation, and given by

$$\hat{y} = \sigma\left(\sum_{j=1}^{N_l}\sum_{i=1}^{N_{l-1}} w_{ij}^{l} x_i + b_j^{l}\right). \tag{2.1}$$

The activation function introduces non-linearity into the network, allowing it to estimate complex nonlinear functions. These function are crucial in determining the accuracy of a model and the computational efficiency of training a model. They have major effect on the network's ability to converge (by finding the optimal weights w and biases b). Without this non-linearity, a multilayered network would act equivalent to a single layered network, as the network would become a linear combinations of linear functions. Refer to section 2.3.3 for a discussion on different types of activation functions.

Training

Training a network consists of tuning the weights w^l and biases b^l in each layer l, which are randomly initialised during the first iteration of forward propagation, such that the network output $\hat{y}$ matches the desired output y, as given by the network's loss function L. The weights and biases are often referred to as network parameters $\theta = (w, b)$, and tuned by back-propagation.

During each iteration of forward propagation, we obtain the network output $\hat{y}$. The error (or loss) is then calculated with respect to the actual objective y, since for DeepONets, we are performing supervised learning (i.e. training through the use of a labelled training data-set of size n). The total error $L \in R$ is then computed, with the aim to minimize this loss, $L \to 0$, by tuning the network parameters θ by performing an iteration of back-propagation. The loss function indicates how well these parameters allow the network to produce the desired result. A generic loss function for a training data-set of size n is

$$L(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{2.2}$$

where y_i, $\hat{y}_i$ are the actual objective and network prediction, respectively, for training sample i.

The optimal parameters θ are obtained by gradient descent, which is an optimization algorithm used to find a local minimum of a differentiable function, by propagating the gradient of L with respect to θ (i.e. $\frac{dL}{d\theta}$) through the network in the reverse direction. In each iteration, the partial derivatives of the loss function, with respect to each network parameters, is computed by the chain rule, as shown in equation 2.3, for w.

$$\frac{dL(y, \hat{y})}{dw} = \frac{dL(y, \hat{y})}{d\hat{y}} \times \frac{d\hat{y}}{dz} \times \frac{dz}{dw}, \tag{2.3}$$

where $z = wx + b$ is a linear combination of input x with parameters w, b.

The model parameters θ in each layer are then updated by taking a step into the opposite direction of the gradient (i.e. descent) for each training iteration, i.e.

$$\theta_{n+1} = \theta_n - \alpha\left(\frac{dL(y, \hat{y})}{d\theta}\right). \tag{2.4}$$

α is a hyper-parameter often referred to as the learning rate and n represents the total number of training iterations.

α controls how much change is applied to the parameters during each training iteration. Training is thus an iterative process, where in order to obtain better results, multiple training iterations are required. Each iterations should reduce the total error L by updating θ, so that the network produces a more desirable output, i.e. $\hat{y} \simeq y$.

Example

Consider a NN with one input neuron (grey), one hidden layer neuron (orange), and one output neuron (green) as shown in figure 2.2. During forward propagation, the network's input or output of the node in the

previous layer x is passed to the neuron in the next (hidden) layer which then performs a weighted sum, $\theta_1 \cdot x$, before applying the activation function g. The output of this neuron, $a^{(2)}$, is then passed as input to the neuron in the next layer to perform the same operation by applying its own weight θ_2 and activation function g. The output from this neuron, $a^{(3)}$, is the final prediction for the true objective, y, by the network.

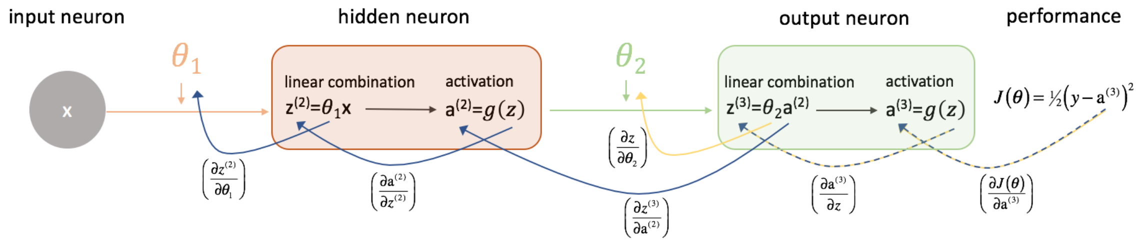


Figure 2.2 A training iteration performed by a Neural Network [11].

To assess the network's performance (i.e. minimize the difference between $a^{(3)}$ and y), the loss function J is computed. Note that the output of J depends on the network's prediction $a^{(3)}$, which depends on the current version of the network's parameters $\theta = \{\theta_1, \theta_2\}$. The partial derivative of J is then determined for each network parameter $\{\theta_1, \theta_2\}$ by

$$\frac{dJ(\theta)}{d\theta_1} = \frac{dJ(\theta)}{da^{(3)}} \times \frac{da^{(3)}}{dz^{(3)}} \times \frac{dz^{(3)}}{da^{(2)}} \times \frac{da^{(2)}}{dz^{(2)}} \times \frac{dz^{(2)}}{d\theta_1}, \tag{2.5}$$

$$\frac{dJ(\theta)}{d\theta_2} = \frac{dJ(\theta)}{da^{(3)}} \times \frac{da^{(3)}}{dz} \times \frac{dz}{d\theta_2}, \tag{2.6}$$

respectively.

Each parameter, θ_1, θ_2, is then updated by gradient descent and the learning rate α before performing a next training iteration by the following operation,

$$\theta_1 = \theta_1 - \alpha\left(\frac{dJ(\theta)}{d\theta_1}\right), \tag{2.7}$$

$$\theta_2 = \theta_2 - \alpha\left(\frac{dJ(\theta)}{d\theta_2}\right), \tag{2.8}$$

respectively. This process is repeated until $J(\theta) \to 0$.

2.2 Introducing Deep Operator Networks (DeepONets)

In 2020, Lu Lu et al. introduced the scientific ML community (SciML) to DeepONets [17, 16], developed from the Universal Approximators Theorem for Operators of Chen and Chen [3], to learn continuous operators from data. They showed the capability and robustness of DeepONet models to learn linear and nonlinear operators from dynamic systems, and physics-based systems described by ODEs and PDEs. They then conducted a series of numerical investigations using several examples to assess the performance of DeepONets, of which, the main conclusions are listed below.

1. **Anti-Derivative Operator**: Both stacked and unstacked DeepONet models (refer to section 2.2.1) were capable of learning the one-dimensional linear anti-derivative operator with reduced generalization error, when compared to a fully-connected neural network (FNN) trained to learn the same as shown in figure 2.3. They also showed that the *stacked* DeepONet performed the best out of all the trained networks, which required less memory and compute power to train, when compared to the *unstacked* variant.

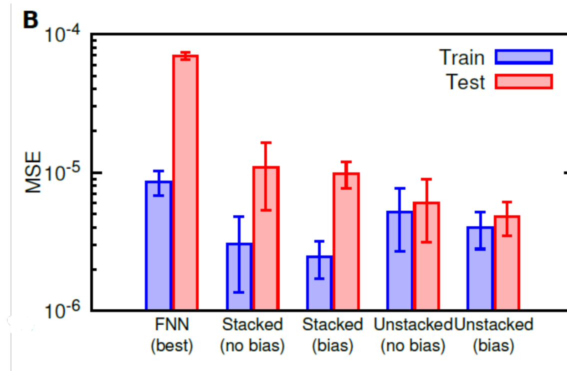


Figure 2.3 Training/test error for stacked/unstacked DeepONets compared to a FNNs [17].

2. **Gravity Pendulum with an External Force**: In this example, they investigated the effects of (1) the number of sensors m, (2) training data-set size $n \times p$, and (3) network architecture on the accuracy of the DeepONet models to learn required operator. The researchers showed that:

   a) Increasing the number of sensors led to improved network performance.

   b) Increasing the training data-set size led to improved generalization accuracy.

   c) Increasing the width of the branch and trunk networks does not automatically result in a decrease in model error, as the mean-squared error (MSE) of the model's predictions would first decrease up to a certain width before increasing.

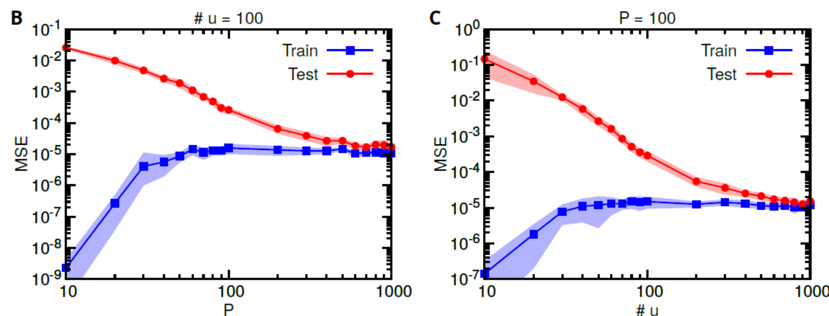


Figure 2.4 Training/test error for (B) different number of u and (C) different number of p [17].

3. **Diffusion-Reaction System with a Source Term**: In this scenario, different forcing terms (input function) u was used as input to the branch net. The researchers showed that increasing the number of input functions u with a fixed number of evaluation locations p, or the number of evaluation locations p with a fixed number of input functions u, led to improved network performance as shown in figure 2.4.

In 2021, Wang et al. expanded upon the primarily data-driven DeepONet models introduced by Lu Lu et al. to include a physics-informed variant [28] by drawing inspiration from PINN [18]. These physics-informed networks offer several advantages over the conventional data-informed networks by:

- Reducing the network's dependence for large training data-sets, and
- Allowing the networks to better capture the physical constraints of the underlying PDE, that was used to generate this data (refer to section 2.2.1).

Remarkably, the researchers showed that the physics-informed DeepONet achieved adequate predictive accuracy with almost no training data, except for the input-output data-set pairs for the initial and boundary conditions (IBCs).

The SciML community has mainly focused on learning functions through the use of PINNs [18, 29, 33]. Several studies focused on advancing the predictive accuracy of the PINNs ML model towards solving several heat conduction problems [21, 8, 31, 27, 10, 1, 33]. In this thesis, we investigate the effectiveness of DeepONets on learning the heat operator (section 3.1.3) on a series of one- and two-dimensional steady-state and transient heat conduction problems, subjected to different IBCs. The papers by Lu Lu et al. and Wang et al. investigated the robustness of the proposed networks on a series of simple examples, without focusing on its application on a specific physical problem. Additionally, the ability of PINNs to learn the heat function has already been extensively studied since its introduction in 2017, but limited studies, such as [13] and more recently [14] have investigated learning the heat operator with DeepONets.

2.2.1 Deep Dive into DeepONets

This section provides an overview of the model architecture of DeepONets [17, 16] with a focus on learning the solution operators of parametric partial differential equations (PDEs), i.e. a PDE system where some parameters are allowed to vary over a certain range. These parameters may determine the shape of the physical domain, the initial or boundary conditions (IBCs), constants or variable coefficients, etc. that define the system. The aim of DeepONets is to solve problems that can be described by a parametric nonlinear PDE of the form

$$N(u, s) = 0, \tag{2.9}$$

where u is the variable parameter (input functions), and s is the corresponding unknown solutions. For each u, we assume that there exists a unique solution $s = s(u)$, subject to equation 2.9 and appropriate IBCs.

Problem Setup

Following the formulation provided in [17], let $G : u \to G(u)$ be a solution operator that takes an input function u to the corresponding output function $G(u) = s(u)$. Then for any evaluation point y in the domain of $G(u)$, the output is a real number given as $G(u)(y) \in R$. The DeepONet predicts the solution map G, which we denote as G_θ, where θ represents the trainable parameters of the network. Thus DeepONets require two inputs, u and y in order to output the predicted solution map $G_\theta(u)(y)$.

As shown in figure 2.5, the input functions u are discretely represented at m consistent locations $\{x_i\}_i^m$, referred to as sensors. No constraint, however, is placed on the number and location of the evaluation points y which can vary for evaluating the output function $G(u)$ (defined for different u).

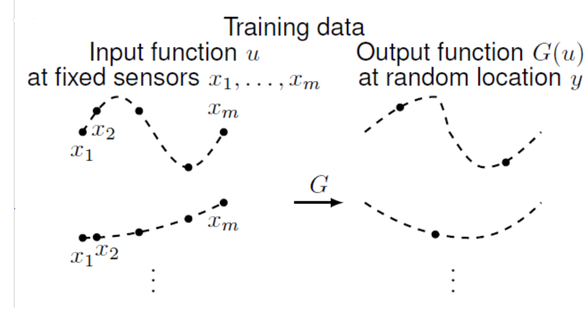


Figure 2.5 Illustration of training data where each input function u, which is represented at consistent m sensor locations, $x_1, x_2, ..., x_m$, and the output function $G(u)$ is evaluated at evaluation point y, which can vary in number and location per u [17].

Theorem 1 (Universal Approximation Theorem for Operator [3]) *Suppose that σ is a continuous non-polynomial function, X is a Banach Space, $K_1 \in X$, $K_2 \in R^d$ are two compact sets in X and R^d, respectively, V is a compact set in $C(K_1)$, G is a nonlinear continuous operator, which maps V into $C(K_2)$. Then for any $\epsilon > 0$, there are positive integers n, p, m, constants c_i^k, ξ_{ij}^k, θ_i^k, $\zeta_i^k \in R$, $w_k \in R^d$, $x_j \in K_1$, $i = 1, ..., n$, $k = 1, ..., p$, $j = 1, ..., m$, such that*

$$\left| G(u)(y) - \sum_{k=1}^{p} \sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right) \sigma(w_k \cdot y + \zeta_k) \right| < \epsilon, \tag{2.10}$$

holds for all $u \in V$ and $y \in K_2$.

Network Architecture

The network architecture for DeepONets is inspired by the Universal Approximation Theorem for Operator of Chen and Chen (theorem 1) [3]. Following this theorem, two separate input components $[u(x_1), u(x_2), ..., u(x_m)]^T$ and y are required by the network. However, we must consider that each $u(x_i)$ and y cannot be treated equally as y can be a vector with d components for high dimensional problems. Two sub-networks are therefore required to handle each component separately, which are referred to as the branch and trunk networks.

As shown in figure 2.6c, the branch net takes u, evaluated at sensors $\{x_i\}_i^m$, as input, and outputs a p dimensional feature embedding $\{b_k\}_k^p \in R$ (each b_k is an output from p branch nets). The trunk net takes the coordinates y as the input and also outputs a p dimensional feature embedding $[t_1, t_2, ..., t_p]^T \in R^p$. The final output of DeepONet G_θ is then obtained by merging the outputs of the branch and trunk nets via a dot product, i.e. $G_\theta = \sum_{k=1}^{p} b_k t_k$.

To reduce the computational and memory expensive requirements of p different branch nets, each having its own unique trainable network parameters, a single network is proposed that produces a p dimensional feature embedding $[b_1, b_2, ..., b_p]^T \in R^p$ as shown in figure 2.6d, where the branch nets now share network parameters. Note that the network architecture of the branch and trunk nets are not defined, and can be chosen to solve different problems allowing for improved flexibility and more efficient training.

Data-Informed vs Physics-Informed DeepONet

A conventional data-informed DeepONet is a data-driven method that requires large annotated data-sets of input-output triplets $\{\{u^{(i)}, y_j^{(i)}, G(u^{(i)})(y_j^{(i)})\}_{i=1}^n\}_{j=1}^p$ to predict the target output operator G. This data-set consists of n different input function $\{u^{(i)}\}_{i=1}^n$, each evaluated at m sensors $\{x_k\}_{k=1}^m$. Each $u^{(i)}$

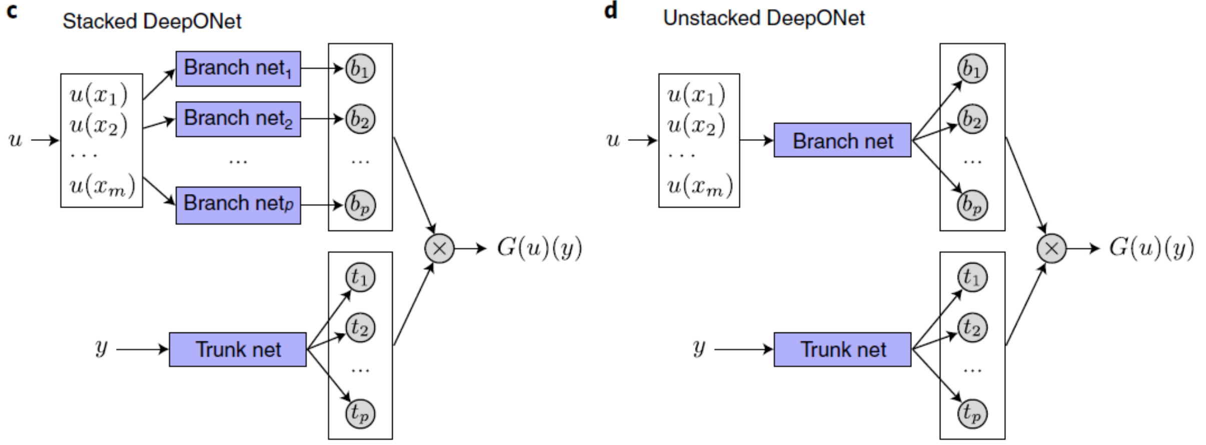


Figure 2.6 (c) Stacked DeepONet, where each branch net has unique network parameters, vs (d) Unstacked DeepONet, where the branch nets share network parameters [17].

and corresponding output function $G(u^{(i)})$ is then evaluated at p different evaluation locations $\{y_j\}_{j=1}^{p}$. Thus the data-set for training a DeepONet $[u,\, y,\, G(u)(y)]$ is of size $(n \times p, m)$, $(n \times p, d)$ and $(n \times p, 1)$ respectively.

However, the learned operator G_θ may not be consistent with the underlying physics (PDE and appropriate IBCs) that generated the data-sets [28]. Additionally, the network outputs $G_\theta(u)(y)$ are differentiable with respect to their inputs $[u(x_1), u(x_2), ..., u(x_m)]^T$ and y, thus allowing for automatic differentiation to be used as a regularization mechanism to bias G_θ to satisfy the underlying PDE constraints. Thus, similar to PINNs [18], the physics-informed DeepONet makes use of the observed data and the physical laws provided by the PDE system, by penalizing a composite loss function L. L is composed of a loss term, L_{data}, representing the observed data while the other, $L_{physics}$, represents the underlying physics.

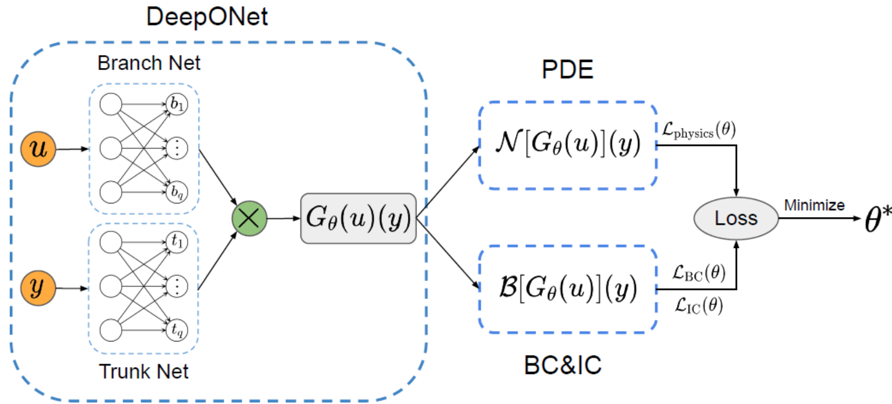


Figure 2.7 Making a DeepONet physics-informed [28], where the composite loss function is a linear combination of $L_{physics}$ representing the underlying physics, and L_{BC}, L_{IC} representing the training data sampled from the initial and boundary conditions.

The loss function for the data-informed DeepONet, L_{data}, is defined as

$$L(G_\theta) = L_{data}(G_\theta) = \frac{1}{np} \sum_{i=1}^{n} \sum_{j=1}^{p} \left| G_\theta(u^{(i)})(y_j^{(i)}) - G(u^{(i)})(y_j^{(i)}) \right|^2, \tag{2.11}$$

which requires the data-set of input-output triplets $[u, y, G(u)(y)]$ obtained by the underlying parametric PDE.

The loss function for the physics-informed DeepONet is defined as

$$L(G_\theta) = L_{data}(G_\theta) + L_{physics}(G_\theta), \tag{2.12}$$

where $L_{physics}$ is given by

$$L_{physics}(G_\theta) = \frac{1}{nqm} \sum_{i=1}^{n} \sum_{j=1}^{q} \sum_{k=1}^{m} \left| N(u^{(i)}(x_k), G_\theta(u^{(i)})(y_j^{(i)}) \right|^2, \tag{2.13}$$

which aims to satisfy the parametric PDE as defined by equation 2.9. L_{data}, in this case, can be defined similarly as above, or with minimal data, i.e. only from the IBCs.

To showcase the inconsistency of the learned operator G_θ with the underlying physics, the researchers in [28] attempted to learn the anti-derivative operator $G : u(x) \rightarrow s(x)$ using only a data-informed DeepONet. The problem was defined by

$$\frac{ds(x)}{dx} = u(x), x \in [0, 1], \tag{2.14}$$

with an initial condition $s(0) = 0$. The loss function, $L = L_{data}$, was defined by equation 2.11, with the data-set generated as per [28].

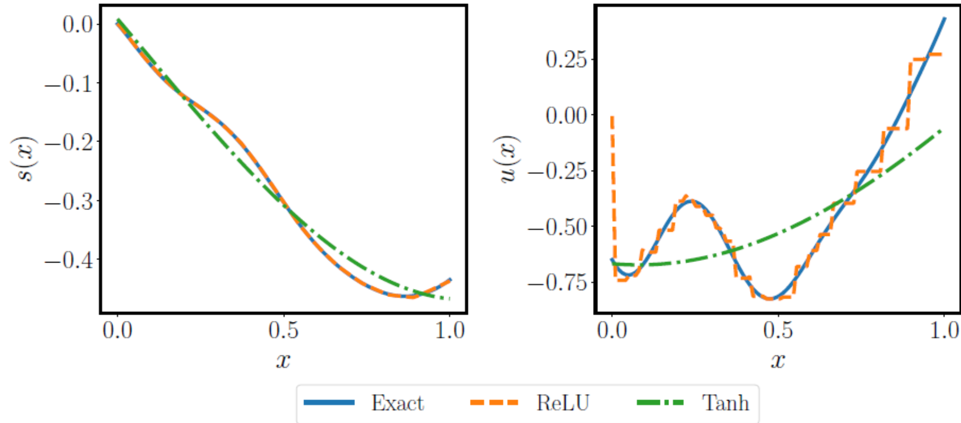


Figure 2.8 Learning the anti-derivative operator by data-informed DeepONet. The predicted solution $s(x)$ (left) and residual $u(x)$ (right) throughout the domain x, are shown for networks trained using different activation functions, ReLU (orange) and Tanh (green), with the ground truth solution and input functions (blue), respectively, provided as reference [28].

As shown in figure 2.8, good agreement between the predicted and the exact solution $s(x)$ is observed with a network using the ReLU activation function (orange), while poor performance is obtained when using the tanh activation function (green). The predicted residual $\frac{ds(x)}{dx}$ for the ReLU network is approximated with the input function $u(x)$ represented by step functions, while the tanh version predictions are largely inaccurate. As a result, both the ReLU and tanh data-informed DeepONet was not able satisfy the anti-derivative problem, even though they were trained using data generated from equation 2.14.

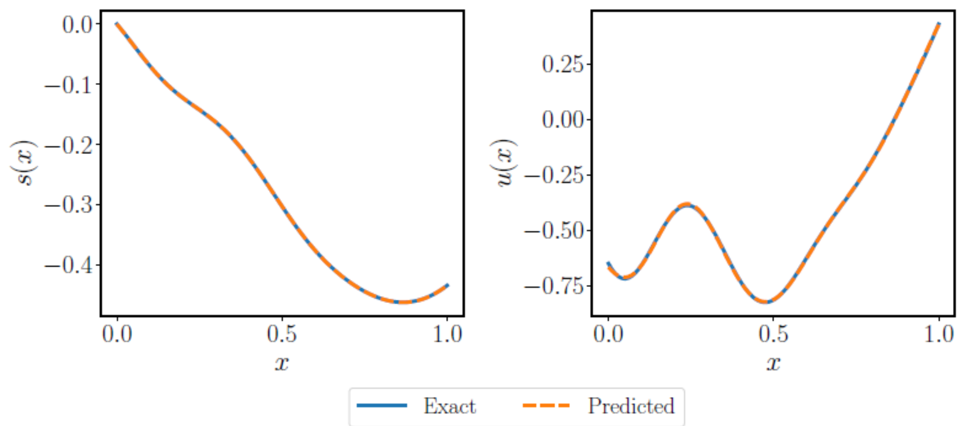


Figure 2.9 Learning the anti-derivative operator with physics-informed DeepONet. The predicted solution $s(x)$ (left) and residual $u(x)$ (right) throughout the domain x, are shown for networks trained using the Tanh activation function (orange), with the ground truth solution and input functions (blue), respectively, provided as reference [28].

By adding the physical constraint, $L_{physics}$, to the loss function L as defined by equation 2.13 and 2.12, respectively, the researchers observed excellent agreement with both $s(x)$ and $u(x)$ when using tanh activation functions as shown in figure 2.9. Thus the physics-informed DeepONet was capable of satisfying equation 2.14.

2.3 Modulus Framework

This section provides a brief overview of the deep learning framework called Modulus that is developed and maintained by NVIDIA for PINNs and Neural Operators (such as DeepONets) [22]. This framework provides various tools to quickly develop machine learning (ML) or NN models that can be applied to physics-based systems described by ODEs and PDEs. For our purposes, we focus on the key aspects necessary for the development of DeepONet models. As this type of network is a new development in the scientific machine learning group, the latest Modulus, version 22.09, provides limited resources for their implementation.

2.3.1 Modulus Building Blocks

Modulus provides several physics-informed and data-driven components that can be used to express the physics-based problem to be simulated. The components used during the development of a standard DeepONet model is listed below.

1. **Geometry and Data**: The physical domain of the problem being solved for can be expressed by either a particular geometric shape or a set of data points. Although the geometry module provided by Modulus allows us to develop complex shapes using basic primitives, the development of DeepONet models are primarily data-driven, and our geometry is created from the data-set.

2. **Nodes**: Nodes are components executed during forward propagation when training, and extend the torch.nn.Module class from Pytorch [23] (which is Pytorch's base class for NN models). As a result, Modulus is able to construct the computational graphs of our physics-based system and compute the required derivatives needed for backward propagation (refer to section 2.1.1). We use Nodes to generate the branch and trunk networks for the DeepONet models, as well as, to represent the heat PDE and more complex BCs such as Neumann and Robin BCs (see section 3.1.2), which are not easily implemented in Modulus for physics-based DeepONets.

3. **Constraints**: Constraints are used to represent the training objectives that comprise our loss function in Modulus. These objectives represent the ground-truth data $G(u)(y)$ that the network is predicting $G_\theta(u)(y)$, as well as, the PDE and IBCs that comprise our loss function (refer to section 2.2.1). Constraints are necessary to properly define the physical problem.

4. **Domain**: The Domain stores the Constraints, and additional Modulus components used during the training process. The additional components used in this thesis include Inferencers, Validators, and Monitors.

5. **Solver**: The Solver is an instance of the Modulus trainer that manages the optimization loop during the training process (refer to section 2.1.1). It uses the Domain to call the various components it has stored, such as Constraints, Inferencers, Validators, and Monitors, when required. During each training iteration, the Solver computes the total loss from all Constraints, then optimizes the network parameters in the Nodes that were passed to the Constraints.

6. **Validators**: Validators perform only the forward pass on a set of Nodes during training using only the validation data. They are used to assess the accuracy of the model during training by validating the network's prediction $\hat{y}$ against the ground truth data y.

7. **Monitors**: Monitors also perform only the forward pass on a set of Nodes during training, but are used to calculate specific measures, such as the PDE residual. For example, we use it to determine how well the heat PDE constraint is satisfied during training using training quantities, such as u_t and u_{xx}, for example.

8. **Inferencers**: An Inferencer also performs only the forward pass on a set of Nodes during training. It is used to monitor certain training quantities, such as the temperature derivatives, u_t and u_{xx}, for example.

9. **Hydra**: Hydra is a configuration package required by Modulus [19]. It is used to set various hyper-parameters that govern the network's training, such as the scheduler and loss function, using a YAML configuration files.

2.3.2 Modulus Workflow

A complete workflow for developing a NN model for physics-based systems is shown in figure 2.10, where N_c, N_i, N_v and N_m represent the number of Constraints, Inferences, Validators and Monitors used in the network's development.

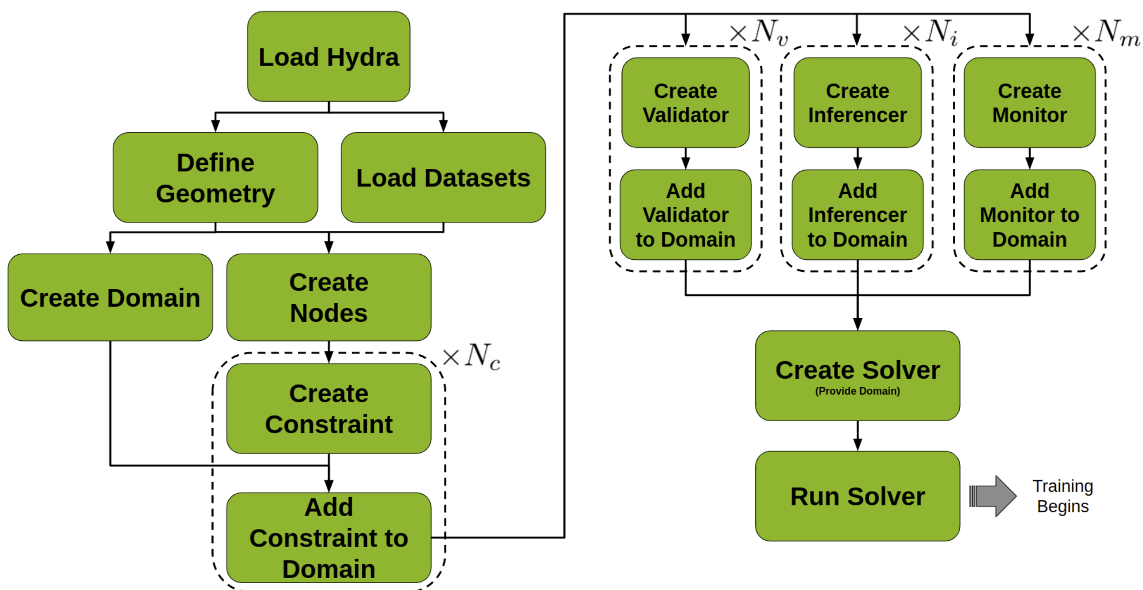


Figure 2.10 A typical workflow for developing NN in Modulus [22].

2.3.3 Activation Functions

Modulus provides different activation functions (AFs) that can be used to introduce non-linearity into NNs. As discussed in section 2.1.1, AFs are crucial in allowing a NN to learn abstract features of the input by performing non-linear transformations between the layers. In this study, we investigate the choice of AFs on the performance of DeepONets, with a particular interest on the physics-informed variant as embedding the physics in $L_{physics}$ requires higher derivative terms. Further details are provided in section 3.2.7.

In this section, we provide a summary of the formulas for the different AFs used in the study in table 2.1. An illustration of each function, along with its first derivative can be seen in Appendix A.1.2.

Table 2.1 Formulas for the different AF provided in Modulus [22].

Name	**Formula**
Rectified Linear Unit (ReLU)	$R(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases}$
Leaky ReLU where $\alpha = 0.01$ is the default hyper-parameter.	$R(z) = \begin{cases} z & z > 0 \\ \alpha z & z \leq 0 \end{cases}$
Exponential Linear Unit (ELU) [4] where α is a neuron-wise parameter to be optimized.	$R(z) = \begin{cases} z & z > 0 \\ \alpha(e^z - 1) & z \leq 0 \end{cases}$
Scaled Exponential Linear Units (SELU) [12] where $\alpha = 1.673$ and $\lambda = 1.051$ are the default parameters from [12], that are optimized during training.	$R(z) = \begin{cases} \lambda z & z \geq 0 \\ \lambda \cdot \alpha(e^z - 1) & z < 0 \end{cases}$
Sigmoid	$R(z) = \frac{1}{1 + e^{-z}}$
Gaussian Error Linear Unit (GELU) [7] where $\Phi(z) = P(Z \leq z), Z \sim N(0, 1)$ is the cumulative distribution function of the standard normal distribution.	$R(z) = z \cdot \Phi(z)$
Tanh	$R(z) = tanh(z)$
Softplus	$R(z) = ln(e^z + 1)$
Mish [20]	$R(z) = z \cdot tanh \cdot ln(1 + e^z)$
Self-scalable Tanh (Stan) [24] where β is a neuron-wise parameter to be optimized.	$R(z) = tanh(z) + \beta \cdot z \cdot tanh(z)$
Sigmoid Linear Units (SiLU) [6]	$R(z) = z \cdot \frac{1}{1 + e^{-z}}$
Sin	$R(z) = sin(z)$
Squareplus where b is a neuron-wise parameter to be optimized.	$R(z) = \frac{1}{2}(z + \sqrt{z^2 + b})$

3 Solving Heat Conduction Problems with DeepONets

In this section, we test the effectiveness of DeepONets to learn the heat conduction operator by performing a series of numerical studies on different heat conduction problems. We create both a data-informed and a physics-informed DeepONet model, which we will refer to by DI-DeepONet and PI-DeepONet respectively.

3.1 Heat Conduction Equation

The heat conduction equation is a PDE that describes the temperature field in a body [5]. The general form of the heat equation in one spatial dimension bounded in the interval $[a, b]$ is given by

$$\rho(x)C_p(x)\frac{du}{dt} = \frac{d}{dx}\left(k(x)\frac{du}{dx}\right) + \dot{Q}(x,t), x \in (a,b), t > 0, \tag{3.1}$$

where $u(x,t)$ is the temperature at any point x at time t, $\rho(x)$ is the density, $C_p(x)$ is the specific heat capacity (the amount of heat energy required to raise one unit of mass by one unit of temperature), $k(x)$ is the coefficient of thermal conductivity (the ability to conduct heat), and $\dot{Q}$ is the internal heat generation rate per unit volume. Note that equation 3.1 can easily be extended to higher spatial dimensions. Additionally, C_p, ρ and k are functions of x, but can also be functions of u or t, and that $\dot{Q}$ is a function of both x and t.

The examples presented in this thesis uses uniform C_p, ρ and k. Thus equation 3.1 can then be simplified to

$$\rho C_p\frac{du}{dt} = k\frac{d^2u}{dx^2} + \dot{Q}(x,t). \tag{3.2}$$

Equation 3.2 represents a one-dimensional transient case. However, a steady-state problem can also be defined, where the transient term $\frac{du}{dt}$ is set to 0. Equation 3.2 thus becomes

$$-k\left(\frac{d^2u}{dx^2}\right) = \dot{Q}(x,t). \tag{3.3}$$

Note that the term, $\frac{k}{\rho c}$, is referred to as thermal diffusivity, α (rate of heat transfer of a material from the hot region to the cold region).

3.1.1 Initial and Boundary Conditions (IBCs)

In order to solve the transient heat equation, appropriate IBCs must be specified for the temperature field $u(x,t)$. The initial conditions (ICs) specifies the values of u at the initial time $t = 0$, while the boundary conditions (BCs) specifies some condition on u at the boundary of the domain $x \in [a,b]$. For example, the BCs on the interval $[a,b]$ may take the form

$$u(a,t) = 0 \text{ and } u(b,t) = 0, \text{for } t > 0, \tag{3.4}$$

and the ICs may be specified by a function $f(x)$ defined only at $t = 0$, i.e.

$$u(x, 0) = f(x). \tag{3.5}$$

The solution $u(x, t)$ must therefore satisfy the PDE equation (3.1, 3.2 or 3.3) in the entire domain ($x \in [a, b]$ and $t > 0$), as well as, the IBCs. The solution domain for this one-dimensional example can be interpreted by having three sides: $x = a, x = b$, and $t = 0$, with the top side open as the solution progresses i.e. $t \to \infty$. An illustration is shown in figure 3.1.

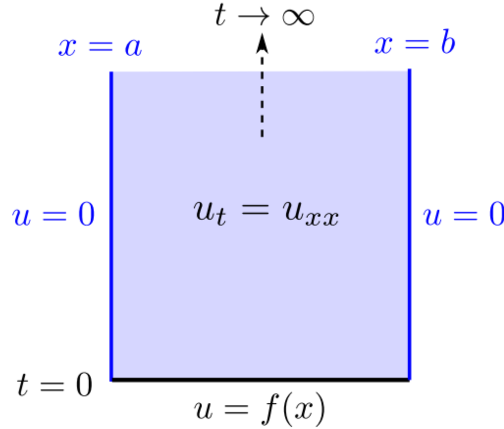


Figure 3.1 Illustration of the Solution Domain for the One-Dimensional Heat Equation [30]. The boundary conditions, $u(a, t) = 0$ and $u(b, t) = 0$ are shown in blue, the initial conditions, $u(x, 0) = f(x)$ is shown in black, while the PDE constraint is imposed on the inner domain.

3.1.2 Types of Boundary Conditions

There are three basic types of BCs that we considered when solving the series of heat equation problems in this thesis. These are:

1. **Dirichlet**: Specifies the value that the unknown temperature field u takes on the boundary of the domain Γ_d given by

   $$u(x, t) = g(x), \forall x \in \Gamma_d, \tag{3.6}$$

   where $g(x)$ is a scalar function defined on Γ_d.

2. **Neumann**: Specifies the value that the derivative of the temperature field u takes on the boundary of the domain Γ_n given by

   $$\hat{n} \cdot \left(k\frac{du}{dx}\right) = g(x), \forall x \in \Gamma_n, \tag{3.7}$$

   where $g(x)$ is a scalar function defined on Γ_n and $\hat{n}$ is the unit normal to the boundary Γ_n. It is commonly used to represent an insulated boundary where $g(x) = 0$.

3. **Robin**: Consists of a linear combination of the values of the temperature field u and its derivatives on the boundary of the domain Γ_r. In the context of the heat equation, it can be referred to as the convective boundary condition and is given by

   $$\hat{n} \cdot \left(k\frac{du}{dx}\right) = h(u_{amb} - u), \forall x \in \Gamma_r, \tag{3.8}$$

   where $\hat{n}$ is the unit normal to the boundary Γ_r, h is the convective heat transfer coefficient, and u_{amb} is the ambient temperature.

3.1.3 Heat Operator

The heat conduction equation is a linear equation for u, and contains a linear operator G that our DeepONet models will attempt to learn. Equation 3.1, for example, can be reduced to its operator form as

$$G(u) = \dot{Q}(x,t), x \in (a,b), t > 0, \tag{3.9}$$

where $G(u)$ is given as

$$G(\bullet) = \rho(x) C_p(x) \frac{d\bullet}{dt} - \frac{d}{dx}\left(k(x)\frac{d\bullet}{dx}\right). \tag{3.10}$$

3.1.4 Test Error Analysis

We compare the predicted solutions $\hat{y}$ from each DeepONet model, to the true ones y obtained from the analytical equation, where available, or from a highly accurate numerical solver, using the relative root mean squared error (RRMSE) [2] as our evaluation metric. The RRMSE is defined as the root mean squared error (RMSE) of the predicted and true solution, $\hat{y}_i$ and y_i respectively, divided by the average value of the predicted solution $\hat{y}_i$, i.e.

$$RRMSE = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2}}{\frac{1}{n}\sum_{i=1}^{n}\hat{y}_i}. \tag{3.11}$$

The RMSE is sensitive to outliers, where a few terrible y samples or predictions $\hat{y}$ can result in a larger loss, thereby forcing the model to learn outliers more rather than the majority. The RMSE is also dependent on the scale of each y sample, increasing in magnitude if the scale of the error increases. This is undesirable when the errors for lower scaled y samples have less effect on the model's predictions when compared to errors for higher scaled y samples. To mitigate these effects, we divide the RMSE by the average value of the predicted solution $\hat{y}_i$, making the models more robust to outliers and scale-independent.

3.2 1D Transient Heat Equation

3.2.1 Problem Definition

The heat conduction problem presented in this section is inspired by an example from DeepXDE [15]. We employ both a DI-DeepONet and a PI-DeepONet to learn the heat operator G given by the heat equation with a parametric thermal diffusivity α

$$\frac{du}{dt} = \alpha \frac{d^2u}{dx^2}, x \in [0,1], t \in [0,1]. \tag{3.12}$$

Additionally, each DeepONet must also satisfy the Dirichlet BCs

$$u(0,t) = u(1,t) = 0, \tag{3.13}$$

and periodic (sinusoidal) ICs

$$u(x,0) = sin(n\pi xL), 0 < x < L, n = 1, 2, ..., \tag{3.14}$$

where L is the length of the bar, n is the frequency of the sinusoidal IC.

The exact solution is given by

$$u(x,t) = e^{\frac{-n^2\pi^2\alpha t}{L^2}} \cdot sin(\frac{n\pi x}{L}). \tag{3.15}$$

The parameters used for defining the heat conduction PDE are shown in table 3.1. α will represent a spatially constant input function for the branch net of the DeepONet models, and takes a value within the specified range.

Table 3.1 PDE parameters for the heat conduction problem given by equation 3.12.

Parameter	Value
α	0 - 2
n	1
L	1

The aim is to learn the heat operator G that maps α to the corresponding PDE solutions $u(x,t)$, i.e. $G : \alpha \to G(\alpha)$, where $G(\alpha) = u(\alpha)$, and for any (x,t) in the domain, $G(\alpha)(x,t) = u(x,t)$.

3.2.2 Data-set Generation

The training and test data-sets consists of input-output triplets $[\alpha, (x,t), G(\alpha)((x,t))]$, where α is a constant function, (x,t) is an evaluation location within the problem domain and $G(\alpha)((x,t)) = u(x,t)$ is the temperature value to be predicted by the network for the given α at the (x,t) coordinate.

To generate the training and test data-sets, we first sample $2 \times n = 1000$ different α values uniformly between the range specified in table 3.1. Then for each α, we use the formula for exact solution (equation 3.15) to generate the temperature solutions u for different (x,t) locations. The evaluation locations, (x,t), to be passed to the trunk network during training were randomly selected for $p = 100$ for the residual, $p = 100$ for the ICs, as well as, $p = 100$ for the Dirichlet BCs. Each temperature solution $u^{(i)}$ is then obtained for each randomly generated collocation point $(x^{(i)}, t^{(i)})$.

In summary, for the DI-DeepONet, for each $\alpha^{(i)}$,

- $\{(x_{r,j}^{(i)}, t_{r,j}^{(i)})\}_{j=1}^{p=100}$ are uniformly sampled within the domain $[0,1] \times [0,1]$ (residual),
- $\{(x_{b,j}^{(i)}, t_{b,j}^{(i)})\}_{j=1}^{p=100}$ are uniformly sampled from the left and right boundaries, each, at times $[0,1]$ (Dirichlet),
- $\{(x_{ic,j}^{(i)}, 0)\}_{j=1}^{p=100}$ are uniformly sampled within the domain $[0,1]$ at time $t=0$ (IC).

This data-set is used when computing the data loss, $L_{data}(\theta)$ (see section 3.2.3). Thus the training data-set contains $[\alpha, (x,t), G(\alpha)(x,t)]$ of size $(1000 \times 400, 1)$, $(1000 \times 400, 2)$ and $(1000 \times 400, 1)$ respectively.

For generating the training data-set for the PI-DeepONet,

- $\{(x_{b,j}^{(i)}, t_{b,j}^{(i)})\}_{j=1}^{p=100}$ are uniformly sampled from the left and right boundaries, each, at times $[0,1]$ (Dirichlet),
- $\{(x_{ic,j}^{(i)}, 0)\}_{j=1}^{p=100}$ are uniformly sampled within the domain $[0,1]$ at time $t=0$ (IC).

In order to satisfy the data loss, $L_{data}(\theta)$. The training data-set contains $[\alpha, (x,t), G(\alpha)(x,t)]$ of size $(1000 \times 300, 1)$, $(1000 \times 300, 2)$ and $(1000 \times 300, 1)$, for the IBCs, and is only required for computing $L_{data}(\theta)$.

Additionally, we need to randomly select $q = 100$ different collocation locations, (x,t), to satisfy the PDE residual for $L_{physics}$ (see sections 3.2.3 and 3.2.3). Thus,

- $\{(x_{r,j}^{(i)}, t_{r,j}^{(i)})\}_{j=1}^{q=100}$ are collocation points sampled within the domain $[0,1] \times [0,1]$ for computing the residual loss, $L_{physics}(\theta)$, to satisfy the PDE constraint.

For testing, we use the entire domain $[0,1] \times [0,1]$, each possessing 200 equally spaced collocation points in $[0,1]$. The testing data-set contains $[\alpha, (x,t), G(\alpha)(x,t)]$ of size $(1000 \times 40000, 1)$, $(1000 \times 40000, 2)$ and $(1000 \times 40000, 1)$ respectively.

The parameters used for generating the data-sets are shown in table 3.2.

Table 3.2 Parameters used for generating the training and test data-sets.

Parameter	DI-DeepONet	PI-DeepONet
n	1000	1000
p	400	300
q	-	100

Note that the DI-DeepONet is trained on paired input-output measurements within the domain, as well as, for the IBCs, but the PI-DeepONet is trained without any paired training data (except for the IBCs). As a result, the PI-DeepONet contains 25% less training data (for predicting $G(\alpha)(x,t)$ within the domain) when compared to the DI-DeepONet. Additionally, as the IBCs are known a priori, the training data for the PI-DeepONet can be assembled without first solving time-consuming FEA models or sensor measurements (in cases where an analytical solution is not available). However, this is not possible with the DI-DeepONet as the temperatures, u, within the domain at randomly generated collocation points, (x_r, t_r), are needed. Thus, $G(\alpha)(x,t)$ in the training data for the ICs and Dirichlet BCs are not obtained from the analytical solution, as its value is already known.

3.2.3 Loss Function

Recall that G_θ represents the approximation of the heat operator G by both DeepONet variants. The loss function for the DI-DeepONet is given by

$$L(\theta) = L_{data}(\theta) = L_{r,data}(\theta) + L_{dc,data}(\theta) + L_{ic,data}(\theta), \tag{3.16}$$

to satisfy the constraints for the residual r, the Dirichlet dc, and the IC ic respectively. Each term is given by

$$L_{r,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(\alpha^{(i)})(x_{r,j}^{(i)}, t_{r,j}^{(i)}) - G(\alpha^{(i)})(x_{r,j}^{(i)}, t_{r,j}^{(i)})\right|^2, \tag{3.17}$$

$$L_{dc,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(\alpha^{(i)})(x_{dc,j}^{(i)}, t_{dc,j}^{(i)}) - G(\alpha^{(i)})(x_{dc,j}^{(i)}, t_{dc,j}^{(i)})\right|^2, \tag{3.18}$$

$$L_{ic,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(\alpha^{(i)})(x_{ic,j}^{(i)}, t_{ic,j}^{(i)}) - G(\alpha^{(i)})(x_{ic,j}^{(i)}, t_{ic,j}^{(i)})\right|^2. \tag{3.19}$$

Let R_θ be the heat PDE residual which is given by

$$R_\theta^{(i)}(x,t) = \frac{dG_\theta(\alpha^{(i)})(x,t)}{dt} - \alpha^{(i)}\frac{d^2G_\theta(\alpha^{(i)})(x,t)}{dx^2}. \tag{3.20}$$

The loss function for the PI-DeepONet can then be defined as

$$L(\theta) = L_{data}(\theta) + L_{physics}(\theta),$$

with $L_{physics}(\theta)$ as

$$L_{physics}(\theta) = \frac{1}{nq}\sum_{i=1}^{n}\sum_{j=1}^{q}\left|R_\theta^{(i)}(x_j^{(i)}, t_j^{(i)})\right|^2, \tag{3.21}$$

and $L_{data}(\theta)$ is composed of only the collocation points sampled on the boundaries of the domain for the Dirichlet and ICs, i.e.

$$L_{data}(\theta) = L_{dc,data}(\theta) + L_{ic,data}(\theta). \tag{3.22}$$

Defining PDE Loss, $L_{physics}(\theta)$

In listing 1, we define the PDE loss $L_{physics}(\theta)$ in Modulus using a torch.nn.Module, and a class called HeatPDE, which uses the input and output tensors of our PI-DeepONet to compute the residual R_θ within its forward method. This module is then incorporated into the computational graph created by Modulus using a Node as shown in listing 2. The PDE loss $L_{physics}(\theta)$ is then added as an additional constraint using Modulus' DeepONetConstraint as shown in listing 3.

```python
class HeatPDE(torch.nn.Module):
    "Custom 1-dimensional Heat PDE definition for PI-DeepONet"

    def __init__(self):
        super().__init__()

    def forward(self, input_var: Dict[str, torch.Tensor])
        -> Dict[str, torch.Tensor]:
        # Get inputs
        u = input_var["u"]
        alpha = input_var["alpha"]
        # Compute gradients
        dudt = input_var["u__t"]
        dduddx = input_var["u__x__x"]
        # Compute Heat equation
        heat = (dudt - alpha*dduddx)

        # Return heat
        output_var = {
            "heat": heat,
        }
        return output_var
```

Listing 1 Defining the Heat PDE in Modulus.

```python
# Incorporate Heat module into Modulus using a Node
heat_node = Node(
    inputs=["u", "alpha", "u__t", "u__x__x"],
    outputs=["heat"],
    evaluate=HeatPDE(),
    name="Heat Node",
)

nodes = [deeponet.make_node('deepo'), heat_node]
```

Listing 2 Adding the Heat Node into the computational graph.

```python
residual = DeepONetConstraint.from_numpy(
    nodes=nodes,
    invar={
        "alpha": a_r_train,
        "x": x_r_train,
        "t": t_r_train,
    },
    outvar={
        "heat": np.zeros_like(u_r_train),
    },
    batch_size=cfg.batch_size.interior,
    lambda_weighting={
        "heat": 0.01*np.ones_like(u_train),
    },
)
domain.add_constraint(residual, "Residual")
```

Listing 3 Adding the Heat PDE loss $L_{physics}(\theta)$ as an additional constraint.

Defining the Dirichlet BCs in Modulus

The Dirichlet BCs for the left and right boundaries, and the ICs are easily added as constraints using Modulus's DeepONetConstraint for the DI-DeepONet and PI-DeepONet. The heat PDE residual for the DI-DeepONet, is similarly enforced. An example is shown in listing 4 for the left boundary.

```
# Dirichlet boundary condition 1
DBC_1 = DeepONetConstraint.from_numpy(
    nodes=nodes,
    invar={
        "alpha": a_train,
        "x": np.zeros_like(x_train),
        "t": t_train,
    },
    outvar={"u": np.zeros_like(u_train)},
    batch_size=cfg.batch_size.BC,
    lambda_weighting={"u": 1.0*np.ones_like(u_train)},
)
domain.add_constraint(DBC_1, "DBC_1")
```

Listing 4 Adding the Dirichlet BC constraint $L_{dc,data}(\theta)$ in Modulus.

3.2.4 Training

Recall (refer to section 2.2.1) that the branch network takes α as input and returns a features embedding vector br as output, and the trunk network takes x and t as input and returns a features embedding vector tr as output. These outputs are then merged together by a dot product to produce the final prediction of the heat operator by each DeepONet, i.e. $G_\theta = br \cdot tr$. The network architecture and training parameters used are summarized in table 3.3.

Table 3.3 Network architecture and training parameters used by each DeepONet.

Name	Value
Architecture	Fully-connected Neural Network
Depth	4
Width	64
Optimiser	Adam
Learning Rate	1.0×10^{-3} with decay
Training Steps	100000
Activation Function	TANH (DI-DeepONet) / SILU (PI-DeepONet)

Training

The PI-DeepONet loss function L_θ is computed in Modulus as a sum of point-wise differences (equation 2.2) for:

- Predicting the temperature u for the IBCs, as given by the data loss L_{data},
- Predicting the PDE residual for the heat equation, R_θ, as given by the physics loss $L_{physics}$.

As a result, we apply lambda weightings to the loss function for each objective, i.e.

$$L(\theta) = \lambda_{data} L_{data}(\theta) + \lambda_{physics} L_{physics}(\theta), \tag{3.23}$$

Table 3.4 Lambda weightings used by the PI-DeepONet.

Name	Value
λ_{data}	1.0
$\lambda_{physics}$	0.01

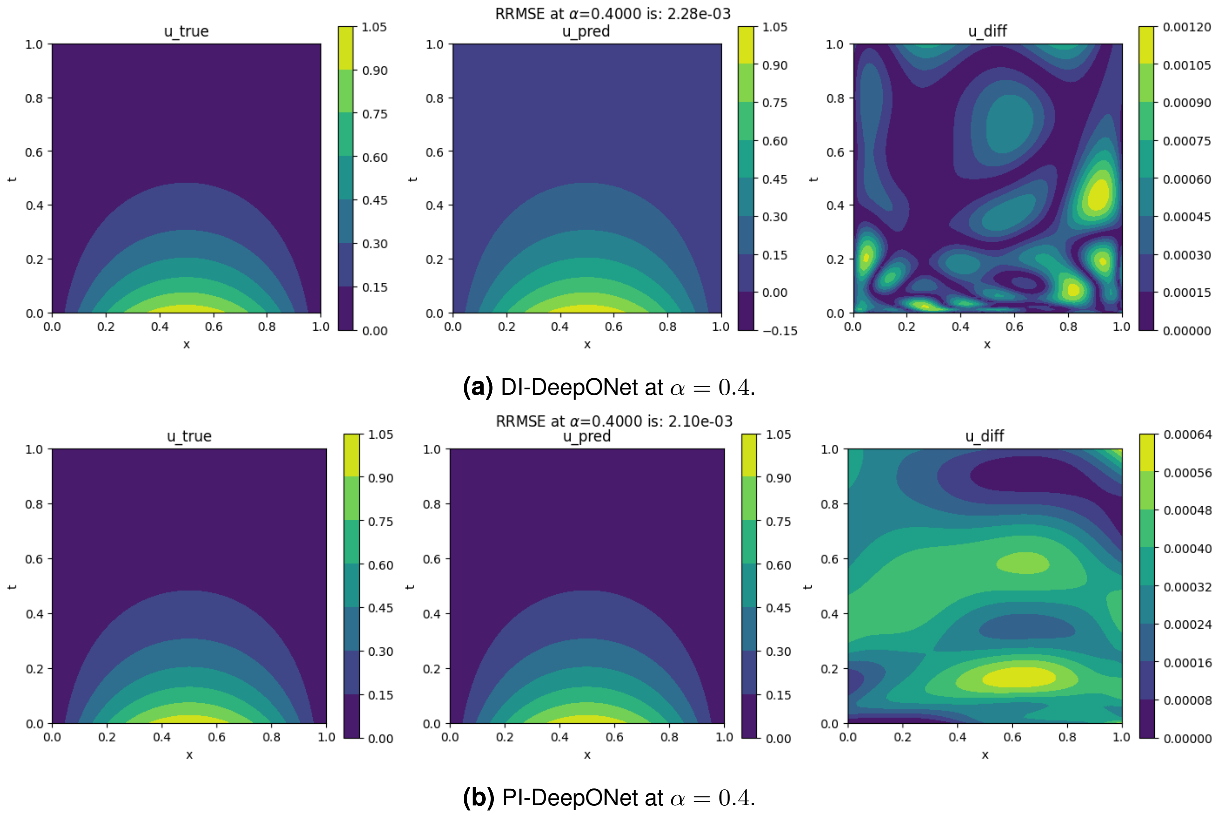
where λ_{data} and $\lambda_{physics}$ are given in table 3.4.

These lambda values were determined by first training the network with $\lambda_{data} = 1.0$ and $\lambda_{physics} = 1.0$. The ratios between the magnitudes of the different loss terms, L_{data} or $L_{physics}$, were then computed to determine which term was dominating the overall loss. These ratios are then applied as the lambda weightings. These weightings are useful in training the network as it ensures that no one loss term dominates the parameter updates, and that each term contributes equally.

The DI-DeepONet loss function L_θ is computed as a sum of point-wise differences for predicting the temperature u for the IBCs, as well as, within the domain, as given by the data loss L_{data}. However, we apply uniform lambda weightings to the loss function for each objective.

3.2.5 Results

The temperature field $\hat{u}$ predicted by the DI-DeepONet and PI-DeepONet are compared to the true u given by the analytical solution (equation 3.15) using Validators from Modulus. The results are shown for $\alpha = 0.4$ for both the DI-DeepONet and the PI-DeepONet as shown in figure 3.2. The RRMSE for the validation data-set at 6 different α values are also provided in table 3.5.



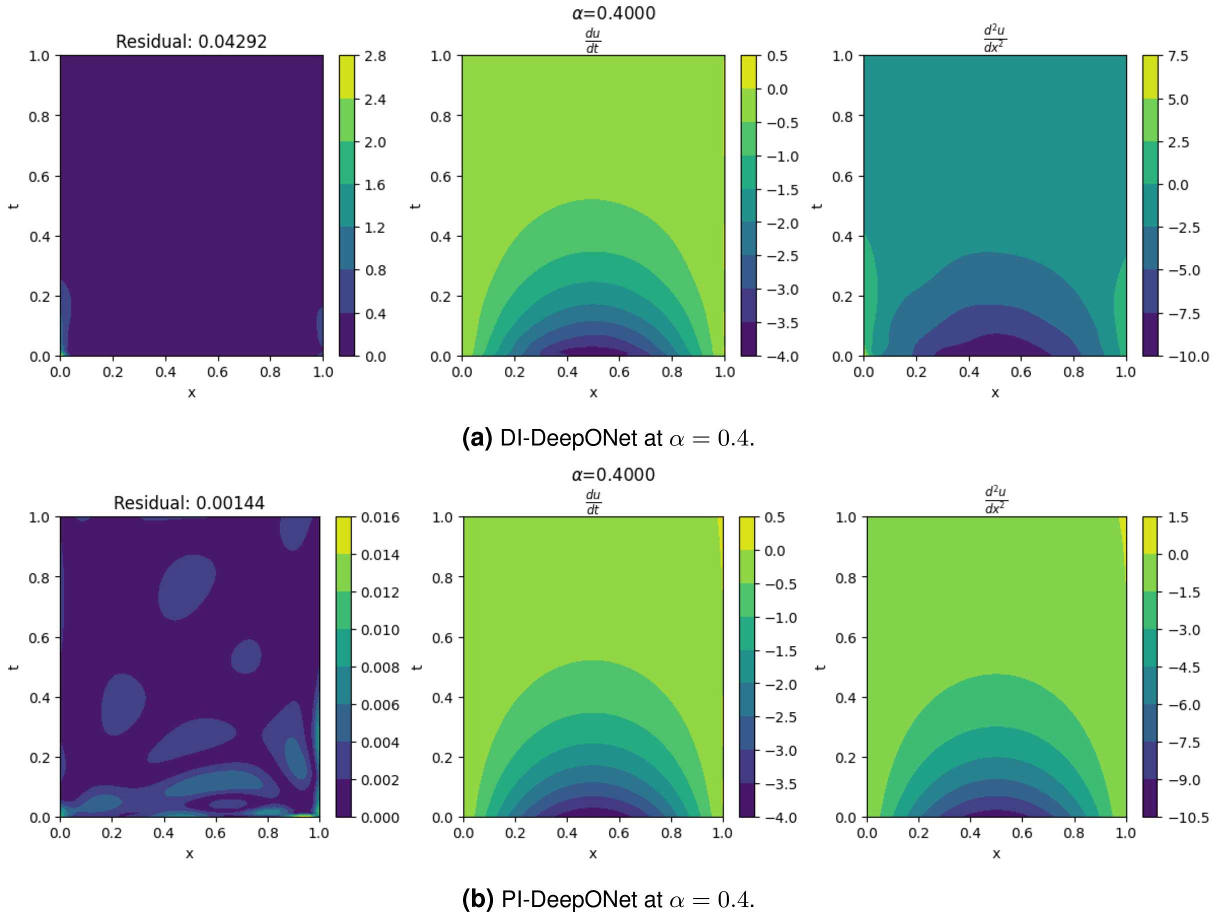
(a) DI-DeepONet at $\alpha = 0.4$.

(b) PI-DeepONet at $\alpha = 0.4$.

Figure 3.2 Temperature Field Predicted by DI-DeepONet (top) and PI-DeepONet (bottom), with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).

Table 3.5 RRMSE for DI-DeepONet and PI-DeepONet at 6 different α.

α	**DI-DeepONet**	**PI-DeepONet**
0.2704	1.94×10^{-3}	1.57×10^{-3}
0.4000	2.28×10^{-3}	2.10×10^{-3}
0.6694	3.42×10^{-3}	1.86×10^{-3}
0.8582	3.43×10^{-3}	2.27×10^{-3}
1.2682	7.25×10^{-3}	1.37×10^{-3}
1.7086	8.46×10^{-3}	3.88×10^{-3}



(a) DI-DeepONet at $\alpha = 0.4$.

(b) PI-DeepONet at $\alpha = 0.4$.

Figure 3.3 PDE Residual (left), u_t (middle), and u_{xx} (right) predicted by DI-DeepONet (top) and PI-DeepONet (bottom) at $\alpha = 0.4$.

We observe that both DeepONets predict the temperature field u with reasonable accuracy, however, the PI-DeepONet performs better than the DI-DeepONet for each α. Additionally, the prediction accuracy of the DI-DeepONet decreases consistently as α increases, which is not observed for the PI-DeepONet. As α increases, the temperature u changes more rapidly over time producing larger temperature gradients u_t which may be more difficult for the DI-DeepONet model to learn and predict.

We then compared how well the PDE residual, R_θ (3.20), is satisfied by each DeepONet. As described in section 3.2.3, the PI-DeepONet is trained with R_θ as an additional objective, which acts as a regularization mechanism that biases the network's output to satisfy this PDE constraint, while the DI-DeepONet is trained only using data. The PDE residual is satisfied more accurately by the PI-DeepONet, res_{phys}, by 2 orders of magnitude better than the DI-DeepONet, res_{data}, as shown in figure

3.3. An Inferencer was used to track the PDE partial derivatives, u_t and u_{xx} within the domain during training, in order to produce these plots.

u_t and u_{xx} over $x \in [0, 1]$ and $t \in [0, 1]$ is also provided for each DeepONet. We observe that u_t and u_{xx} have a more similar shape for the PI-DeepONet, as compared to the DI-DeepONet. By comparison, the DI-DeepONet is unable to adequately predict u_{xx}, particularly at early time steps where we expect higher temperature gradients. Recall that $L_{physics}$ (3.21) for the PI-DeepONet is composed of these gradient terms for satisfying the heat PDE residual, R_θ (3.20), while $L_{r,data}$ (3.17) for the DI-DeepONet is composed of only training data, and not directly related to these gradients. As a result, the inability of the DI-DeepONet to adequately satisfy u_{xx} is expected.

3.2.6 Data Quality

The training and test data-sets consists of input-output triplets $[\alpha, (x, t), G(\alpha)((x, t))]$. In our previous analysis, $G(\alpha)((x, t)) = u(x, t)$ is the temperature value determined using the analytical solution (equation 3.15). However, to assess the impact of the quality of our training data-set on the DI-DeepONet, we use the backward time central difference (BTCS) scheme on a 200×200 fine grid to generate the temperature solutions u for different x and t locations. The collocation points, $\alpha^{(i)}$, $\{(x_{d,j}^{(i)}, t_{d,j}^{(i)})\}_{j=1}^{p=100}$ and $\alpha^{(i)}$, $\{(x_{b,j}^{(i)}, t_{b,j}^{(i)})\}_{j=1}^{p=100}$ are similarly sampled as described in section 3.2.2. Each temperature solution $u^{(i)}$ is then obtained from the fine grid solution by cubic interpolation for each randomly generated collocation point.

We then compared the results for the networks trained using the analytical solution and the BTCS scheme as shown in figures 3.2a and 3.4 respectively. We observed that the model trained using the analytical solution provides better prediction accuracy than the one trained using the numerical solution. This may be due to the network learning a different solution operator $\hat{G}$ than the true solution operator G. As a result, special care should be taken when computing the training data numerical, in cases where an analytical solution is not available.

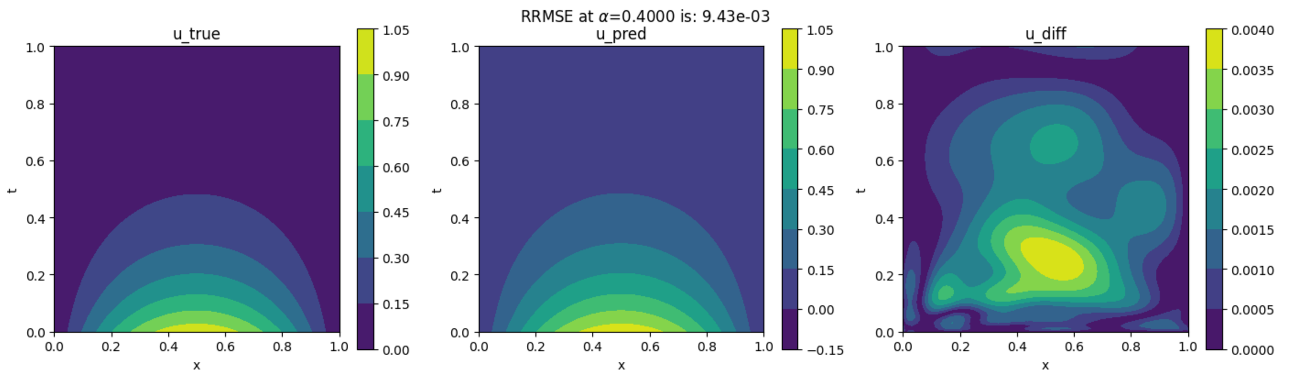


Figure 3.4 Temperature Field Predicted by DI-DeepONet at $\alpha = 0.4$ when trained using Numerical data, with the True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).

3.2.7 Activation Functions (AFs)

In this section, we study the effects on the choice of different AFs, provided by Modulus (refer to section 2.3.3), on the prediction accuracy and training efficiency on the DI-DeepONet and PI-DeepONet. The architecture of the DeepONets is kept the same, except for the choice of AF applied to both the branch and trunk networks. We then compute the RRMSE to assess the performance of each DeepONet defined

using the various AFs.

The PI-DeepONet has a loss term, $L_{physics}(\theta)$ (equation 3.21), that directly embeds the physical constraints given by the heat PDE (3.12). Computing $L_{physics}(\theta)$ requires the network to compute the derivatives of the output with respect to the inputs , x and t, (by automatic differentiation) i.e. $\frac{dG_\theta}{dt}$ and $\frac{d^2G_\theta}{dx^2}$ respectively, to determine R_θ (3.20) during each training iteration i. In contrast, the DI-DeepONet has a similar term, $L_{r,data}(\theta)$ (equation 3.17), that indirectly satisfies the heat PDE. It is computed as the sum of point-wise difference for predicting u directly through training data. This term is determined without the need to compute the derivatives $\frac{dG_\theta}{dt}$ and $\frac{d^2G_\theta}{dx^2}$.

Recall that the gradients of the loss function are used in back-propagation (see section 2.1.1) for the optimization of model parameters, θ. As a result, the PI-DeepONet requires the second and third derivatives, $\frac{d^2G_\theta}{dt^2}$ and $\frac{d^3G_\theta}{dx^3}$, while the DI-DeepONet only requires the first derivatives, $\frac{dG_\theta}{dt}$ and $\frac{dG_\theta}{dx}$, during optimisation. A poorly chosen AF can cause (1) loss of information of the input during forward propagation (see section 2.1.1) and (2) vanishing or exploding gradients during back-propagation, which results in poor convergence.

Ranking the Activation Functions

For each AF, we compute the RRMSE of the differences between the network's predicted temperature, u_{pred}, and true temperature, u_{true}, i.e. $u_{diff} = np.abs(u_{pred} - u_{true})$, made using DeepONets at 10 different α values as input to the branch network. This metrics is then used to rank the AFs in terms of predictions over the entire domain.

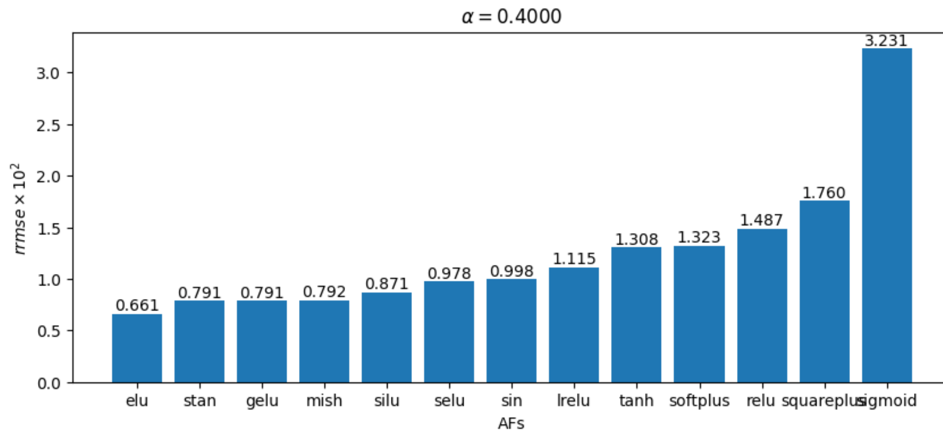
The RRMSE is visualized using a bar graph at $\alpha = 0.4$ in figure 3.5 for the DI-DeepONet and PI-DeepONet. For the DI-DeepONet, we observed that squareplus and sigmoid AFs produce the highest RRMSE consistently across 10 different α values, where as, the remaining AFs produced lower RRMSEs, with no set of AFs performing best across the set. For the PI-DeepONet, we observed that stan, mish and silu AFs produce the lowest RRMSE consistently across 10 different α values, where as, relu, lrelu, selu and sigmoid AFs produce the highest RRMSE consistently. The remaining AFs produce satisfactory RRMSE.

The range for the RRMSE for a DI-DeepONet trained with the best and worst AFs is 1 order of magnitude less, when compared to the same for the PI-DeepONet. Thus, the PI-DeepONet is more susceptible to performance variations based on the choice of AF used for the branch and trunk networks. Additional RRMSE bar graphs at 5 different α between the range given in table 3.1, is provided in Appendix A.1.2.

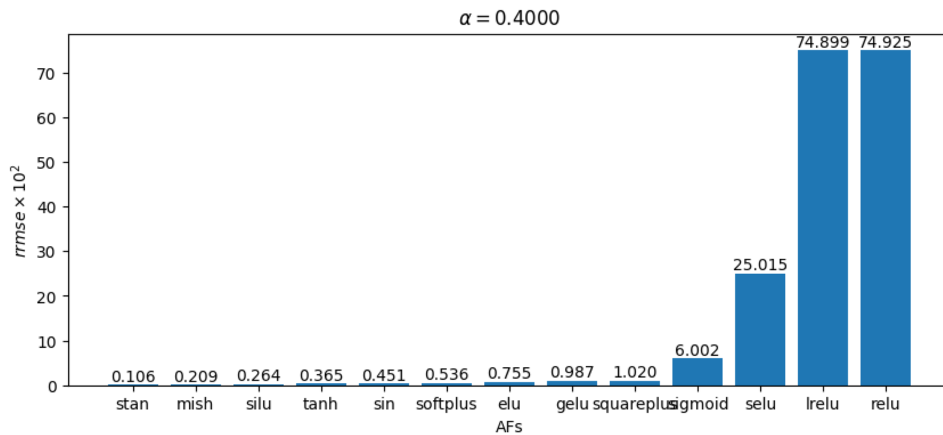
PI-DeepONet Analysis

For the PI-DeepONet, the worst performers are the relu, leaky relu, and selu AFs, while the best performers are mish, silu, and stan AFs. Some possibilities for their performance are listed below.

- The relu, leaky relu, and selu AFs are monotonic functions, i.e. these functions move in one direction (from left to right, the functions trend upwards as shown in figures A.1a, A.1b, A.1d). As a result, the derivatives are always positive and may limit the learning capacity of the neural network. However, mish, silu, and stan AFs are non-monotonic functions, i.e. these functions move in two direction (from left to right, the functions trend downwards then upwards as shown in figures A.2e, A.3b, A.3a). Thus we obtain positive derivatives for certain inputs and negative derivatives for others thereby improving the model's expressivity.



(a) DI-DeepONet at $\alpha = 0.4$.



(b) PI-DeepONet at $\alpha = 0.4$.

Figure 3.5 RRMSE Bar Graphs for the DI-DeepONet (top) and PI-DeepONet (bottom) with different AFs at $\alpha = 0.4$.

- All AFs are unbounded above and do not suffer from gradient saturation, i.e for large inputs, the outputs do not saturate to some max value (as opposed to sigmoid (figure A.2a) and tanh (figure A.2c) AFs), thus the gradients are never 0.

- Relu suffers from vanishing gradients for negative inputs and does not utilize these negative values producing outputs and gradients with a value of 0, thus resulting in no weight updates during back-propagation, negatively impacting the learning capacity of the model. All other AFs make use of a limited range of negative inputs before saturating to some constant value, except for leaky relu and stan which both utilize values up to $-\infty$ and produces a non-zero output. However, leaky relu produces large negative outputs while stan produces large positive outputs, as the input tends to $-\infty$. The use of a small amount of negative inputs improves the expressivity of the model and allows for inputs to flow through the network during forward propagation.

- All AFs, except stan and leaky relu are bounded below, thus as the input tends to $-\infty$, the output tends to some constant value and the gradients become 0. This makes these AFs noise-robust whereby large negative inputs saturate to some constant reducing the impact of noisy inputs. This introduces a strong regularization effect while training.

- Mish, silu, and stan AFs are smooth, continuously differentiable functions, i.e. they do not change direction suddenly like relu, leaky relu, and selu but bend smoothly near 0. Smoother transition results in a smoother loss function allowing the optimizer to go through fewer oscillations which helps in faster convergence, effective optimization and generalization.

- Selu and stan AFs are adaptive activation functions with one or more neuron-wise parameters that are optimized during training. This controls the slope of the AFs and their derivatives, which that can improve the flow of gradients through the model.

3.2.8 Summary of Findings

Important findings we observed for the DI-DeepONet and the PI-DeepONet is given below.

- DI-DeepONet contains 100% training data for the IBCs and within the domain.
- PI-DeepONet has only 75% training data for the IBCs.
- PI-DeepONet has a more complex learning task, as compared to the DI-DeepONet, as it has to satisfy the heat PDE constraint through physics, as opposed from data.
- Both DI-DeepONet and PI-DeepONet reach reasonable accuracy in their predictions, with the PI-DeepONet performing one order of magnitude better, even though it lacks training data.
- The gradient terms, u_t and u_{xx}, are more similar in shape for the PI-DeepONet, as opposed to the DI-DeepONet.
- The quality of the training data has an impact on the solution operator G learned by the DI-DeepONet.
- Different AFs have a greater impact on the accuracy and training efficiency for the PI-DeepONet, where as, a lesser impact is observed for the DI-DeepONet.

3.2.9 Conclusions

The results highlight the ability of the DI-DeepONet and the PI-DeepONet to learn the heat operator G, that maps α to the corresponding PDE solutions $u(x,t)$, for the heat equation (3.12). The PI-DeepONet performed better than the DI-DeepONet, even though it was trained with 25% less training data (for satisfying the heat PDE constraint). This constraint is enforced by defining a custom Pytorch module and adding an additional Node to the computational graph generated by Modulus. This ensures that the PI-DeepONet is more consistent with physics of the defined problem (3.2.1).

3.3 2D Steady-State Heat Equation

In this example, we perform a numerical study on the steady-state 2D heat transfer test case, without the radiation BC, provided by WolframAlpha [32], in section *HeatTransfer-FEM-Stationary-2D-Single-HeatTransfer-0002* as shown in figure 3.6. We attempt to learn the heat PDE operator, G, using a DI-DeepONet and two versions of a PI-DeepONet, where the input functions h alter the convective heat transfer coefficient for the Robin BC. The training and test data-sets are generated using the FEM solver, FEniCS [26, 25].

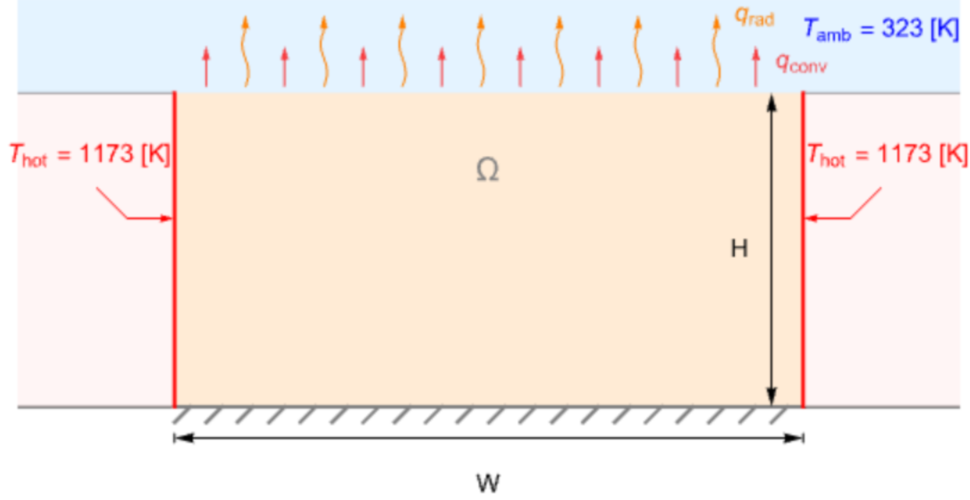


Figure 3.6 Illustration of the 2D Steady-State Heat Transfer Problem [32].

3.3.1 Problem Definition

Here we solve the heat conduction equation with an parametric convective heat transfer coefficient h $[W/m^2K]$ given by equation 3.24. The aim is to learn the heat operator G that maps h to the corresponding PDE solutions $u(x, y)$ using a DI-DeepONet and two versions of a PI-DeepONet, G_θ. The difference between the two versions is given below.

- The first version, the PI-DeepONet, is trained with its physics defined by the heat conduction equation (3.24), as well as, for the Neumann and Robin BC as given by equations 3.26 and 3.27 respectively.
- The second version, the HPI-DeepONet, is trained with all BCs supplied by data, and the physics defined only by the heat conduction equation (3.24).

The problem to be solved is steady-state and given by

$$-k\left(\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2}\right) = 0, x[m] \in [0, 0.02], y[m] \in [0, 0.01], \tag{3.24}$$

where k $[W/mK]$ is the thermal conductivity of the material domain.

The left and right boundaries are Dirichlet BCs,

$$u(0, y) = u(0.02, y) = 1173K, \forall y \in [0, 0.01], \tag{3.25}$$

the bottom boundary is Neumann BC,

$$\hat{n} \cdot k\left(\frac{du}{dx} + \frac{du}{dy}\right) = 0, \forall x \in [0, 0.02] \cup y = 0, \tag{3.26}$$

and the top boundary is Robin BC

$$\hat{n} \cdot k\left(\frac{du}{dx} + \frac{du}{dy}\right) = h(u_{amb} - u), \forall x \in [0, 0.02] \cup y = 0.01. \tag{3.27}$$

The table below provides the parameters used for defining the heat conduction PDE. h will represent a constant input function for the branch net of the DeepONet models, and will take a value within the range specified in table 3.6.

Table 3.6 PDE parameters for the heat conduction problem given by equation 3.24.

Parameter	Value
k $[W/mK]$	3
h $[W/m^2K]$	5 - 180
u_{amb} $[K]$	323

3.3.2 Data-set Generation

The training and test data-sets consists of input-output triplets $[h, (x, y), G(h)((x, y))]$, where h is a constant function, (x, y) is an evaluation location within the problem domain and $G(h)((x, y)) = u(x, y)$ is the temperature value to be predicted by the network for the given h at the (x, y) coordinate. The FEM solutions for u was generated using a FEniCS python solution file, provided to me by Hitachi. Additional steps, highlighted below, were then performed to format the generated results file for training the DeepONet models.

To generate the training and test data-sets, we first sample $2 \times n = 500$ different h values uniformly between the range $[5, 180]$. Using these values, we generate the FEM solutions for the temperature values u at 23000 different (x, y) coordinates. This was written to a file, where the first two columns provide the x and y coordinates, the third column provides the temperature values u and the fourth the specific h value used. We then reduce the number of evaluation locations (x, y) to be passed to the trunk network during training by randomly selecting $p = 100$ for the residual, and $p = 100$ for each BC, from each of the $n = 500$ files.

In summary, for the DI-DeepONet, for each $h^{(i)}$,

- $\{(x_{r,j}^{(i)}, y_{r,j}^{(i)})\}_{j=1}^{p=100}$ are uniformly sampled within the domain $[0, 0.02] \times [0, 0.01]$ (residual),
- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)})\}_{j=1}^{p=100}$ are uniformly sampled from the left and right boundaries, each, (Dirichlet),
- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)})\}_{j=1}^{p=100}$ are uniformly sampled from the bottom and top boundaries, each, (Neumann and Robin respectively),

for computing the data loss, $L_{data}(\theta)$ (3.28). Thus the training data-set contains $[h, (x, y), G(h)(x, y)]$ of size $(500 \times 500, 1)$, $(500 \times 500, 2)$ and $(500 \times 500, 1)$ respectively.

For generating the training data-set for the PI-DeepONet,

- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)})\}_{j=1}^{p=100}$ are uniformly sampled from the left and right boundaries, each, (Dirichlet),

In order to satisfy the data loss, $L_{data}(\theta)$ (3.40). The training data-set contains $[h, (x, y), G(h)(x, y)]$ of size $(500 \times 200, 1)$, $(500 \times 200, 2)$ and $(500 \times 200, 1)$, for the Dirichlet BCs (where $x = 0$ and $x = 0.02$),

and is only required for computing $L_{data}(\theta)$.

We then randomly select $q = 100$ different (x, y) coordinates from the n different fine grid locations to satisfy the Neumann and Robin BCs, as well as, the PDE residual for $L_{physics}(\theta)$ (3.36). Thus

- $\{(x_{r,j}^{(i)}, y_{r,j}^{(i)})\}_{j=1}^{q=100}$ are uniformly sampled within the domain $[0, 0.02] \times [0, 0.01]$ (residual),
- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)})\}_{j=1}^{q=100}$ are uniformly sampled from the bottom and top boundaries, each, (Neumann and Robin respectively).

For generating the training data-set for the HPI-DeepONet,

- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)})\}_{j=1}^{p=100}$ are uniformly sampled from the left and right boundaries, each, (Dirichlet),
- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)})\}_{j=1}^{p=100}$ are uniformly sampled from the bottom and top boundaries, each, (Neumann and Robin respectively),

in order to satisfy the data loss, $L_{data}(\theta)$ (3.42). The training data-set contains $[h, (x, y), G(h)(x, y)]$ of size $(500 \times 400, 1)$, $(500 \times 400, 2)$ and $(500 \times 400, 1)$ for the BCs, and is only required for computing $L_{data}(\theta)$.

We then randomly select $q = 100$ different (x, y) coordinates from the n different fine grid locations to satisfy the PDE residual for $L_{physics}(\theta)$ (3.41). Thus,

- $\{(x_{r,j}^{(i)}, y_{r,j}^{(i)})\}_{j=1}^{q=100}$ are collocation points sampled within the domain $[0, 0.02] \times [0, 0.01]$ for computing the residual loss, to satisfy the heat PDE constraint.

For testing, we use the entire data-set generated on a fine-grid with 23000 (x, y) coordinates. The testing data-set contains $[h, (x, y), G(h)(x, y)]$ of size $(500 \times 23000, 1)$, $(500 \times 23000, 2)$ and $(500 \times 23000, 1)$ respectively.

The parameters used for generating the data-sets are shown in the table 3.7.

Table 3.7 Parameters used for generating the training and test data-sets.

Parameter	DI-DeepONet	PI-DeepONet	HPI-DeepONet
n	500	500	500
p	500	200	400
q	-	300	100

We also augment the data-set by applying scaling factors to the evaluation points (x, y) and the temperature u values, which are of order of magnitude 10^{-2} and 10^3 respectively, to bring them closer to unity. In addition, we also center the (x, y) points around 0. For example,

- To unity: $x \in [0, 0.02] \times 10^{-2} = [0, 2.0]$.
- Centering: $x \in [0, 2.0] - \frac{(2.0-0)}{2} = [-1.0, 1.0]$.

The h and k values are then adjusted as these values are derived from the spatial and temperature values. This augmentation was performed to make training the DeepONet models easier.

Note:

- DI-DeepONet: Trained on paired input-output measurements for the domain, Dirichlet, Neumann and Robin BCs.

- PI-DeepONet: Trained without any paired training data (except for the Dirichlet BCs).
- HPI-DeepONet: Trained without any paired training data (except for the Dirichlet, Neumann and Robin BCs).

As a result, the PI-DeepONet contains 60% less training data (for predicting $G(h)(x,y)$ within the domain, as well as, the top and bottom boundaries), while the HPI-DeepONet contains 20% less training data (for predicting $G(h)(x,y)$ within the domain), when compared to the DI-DeepONet. Additionally, as the Dirichlet BCs are known a priori, the training data for the PI-DeepONet can be assembled without first solving time-consuming FEA models or sensor measurements. However, this is not possible with the DI-DeepONet as the temperatures, u, within the domain, and along the top and bottom boundaries at randomly generated collocation points, (x,y), are needed. The HPI-DeepONet also requires the use of these time-consuming methods as u is required for (x,y) along the top and bottom boundaries. Thus, $G(h)(x,y)$ in the training data for the Dirichlet BCs are not obtained from the numerical solution, as its value is already known.

3.3.3 Loss Function

Recall that G_θ represents the approximation of the heat operator by each DeepONet.

DI-DeepONet Loss Function

The loss function for the DI-DeepONet is given by

$$L(\theta) = L_{data}(\theta) = L_{r,data}(\theta) + L_{dc,data}(\theta) + L_{n,data}(\theta) + L_{c,data}(\theta), \tag{3.28}$$

to satisfy the constraints for the residual r, Dirichlet dc, Neumann n, and Robin c boundaries respectively. Each term is defined as

$$L_{r,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(h^{(i)})(x_{r,j}^{(i)}, y_{r,j}^{(i)}) - G(h^{(i)})(x_{r,j}^{(i)}, y_{r,j}^{(i)})\right|^2, \tag{3.29}$$

$$L_{dc,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(h^{(i)})(x_{dc,j}^{(i)}, y_{dc,j}^{(i)}) - G(h^{(i)})(x_{dc,j}^{(i)}, y_{dc,j}^{(i)})\right|^2, \tag{3.30}$$

$$L_{n,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(h^{(i)})(x_{n,j}^{(i)}, y_{n,j}^{(i)}) - G(h^{(i)})(x_{n,j}^{(i)}, y_{n,j}^{(i)})\right|^2, \tag{3.31}$$

$$L_{c,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(h^{(i)})(x_{c,j}^{(i)}, y_{c,j}^{(i)}) - G(h^{(i)})(x_{c,j}^{(i)}, y_{c,j}^{(i)})\right|^2. \tag{3.32}$$

PI-DeepONet Loss Function

Let R_θ be the heat PDE residual which is given by

$$R_\theta^{(i)}(x,y) = -k\left(\frac{d^2G_\theta(h^{(i)})(x,y)}{dx^2} + \frac{d^2G_\theta(h^{(i)})(x,y)}{dy^2}\right), \tag{3.33}$$

let N_θ be the residual for the Neumann BC given by

$$N_\theta^{(i)}(x,y) = \hat{n}\cdot k\left(\frac{du^{(i)}(x,y)}{dx} + \frac{du^{(i)}(x,y)}{dy}\right), \tag{3.34}$$

and let C_θ be the residual for the Robin BC given by

$$C_\theta^{(i)}(x,y) = \hat{n} \cdot k\Big(\frac{du^{(i)}(x,y)}{dx} + \frac{du^{(i)}(x,y)}{dy}\Big) - h^{(i)}(u_{amb} - u^{(i)}(x,y)). \tag{3.35}$$

The loss function for the PI-DeepONet can then be defined as

$$L(\theta) = L_{data}(\theta) + L_{physics}(\theta),$$

where $L_{physics}(\theta)$ is given by

$$L_{physics}(\theta) = L_{r,physics}(\theta) + L_{n,physics}(\theta) + L_{c,physics}(\theta), \tag{3.36}$$

with each term given as

$$L_{r,physics}(\theta) = \frac{1}{nq}\sum_{i=1}^{n}\sum_{j=1}^{q}\Big|R_\theta^{(i)}(x_{r,j}^{(i)}, y_{r,j}^{(i)})\Big|^2, \tag{3.37}$$

$$L_{n,physics}(\theta) = \frac{1}{nq}\sum_{i=1}^{n}\sum_{j=1}^{q}\Big|N_\theta^{(i)}(x_{n,j}^{(i)}, y_{n,j}^{(i)})\Big|^2, \tag{3.38}$$

$$L_{c,physics}(\theta) = \frac{1}{nq}\sum_{i=1}^{n}\sum_{j=1}^{q}\Big|C_\theta^{(i)}(x_{c,j}^{(i)}, y_{c,j}^{(i)})\Big|^2. \tag{3.39}$$

$L_{data}(\theta)$ is composed of only the collocation points of the Dirichlet boundaries, i.e. $(x_{dc,j}^{(i)}, y_{dc,j}^{(i)})$, and is given by

$$L_{data}(\theta) = L_{dc,data}(\theta). \tag{3.40}$$

HPI-DeepONet Loss Function

The loss function for the HPI-DeepONet is defined as

$$L(\theta) = L_{data}(\theta) + L_{physics}(\theta),$$

where $L_{physics}(\theta)$ is defined only for the heat PDE residual, i.e.

$$L_{physics}(\theta) = L_{r,physics}(\theta). \tag{3.41}$$

$L_{data}(\theta)$ is composed of the collocation points of the boundaries of the domain for the Dirichlet, Neumann and Robin BCs, and is given by

$$L_{data}(\theta) = L_{dc,data}(\theta) + L_{n,data}(\theta) + L_{c,data}(\theta). \tag{3.42}$$

Defining the heat PDE Loss, $L_{r,physics}(\theta)$, in Modulus

In listing 5, we define the PDE loss $L_{r,physics}(\theta)$ in Modulus using a torch.nn.Module, and a class called HeatPDE, which uses the input and output tensors of our DeepONet model to compute the residual R_θ within its forward method. This module is then incorporated into the computational graph created by Modulus using a Node as shown in listing 6. The PDE loss $L_{r,physics}(\theta)$ is then added as an additional constraint using Modulus's DeepONetConstraint as shown in listing 7.

```python
class HeatPDE(torch.nn.Module):
    "Custom 2-dimensional Heat PDE definition for PI-DeepONet"

    def __init__(self, K):
        super().__init__()
        self.K = K

    def forward(self, input_var: Dict[str, torch.Tensor])
        -> Dict[str, torch.Tensor]:
        # Get inputs
        u = input_var["u"]
        h = input_var["h"]
        # Compute gradients
        dduddx = input_var["u__x__x"]
        dduddy = input_var["u__y__y"]

        # Compute Heat equation
        heat_pde = (-self.K*(dduddy + dduddx))

        # Return heat residual
        output_var = {
            "heat": heat_pde,
        }
        return output_var
```

Listing 5 Defining the Heat PDE in Modulus.

```python
# Incorporate Heat module into Modulus using a Node
heat_node = Node(
    inputs=["u", "h", "u__x__x", "u__y__y"],
    outputs=["heat"],
    evaluate=HeatPDE(K),
    name="Heat Node",
)
```

Listing 6 Adding the Heat Node into the computational graph.

```python
# Residual
residual = DeepONetConstraint.from_numpy(
    nodes=nodes,
    invar={
        "h": h_r_train,
        "x": xx_r_train,
        "y": yy_r_train,
    },
    outvar={
        "heat": np.zeros_like(u_r_train),
    },
    batch_size=cfg.batch_size.interior,
    lambda_weighting={
        "heat": 1.0*(np.ones_like(u_train)),
    },
)
domain.add_constraint(residual, "residual")
```

Listing 7 Adding the Heat PDE loss $L_{r,physics}(\theta)$ as an additional constraint.

Defining the Neumann and Robin BCs for the PI-DeepONet in Modulus

In listings 8 and 11, we define the Neumann and Robin BCs in Modulus using a torch.nn.Module and two classes called NeumannBC and RobinBC respectively, which use the input and output tensors of our DeepONet model to compute N_θ and C_θ respectively, within their forward methods. In order to obtain the unit normal vectors, $\hat{n}$, for the derivatives of u on the bottom and top boundaries, $\frac{du}{dx}$ and $\frac{du}{dy}$, we use the 'sample boundary' method from the 'rectangular geometry' instance representing our domain, created using Modulus' geometry module. These module are then incorporated into the computational graph created by Modulus using a Node for each BC as shown in listings 9 and 12 respectively. NeumannBC and RobinBC are then added as additional constraints using Modulus's DeepONetConstraint for computing the losses, $L_{n,physics}(\theta)$ and $L_{c,physics}(\theta)$, respectively, as shown in listings 10 and 13 respectively.

```
class NeumannBC(torch.nn.Module):
    "Custom Neumann BC for 2D Heat Equation"

    def __init__(self, K, geo):
        super().__init__()
        self.K = K
        self.geo = geo

    def forward(self, input_var: Dict[str, torch.Tensor])
        -> Dict[str, torch.Tensor]:
        # Get inputs
        u = input_var["u"]
        h = input_var["h"]
        # Compute gradients
        dudx = input_var["u__x"]
        dudy = input_var["u__y"]
        # Get normals
        y = Symbol("y")
        samples = self.geo.sample_boundary(10, Eq(y, 0))
        normal_x = samples["normal_x"][0, 0]
        normal_y = samples["normal_y"][0, 0]

        # Compute Neumann BC
        normal_gradient_u = (self.K)*(normal_x*dudx + normal_y*dudy)

        # Return Neumann BC
        output_var = {
            "normal_gradient_u": normal_gradient_u,
        }
        return output_var
```

Listing 8 Defining the Neumann BC in Modulus.

```
# Neumann BC
NBC_node = Node(
    inputs=["u", "h", "u__x", "u__y"],
    outputs=["normal_gradient_u"],
    evaluate=NeumannBC(K, geo),
    name="Neumann Node",
)
```

Listing 9 Adding the Neumann Node into the computational graph.

```python
# Neumann boundary condition 1, y=0, bottom wall
NBC_1 = DeepONetConstraint.from_numpy(
    nodes=nodes,
    invar={
        "h": h_train,
        "x": xx_train,
        "y": y_range[0]*np.ones_like(yy_train),
    },
    outvar={"normal_gradient_u": np.zeros_like(u_train)},
    batch_size=cfg.batch_size.NBC,
    lambda_weighting={"normal_gradient_u": 1.0*(np.ones_like(u_train))},
)
domain.add_constraint(NBC_1, "NBC_1")
```

Listing 10 Adding the Neumann loss $L_{n,physics}(\theta)$ as an additional constraint.

```python
class RobinBC(torch.nn.Module):
    "Custom Robin BC for 2D Heat Equation"

    def __init__(self, T_amb, K, geo):
        super().__init__()
        self.T_amb = T_amb
        self.K = K
        self.geo = geo

    def forward(self, input_var: Dict[str, torch.Tensor])
        -> Dict[str, torch.Tensor]:
        # Get inputs
        u = input_var["u"]
        h = input_var["h"]
        # Compute gradients
        dudx = input_var["u__x"]
        dudy = input_var["u__y"]
        # Get normals
        y = Symbol("y")
        samples = self.geo.sample_boundary(10, Eq(y, 1))
        normal_x = samples["normal_x"][0, 0]
        normal_y = samples["normal_y"][0, 0]

        # Compute Robin BC
        ## [normal_x * u__x + normal_y * u__y] - [(h/k)*(T_amb - T)]
        normal_gradient_rbc = (
            (self.K)*(normal_x*dudx + normal_y*dudy) - (h)*(self.T_amb - u))

        # Return Neumann BC
        output_var = {
            "normal_gradient_rbc": normal_gradient_rbc,
        }
        return output_var
```

Listing 11 Defining the Robin BC in Modulus.

```python
# Robin BC
RBC_node = Node(
    inputs=["u", "h", "u__x", "u__y"],
    outputs=["normal_gradient_rbc"],
    evaluate=RobinBC(T_amb, K, geo),
    name="Robin Node",
)
```

Listing 12 Adding the Robin Node into the computational graph.

```python
# Robin boundary condition 1, y=0.01, top wall
RBC_1 = DeepONetConstraint.from_numpy(
    nodes=nodes,
    invar={
        "h": h_train,
        "x": xx_train,
        "y": y_range[1]*np.ones_like(yy_train),
    },
    outvar={"normal_gradient_rbc": np.zeros_like(u_train)},
    batch_size=cfg.batch_size.RBC,
    lambda_weighting={
        "normal_gradient_rbc": 1.0 * (np.ones_like(u_train))},
)
domain.add_constraint(RBC_1, "RBC_1")
```

Listing 13 Adding the Robin loss $L_{c,physics}(\theta)$ as an additional constraint.

Defining the Dirichlet BCs in Modulus

The Dirichlet BCs for the left and right boundaries are easily added as constraints using Modulus's DeepONetConstraint for each DeepONet. Note that the Neumann and Robin BCs are similarly enforced for the DI-DeepONet and HPI-DeepONet, as well as, the heat PDE residual for the DI-DeepONet. An example is shown in listing 14 for the left boundary.

```python
# Dirichlet boundary condition 1, x=0, left wall
DBC_1 = DeepONetConstraint.from_numpy(
    nodes=nodes,
    invar={
        "h": h_train,
        "x": x_range[0]*np.ones_like(xx_train),
        "y": yy_train,
    },
    outvar={"u": T_hot*np.ones_like(u_train)},
    batch_size=cfg.batch_size.DBC,
    lambda_weighting={
        "u": 1.0*(np.ones_like(u_train)),
    },
)
domain.add_constraint(DBC_1, "DBC_1")
```

Listing 14 Adding the Dirichlet BC constraint $L_{dc,data}(\theta)$ in Modulus.

3.3.4 Training

Recall (refer to section 2.2.1):

- The branch network takes h as input and returns a features embedding vector br as output.
- The trunk network takes (x, y) as input and returns a features embedding vector tr as output.
- These outputs are then merged together by a dot product to produce the final prediction of the heat operator by each DeepONet, i.e. $G_\theta = br \cdot tr$.

The network architecture and training parameters used are summarized in table 3.8.

Table 3.8 Network architecture and training parameters used by each DeepONet.

Name	**Value**
Architecture	Fully-connected Neural Network
Depth	4
Width	128
Optimiser	Adam
Learning Rate	1.0×10^{-3} with decay
Training Steps	100000
Activation Function	MISH (DI) / SILU (PI, HPI)

PI-DeepONet Training

The PI-DeepONet loss function, L_θ, is computed as a sum of point-wise differences (equation 2.2) for:

- Predicting the temperature u for the Dirichlet BCs, as given by the data loss L_{data},
- Predicting the PDE residual for the heat equation, R_θ, the Neumann residual for the Neumann BC, N_θ, and Robin residual for the Robin BC, C_θ, as given by the physics loss $L_{physics}$ (3.36).

As a result, we apply lambda weightings to the loss function for each objective, i.e.

$$L(\theta) = \lambda_{dc,data} L_{dc,data}(\theta) + \lambda_{n,physics} L_{n,physics}(\theta) + \lambda_{c,physics} L_{c,physics}(\theta) + \lambda_{r,physics} L_{r,physics}(\theta), \quad (3.43)$$

where $\lambda_{dc,data}$, $\lambda_{n,physics}$, $\lambda_{c,physics}$, and $\lambda_{r,physics}$ are given in table 3.9.

Table 3.9 Lambda weightings used by the PI-DeepONet.

Name	**Value**
$\lambda_{dc,data}$	$70.0 + 100.0 * y$, $(y \in [-0.5, 0.5])$
$\lambda_{n,physics}$	1.0
$\lambda_{c,physics}$	1.0
$\lambda_{r,physics}$	1.0

These lambda values were selected by training the network with uniform lambda weightings and observing the loss obtained from each loss term on the validation data-set. The Dirichlet BCs proved most difficult for the PI-DeepONet to approximate to reasonable accuracy, especially around the upper portion of the domain. The function, $\lambda_{dc,data}$, in the table 3.9 was used to ensure that the parameter updates, θ, during training weighed $L_{dc,data}$ more as compared to the other loss terms, and more around the upper portion of the domain. This function was obtained by trial and error.

HPI-DeepONet Training

The HPI-DeepONet loss function is computed similar to the PI-DeepONet, however, the point-wise differences is computed for:

- Predicting the temperature u for the Dirichlet, Neumann and Robin BCs, as given by the data loss L_{data} (3.42),
- Predicting the PDE residual for the heat equation, R_θ, as given by the physics loss $L_{physics}$.

We apply lambda weightings to the loss function for each objective as

$$L(\theta) = \lambda_{dc,data} L_{dc,data}(\theta) + \lambda_{n,data} L_{n,data}(\theta) + \lambda_{c,data} L_{c,data}(\theta) + \lambda_{r,physics} L_{r,physics}(\theta), \tag{3.44}$$

where $\lambda_{dc,data}$, $\lambda_{n,data}$, $\lambda_{c,data}$, and $\lambda_{r,physics}$ are given in table 3.10.

Table 3.10 Lambda weightings used by the HPI-DeepONet.

Name	Value
$\lambda_{dc,data}$	$2.0 + 100.0 * \|y\|, (y \in [-0.5, 0.5])$
$\lambda_{n,data}$	5.0
$\lambda_{c,data}$	$30.0 + 20.0 * \|x\|, (x \in [-1.0, 1.0])$
$\lambda_{r,physics}$	1.0

These lambda values were selected by training the network with uniform lambda weightings and observing the loss obtained from each loss term on the validation data-set. The Dirichlet, Neumann and Robin BCs proved difficult for the HPI-DeepONet to approximate to reasonable accuracy, with the Robin BC proving most difficult followed by the Dirichlet BC. We observed that both the Robin and Dirichlet BCs had a higher loss around the corners of the domain. The functions, $\lambda_{dc,data}$ and $\lambda_{c,data}$, in the table 3.10 was used to ensure that the parameter updates θ during training weighed $L_{dc,data}$ and $L_{c,data}$ more as compared to the other loss terms, and more around the corners. These weightings were obtained by trial and error.

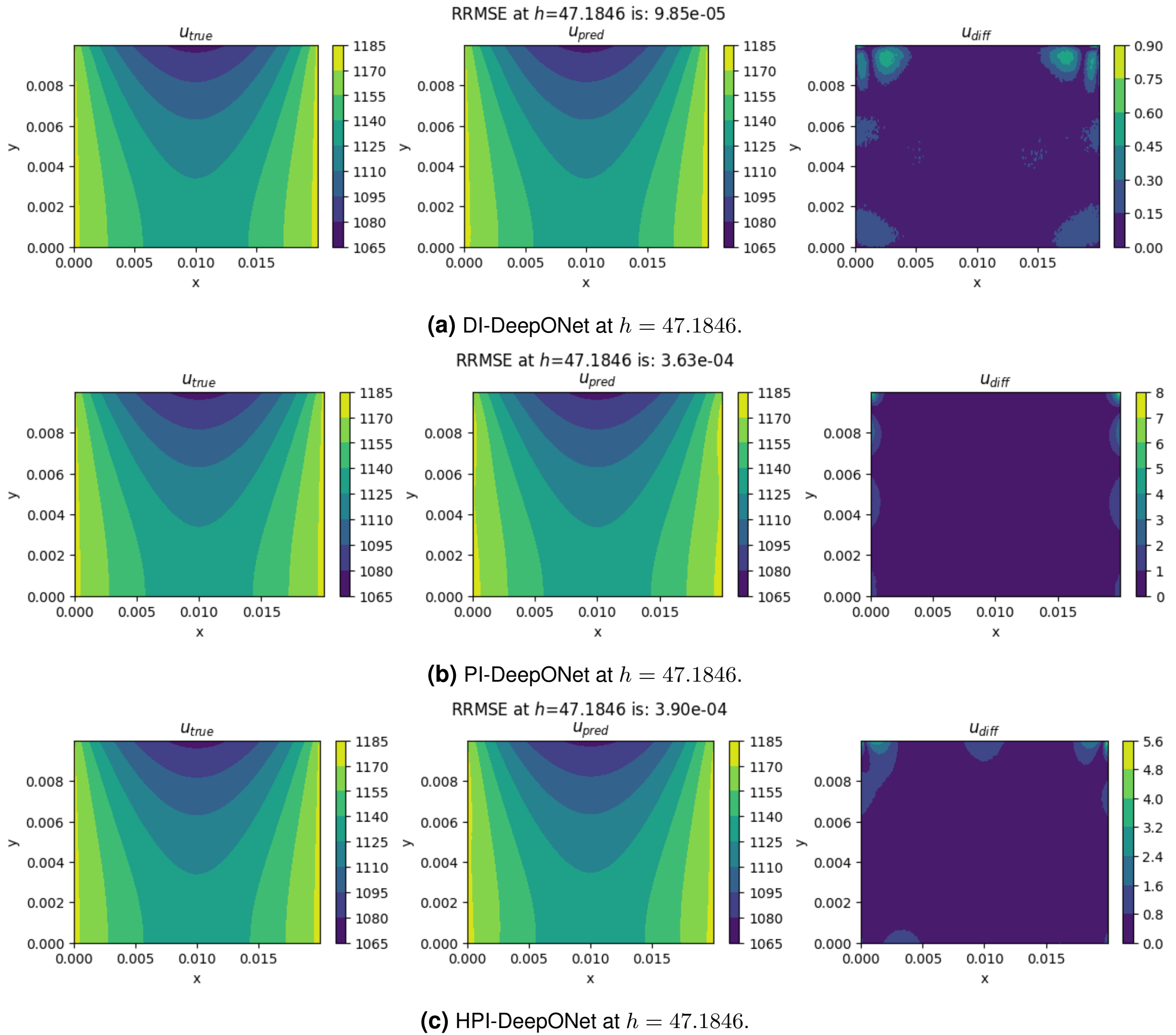
DI-DeepONet Training

The DI-DeepONet loss function L_θ is computed as a sum of point-wise differences for predicting the temperature u for the BCs, as well as, within the domain, as given by the data loss L_{data} (3.28). However, we apply uniform lambda weightings to the loss function for each objective.

3.3.5 Results

The temperature field $\hat{u}$ predicted by the DI-DeepONet, PI-DeepONet, and HPI-DeepONet are compared to the true u, obtained from the FEM solution (see section 3.3.2), using Modulus' Validators. These results are shown for a convective heat transfer coefficient $h = 47.1846$ for the DI-DeepONet, PI-DeepONet and HPI-DeepONet, as shown in figure 3.7. The RRMSE for the validation data-set at 6 different h coefficients are also provided in table 3.11. We observe:

- DI-DeepONet (3.7a) achieved reasonable accuracy throughout the domain, except at the upper corners of the domain. The accuracy also degrades as h increases.
- PI-DeepONet (3.7b) achieved reasonable accuracy throughout the domain, except at the left and right boundaries of the domain, which degrades as we approach the upper corners of the domain. The accuracy also degrades as h increases.



(a) DI-DeepONet at $h = 47.1846$.

(b) PI-DeepONet at $h = 47.1846$.

(c) HPI-DeepONet at $h = 47.1846$.

Figure 3.7 Temperature Field Predicted by DI-DeepONet (top), PI-DeepONet (middle) and HPI-DeepONet (bottom), with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).

- HPI-DeepONet (3.7c) achieved reasonable accuracy throughout the domain, except at the top, left and right boundaries of the domain, which degrades as we approach the corners of the domain. Similar to the PI-DeepONet, the temperature field u in the upper portion of the domain proves to be more difficult for the networks to predict. The accuracy also degrades as h increases.

Table 3.11 RRMSE for DI-DeepONet, PI-DeepONet and HPI-DeepONet at 6 different h.

h	**DI-DeepONet**	**PI-DeepONet**	**HPI-DeepONet**
16.2291	0.656×10^{-4}	1.341×10^{-4}	1.825×10^{-4}
47.1846	0.985×10^{-4}	3.629×10^{-4}	3.903×10^{-4}
69.0142	1.199×10^{-4}	5.329×10^{-4}	5.241×10^{-4}
87.1556	1.583×10^{-4}	6.787×10^{-4}	6.607×10^{-4}
141.3821	2.594×10^{-4}	11.44×10^{-4}	10.62×10^{-4}
156.2909	2.879×10^{-4}	13.10×10^{-4}	11.81×10^{-4}

Each DeepONet achieved reasonable accuracy on the validation data-set, however, the DI-DeepONet performed best which is likely due to the simplicity of its loss function L (section 3.3.3) comprised of only training data for the entire domain and the BCs. The HPI-DeepONet was second, with a loss function L (section 3.3.3) comprised of mostly training data for all the BCs and a smaller portion for the physics for the heat PDE. Lastly, the PI-DeepONet has the most complex loss function L (section 3.3.3 where a small portion is obtained from training data for the Dirichlet BCs, and a larger portion obtained from physics for the Robin and Neumann BCs, as well as, the heat PDE.

The accuracy of all DeepONets degrade as h increases, particularly towards the upper boundary of the domain. The HPI-DeepONet is greater affected by this increase compared to the other variants, as shown in the figure 3.7c, with both the Robin and Dirichlet BCs showing larger differences between the true and predicted temperatures u_{diff}. u_{diff} for the PI-DeepONet along the upper boundary (figure 3.7b) is not as prominent, and indicates that learning the heat operator G, for this boundary, from data is more challenging for the DeepONets. This conclusion is further extended to all boundaries as we had to apply stronger lambda weightings to the HPI-DeepONet loss function for the Neumann and Robin BCs to be predicted to reasonable accuracy, when compared to the similar terms in the PI-DeepONet loss function.

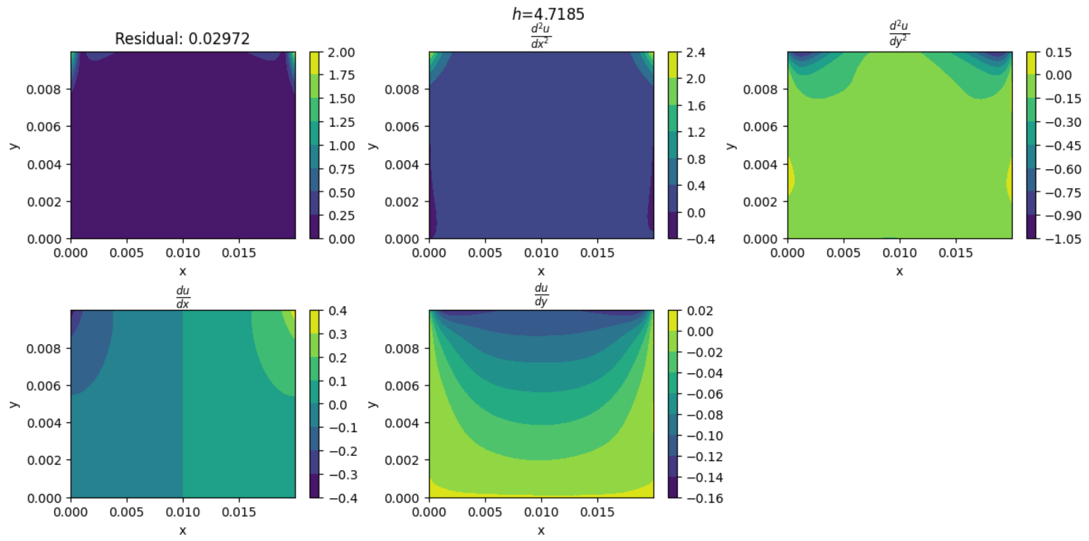
Additionally, as h increases, the effect of convective heat transfer along the upper boundary increases which results in larger temperature gradients in the y-direction, u_{yy}, within the domain towards the upper portion. The left and right sides of the domain are also at higher temperatures u, due to the Dirichlet BCs, which also results in higher u_{yy}. As a result, u_{yy} increases to a maximum as we approach the upper left and upper right corners. These large gradients may pose a challenge for the DeepONet models to learn when trying to satisfy the heat PDE and Robin BC. It may also explain the higher loss observed at the upper corners of the domain for the PI-DeepONet when compared to the other variants, as it is attempting to satisfy u_{yy} from the heat PDE and the Robin BC.

We then compared how well the PDE residual, R_θ (3.33), is satisfied by each DeepONet. As described in section 3.3.3, the PI-DeepONet and the HPI-DeepONet is trained with R_θ as an additional objective, which acts as a regularization mechanism that biases the network's output to satisfy the heat PDE constraint, while the DI-DeepONet is trained only with data. Also note that, N_θ and C_θ are also additional regularization mechanisms added to the PI-DeepONet, as opposed to the HPI-DeepONet, to satisfy the Neumann and Robin constraints respectively.

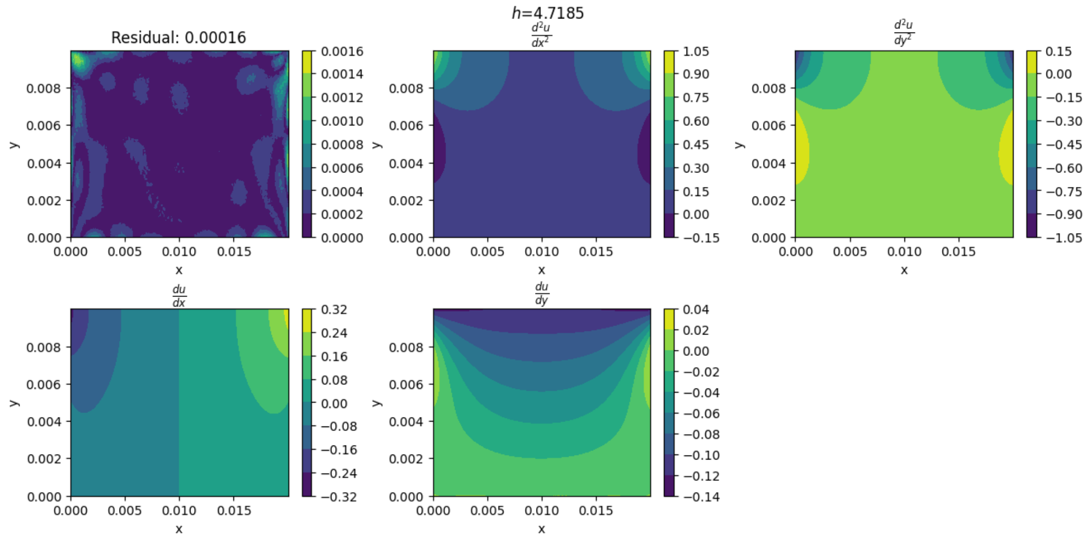
The PDE residual is satisfied more accurately by the PI-DeepONet and HPI-DeepONet by 4 orders of magnitude better than the DI-DeepONet as shown in figure 3.8. Both the PI-DeepONet and HPI-DeepONet perform similarly in satisfying this constraint. An Inferencer was used to track the PDE partial derivatives, u_x and u_y, u_{xx} and u_{yy} within the domain during training, in order to produce these plots.

u_{xx} and u_{yy} within the domain are also provided for each DeepONet in figure 3.8. We observe that u_{xx} and u_{yy} have a more similar profiles for the PI-DeepONet and HPI-DeepONet, as compared to the DI-DeepONet. By comparison, the DI-DeepONet is unable to adequately predict u_{xx} and u_{yy}, particularly at the upper corners of the domain, where higher temperature gradients are observed. Additionally, the areas with larger u_{diff}, as shown in figure 3.7, match closely with the areas where u_{xx} and u_{yy} differ (3.8a) for the DI-DeepONet, or have high gradients (3.8b and 3.8c) for the PI-DeepONet and HPI-DeepONet respectively.

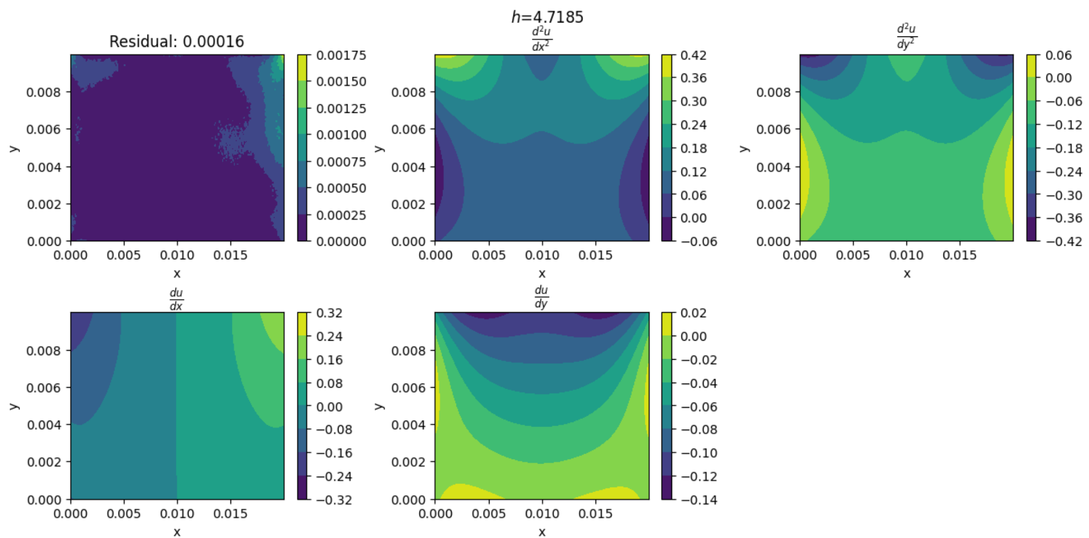
u_y along the Neumann (bottom) boundary is also satisfied more closely for the PI-DeepONet with a value more consistently near 0 as opposed to DI-DeepONet and HPI-DeepONet. u_y along the Robin (top) boundary is also satisfied more closely for the PI-DeepONet with an isosceles triangle profile that peaks at the center of the domain that steady declines towards the corners, in order to negate the term $h(u_{amb} - u)$. This is opposed to DI-DeepONet and HPI-DeepONet, which appear to have a saw-tooth



(a) DI-DeepONet at $h = 47.1846$.



(b) PI-DeepONet at $h = 47.1846$.



(c) HPI-DeepONet at $h = 47.1846$.

Figure 3.8 PDE Residual (left), u_{xx} (middle), and u_{yy} (right) predicted by the DI-DeepONet (top), PI-DeepONet(middle), and HPI-DeepONet (bottom).

profile as we move towards the corners of the domain.

The temperature field $\tilde{u}$ predicted for 2 convective heat transfer coefficients h at the extremes of the range given in table 3.6 for the DI-DeepONet, PI-DeepONet and HPI-DeepONet respectively, are provided in Appendix A.2.

3.3.6 Summary of Findings

Important findings we observed for the DI-DeepONet, PI-DeepONet and HPI-DeepONet is given below.

- DI-DeepONet contains 100% training data for the BCs and within the domain.
- PI-DeepONet has 60% less training data, only 40% for the Dirichlet BCs, with the remainder of the constraints obtained by physics for the heat PDE, Neumann and Robin BCs.
- HPI-DeepONet is trained from 80% training data for the Dirichlet, Neumann and Robin BCs, with the remainder obtained by physics for the heat PDE.
- PI-DeepONet has a more complex learning task, when compared to the DI-DeepONet and HPI-DeepONet, as it has to satisfy the heat equation, as well as, the Neumann and Robin BC constraints without any data.
- PI-DeepONet is more consistent with the underlying physics as R_θ, N_θ and C_θ, are added as additional objectives, which act as a regularization mechanism biasing the network's output to satisfy the heat PDE, Neumann and Robin BCs constraints.
- HPI-DeepONet is only biased to satisfy R_θ for the heat PDE constraint.
- The upper corners of the domain are challenging for each DeepONet to learn to reasonable accuracy due to high thermal gradients, u_{xx} and u_{yy}, due to convective heat transfer and the Dirichlet BCs.
- Each variant reached reasonable accuracy in their predictions, with DI-DeepONet performing the best, followed by the HPI-DeepONet.

3.3.7 Conclusions

The DI-DeepONet, PI-DeepONet and HPI-DeepONet are able to learn the solution operator that maps h to the corresponding PDE solutions $u(x)$ for the heat equation (3.24). Additionally, although not as accurate as the DI-DeepONet, the PI-DeepONet achieved comparable accuracy with only 40% training data for the Dirichlet BCs, and the remaining constraints satisfied through physics for the heat PDE, Neumann and Robin BCs. These constraints are defined using custom Pytorch modules and enforced by adding additional Nodes for each to the computational graph generated by Modulus. This ensures that the PI-DeepONet is more consistent with physics of the defined problem. The HPI-DeepONet, with 80% training data for the BCs and physics defined for the heat PDE, also achieves comparable accuracy. This variant is also more consistent with the defined problem (3.3.1).

3.4 2D Axisymmetric Transient Heat Equation

In this example, we perform a numerical study on a modified version of the 2D axisymmetric transient heat conduction test case provided by WolframAlpha [32], in section *HeatTransfer-FEM-Transient-2DAxisym-Single-HeatTransfer-0001*, with a source term $\dot{q}$. We attempt to learn the heat PDE operator, G, using a DI-DeepONet, a PI-DeepONet and a HPI-DeepONet, where the input functions $\dot{q}$ alter the source term. The training and test data-sets are supplied by Hitachi.

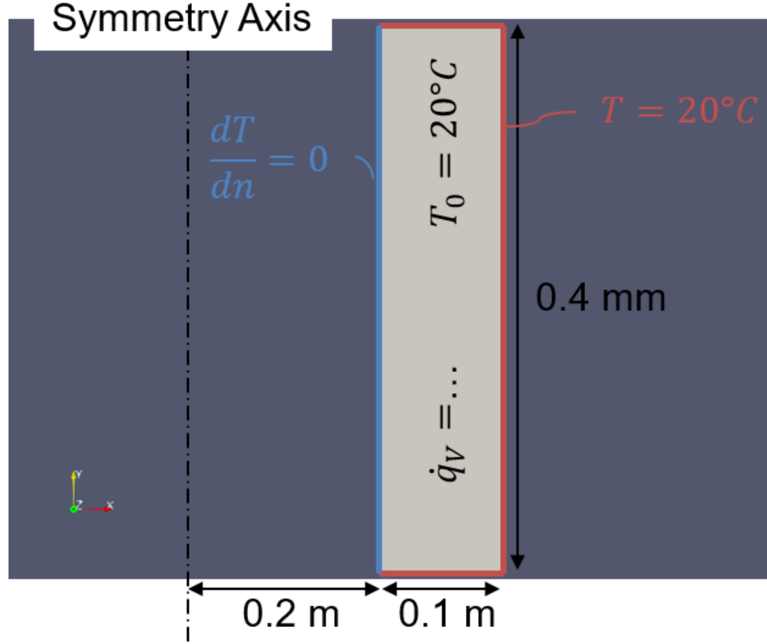


Figure 3.9 Illustration of the 2D Axisymmetric Transient Heat Transfer Problem.

3.4.1 Problem Definition

Here we solve the axisymmetric transient heat conduction equation with an parametric source $\dot{q}$ $[W/m^3]$. The problem domain is shown in figure 3.9. The aim is to learn the heat operator G that maps $\dot{q}$ to the corresponding PDE solutions $u(x, y, t)$ using a DI-DeepONet, a PI-DeepONet and a HPI-DeepONet, G_θ. The main differences between the PI-DeepONet and HPI-DeepONet is given below.

- PI-DeepONet is trained with its physics defined by the heat conduction equation (3.45), and by the Neumann BC as given by equation 3.48.
- HPI-DeepONet is trained with all BCs supplied by data, and the physics defined only by the heat conduction equation (3.45).

The problem to be solved is transient and given by

$$\begin{gathered}\rho C_p \frac{du}{dt} + \frac{1}{x}\frac{d}{dx}\Big(-k \cdot x\frac{du}{dx}\Big) + \frac{d}{dy}\Big(-k\frac{du}{dy}\Big) = \dot{q}, \\ x[m] \in [0.2, 0.3], y[m] \in [0, 0.4], t[s] \in [0, 200],\end{gathered} \tag{3.45}$$

where ρ is the material density $[kg/m^3]$, C_p is the specific heat in constant pressure $[Ws/kg°C]$, and k $[W/m°C]$ is the thermal conductivity of the domain.

The top, bottom and right boundaries are Dirichlet BCs,

$$u(x, 0.0, t) = u(x, 0.4, t) = 20°C, \forall x \in [0.2, 0.3] \cup \forall t \in [0, 200], \tag{3.46}$$

$$u(0.3, y, t) = 20°C, \forall y \in [0.0, 0.4] \cup \forall t \in [0, 200], \tag{3.47}$$

respectively. The left boundary is Neumann BC,

$$\hat{n} \cdot k\left(\frac{du}{dx} + \frac{du}{dy}\right) = 0, x = 0.2 \cup \forall y \in [0.0, 0.4] \cup \forall t \in [0, 200], \tag{3.48}$$

with ICs given as

$$u(x, y, 0) = 20°C \forall x \in [0.2, 0.3] \cup \forall y \in [0.0, 0.4]. \tag{3.49}$$

Table 3.12 provides the parameters used for defining the axisymmetric transient heat conduction PDE. $\dot{q}$ will represent a constant input function for the branch net of the DeepONet models, and will take a value within the given range.

Table 3.12 PDE parameters for the heat conduction problem given by equation 3.45.

Parameter	**Value**
k $[W/m°C]$	200
ρC_p $[Ws/°Cm^3]$	2.403×10^6
$\dot{q}$ $[W/m^3]$	$(2-8) \times 10^6$
u_{init} $[°C]$	20

3.4.2 Data-set Generation

The training and test data-sets consists of input-output triplets $[\dot{q}, (x, y, t), G(\dot{q})((x, y, t))]$, where $\dot{q}$ is a constant function, (x, y, t) is an evaluation location within the problem domain and $G(\dot{q})((x, y, t)) = u(x, y, t)$ is the temperature value to be predicted by the network for the given $\dot{q}$ at the (x, y) coordinate at time t. This data-set was obtained from Hitachi.

To generate the training and test data-sets, we first sample $n = 85$ and $n = 15$ different $\dot{q}$ values respectively, uniformly between the range specified in table 3.12. Using these values, we obtain the solutions providing the temperatures u at different (x, y) coordinates, and at different times t. This initial data-set is of size $(341097, 4)$ for (t, x, y, u), for each $\dot{q}$. We reduce the number of evaluation locations (x, y, t) to be passed to the trunk network during training by randomly selecting $p = 1000$ for the residual and ICs, $p = 400$ for the left and right boundaries, and $p = 100$ for the top and bottom boundaries, from each of the n files (a specific $\dot{q}$).

In summary, for each $\dot{q}^{(i)}$,

- $\{(x_{r,j}^{(i)}, y_{r,j}^{(i)}, t_{r,j}^{(i)})\}_{j=1}^{p=1000}$ are uniformly sampled within the spatial domain $[0.2, 0.3] \times [0.0, 0.4]$ at times $[0, 200]$ (residual),
- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)}, t_{b,j}^{(i)})\}_{j=1}^{p=400}$ are sampled from the left and right boundaries, each, at times $[0, 200]$ (Neumann and Dirichlet respectively),
- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)}, t_{b,j}^{(i)})\}_{j=1}^{p=100}$ are sampled from the bottom and top boundaries, each, at times $[0, 200]$ (Dirichlet),
- $\{(x_{ic,j}^{(i)}, y_{ic,j}^{(i)}, 0)\}_{j=1}^{p=1000}$ are uniformly sampled within the spatial domain $[0.2, 0.3]$ at time $[0]$ (IC).

This data-set is used by the DI-DeepONet when computing the data loss, $L_{data}(\theta)$ (3.52), and contains $[\dot{q}, (x, y, t), G(\dot{q})(x, y, t)]$ of size $(85 \times 3000, 1)$, $(85 \times 3000, 3)$ and $(85 \times 3000, 1)$ respectively.

For generating the training data-set for the PI-DeepONet,

- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)}, t_{b,j}^{(i)})\}_{j=1}^{p=400}$ are sampled from the right boundary at times $[0, 200]$ (Dirichlet),
- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)}, t_{b,j}^{(i)})\}_{j=1}^{p=100}$ are sampled from the bottom and top boundaries, each, at times $[0, 200]$ (Dirichlet),
- $\{(x_{ic,j}^{(i)}, y_{ic,j}^{(i)}, 0)\}_{j=1}^{p=1000}$ are uniformly sampled within the spatial domain $[0.2, 0.3]$ at time $[0]$ (IC),

in order to satisfy the data loss, $L_{data}(\theta)$ (3.62). The training data-set contains $[\dot{q}, (x, y, t), G(\dot{q})(x, y, t)]$ of size $(85 \times 1600, 1)$, $(85 \times 1600, 3)$ and $(85 \times 1600, 1)$ for the Dirichlet BCs and ICs, and is only required for computing $L_{data}(\theta)$.

We then randomly select $q = 400$ different (x, y, t) coordinates to satisfy the Neumann BC, as well as, $q = 1000$ different (x, y, t) coordinates to satisfy the PDE residual for $L_{physics}(\theta)$ (3.59). Thus,

- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)}, t_{b,j}^{(i)})\}_{j=1}^{q=400}$ are collocation points sampled from the left boundary at times $[0, 200]$ (Neumann).
- $\{(x_{r,j}^{(i)}, y_{r,j}^{(i)}, t_{r,j}^{(i)})\}_{j=1}^{q=1000}$ are collocation points sampled from within the spatial domain $[0.2, 0.3] \times [0.0, 0.4]$ at times $[0, 200]$ for computing the residual loss.

For generating the training data-set for the HPI-DeepONet,

- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)}, t_{b,j}^{(i)})\}_{j=1}^{p=400}$ are sampled from the left and right boundaries at times $[0, 200]$ (Neumann and Dirichlet respectively),
- $\{(x_{b,j}^{(i)}, y_{b,j}^{(i)}, t_{b,j}^{(i)})\}_{j=1}^{p=100}$ are sampled from the bottom and top boundaries, each, at times $[0, 200]$ (Dirichlet),
- $\{(x_{ic,j}^{(i)}, y_{ic,j}^{(i)}, 0)\}_{j=1}^{p=1000}$ are uniformly sampled within the spatial domain $[0.2, 0.3]$ at time $[0]$ (IC),

in order to satisfy the data loss, $L_{data}(\theta)$ (3.64). The training data-set contains $[\dot{q}, (x, y, t), G(\dot{q})(x, y, t)]$ of size $(85 \times 2000, 1)$, $(85 \times 2000, 3)$ and $(85 \times 2000, 1)$ for the Neumann and Dirichlet BC, as well as the ICs, and is only required for computing $L_{data}(\theta)$.

We then randomly select $q = 1000$ different (x, y, t) coordinates to satisfy the PDE residual for $L_{physics}(\theta)$(3.63). Thus,

- $\{(x_{r,j}^{(i)}, y_{r,j}^{(i)}, t_{r,j}^{(i)})\}_{j=1}^{q=1000}$ are collocation points sampled from within the spatial domain $[0.2, 0.3] \times [0.0, 0.4]$ at times $[0, 200]$ for computing the residual loss.

For testing, we use the entire spatial domain $[0.2, 0.3] \times [0.0, 0.4]$ at 3 time-steps $t = [0, 100, 200]s$. The testing data-set contains $[\dot{q}, (x, y, t), G(\dot{q})(x, y, t)]$ of size $(15 \times 1697 \times 3, 1)$, $(15 \times 1697 \times 3, 3)$ and $(15 \times 1697 \times 3, 1)$ respectively.

The parameters used for generating the data-sets are shown in the table 3.13.

Table 3.13 Parameters used for generating the training and test data-sets.

Parameter	DI-DeepONet	PI-DeepONet	HPI-DeepONet
n	85	85	85
p	3000	1600	2000
q	-	1400	1000

We also augment the training data-set by non-dimensionalizing the spatial and time coordinates (x, y, t), the source terms $\dot{q}$, and the temperature u values respectively, in order to make training the DeepONet models easier. The procedure used to non-dimensionalize the data-set is section 3.4.3. Note that for training the DI-DeepONet, the dimensional time coordinate t was used as it was observed to make training easier and improve prediction accuracy. This, however, was not observed for the PI-DeepONet or HPI-DeepONet.

Note:

- DI-DeepONet: Trained on paired input-output measurements for the domain, ICs, Dirichlet and Neumann BCs.
- PI-DeepONet: Trained without any paired training data (except for the ICs and Dirichlet BCs).
- HPI-DeepONet: Trained without any paired training data (except for the ICs, Dirichlet and Neumann BCs).

As a result, the PI-DeepONet contains 47% less training data (for predicting $G(\dot{q})(x, y, t)$ within the domain, and left boundary), while the HPI-DeepONet contains 33% less training data (for predicting $G(\dot{q})(x, y, t)$ within the domain), when compared to the DI-DeepONet. Additionally, as the ICs and Dirichlet BCs are known a priori, the training data for the PI-DeepONet can be assembled without first solving time-consuming FEA models or sensor measurements. However, this is not possible with the DI-DeepONet as the temperatures, u, within the domain, and along the left boundary at randomly generated collocation points, (x, y, t), are needed at $t > 0$. The HPI-DeepONet also requires the use of these time-consuming methods as u is required for (x, y, t) along the left boundary at $t > 0$. Thus, $G(\dot{q})(x, y, t)$ in the training data for the ICs and Dirichlet BCs are not obtained from the numerical solution, as its value is already known.

3.4.3 Non-dimensionalizing the Axisymmetric Heat Equation

To non-dimensionalize the heat equation given in equation 3.45, we first define new non-dimensional variables for the spatial terms x, y and temperature term u, i.e.

$$\hat{x} = \frac{x}{X}, \hat{y} = \frac{y}{X}, \hat{u} = \frac{u}{U}.$$

X is chosen to be the maximum value within the spatial domain, $0.4m$, and U is chosen to be the maximum value expected within the temperature domain, $150°C$.

The first derivative terms dx, dy, and du are thus,

$$dx = Xd\hat{x}, dy = Xd\hat{y}, du = Ud\hat{u}.$$

The time, t, and source term, $\dot{q}$, are non-dimensionalized by first dividing throughout by k,

$$\frac{\rho C_p}{k}\frac{du}{dt} - \frac{1}{x}\frac{d}{dx}\left(x\frac{du}{dx}\right) - \frac{d}{dy}\left(\frac{du}{dy}\right) = \frac{\dot{q}}{k}.$$

We then observed that the term $\frac{k}{\rho C_p X^2}$ has units $1/s$ and can be used to non-dimensionalize t to $\hat{t}$, i.e.

$$\hat{t} = \frac{k}{\rho C_p X^2}t,$$

with first derivative,

$$dt = \frac{\rho C_p X^2}{k}d\hat{t}.$$

Similarly, for $\dot{q}$ to β,

$$\beta = \frac{X^2}{U}\frac{\dot{q}}{k}.$$

By substitution, we arrive at the non-dimensional axisymmetric heat equation,

$$\frac{U}{X^2} \cdot \frac{d\hat{u}}{d\hat{t}} - \frac{U}{X^2} \cdot \frac{1}{\hat{x}}\frac{d}{dx}\left(\hat{x}\frac{d\hat{u}}{d\hat{x}}\right) - \frac{U}{X^2} \cdot \frac{d}{dy}\left(\frac{d\hat{u}}{d\hat{y}}\right) = \frac{U}{X^2} \cdot \beta, \tag{3.50}$$

or simply,

$$\frac{d\hat{u}}{d\hat{t}} - \frac{1}{x}\frac{d}{dx}\left(\hat{x}\frac{d\hat{u}}{d\hat{x}}\right) - \frac{d}{dy}\left(\frac{d\hat{u}}{d\hat{y}}\right) = \beta. \tag{3.51}$$

3.4.4 Loss Function

Recall that G_θ represents the approximation of the non-dimensional heat operator by each DeepONet, using the non-dimensional spatial and time coordinates $(\hat{x}, \hat{y}, \hat{t})$, the source terms β, and the temperature $\hat{u}$ values respectively. Note that the DI-DeepONet uses dimensional time t for training, but this not reflected in equations 3.52 to 3.56 for ease of notation.

DI-DeepONet Loss Function

The loss function for the DI-DeepONet is given by

$$L_{data}(\theta) = L_{r,data}(\theta) + L_{dc,data}(\theta) + L_{n,data}(\theta) + L_{ic,data}(\theta), \tag{3.52}$$

to satisfy the constraints for the residual r, Dirichlet dc, and Neumann n boundaries, and the ICs ic. Each term is defined as

$$L_{r,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(\beta^{(i)})(\hat{x}_{r,j}^{(i)}, \hat{y}_{r,j}^{(i)}, \hat{t}_{r,j}^{(i)}) - G(\beta^{(i)})(\hat{x}_{r,j}^{(i)}, \hat{y}_{r,j}^{(i)}, \hat{t}_{r,j}^{(i)})\right|^2, \tag{3.53}$$

$$L_{dc,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(\beta^{(i)})(\hat{x}_{dc,j}^{(i)}, \hat{y}_{dc,j}^{(i)}, \hat{t}_{dc,j}^{(i)}) - G(\beta^{(i)})(\hat{x}_{dc,j}^{(i)}, \hat{y}_{dc,j}^{(i)}, \hat{t}_{dc,j}^{(i)})\right|^2, \tag{3.54}$$

$$L_{n,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(\beta^{(i)})(\hat{x}_{n,j}^{(i)}, \hat{y}_{n,j}^{(i)}, \hat{t}_{n,j}^{(i)}) - G(\beta^{(i)})(\hat{x}_{n,j}^{(i)}, \hat{y}_{n,j}^{(i)}, \hat{t}_{n,j}^{(i)})\right|^2, \tag{3.55}$$

$$L_{ic,data}(\theta) = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}\left|G_\theta(\beta^{(i)})(\hat{x}_{ic,j}^{(i)}, \hat{y}_{ic,j}^{(i)}, 0) - G(\beta^{(i)})(\hat{x}_{ic,j}^{(i)}, \hat{y}_{ic,j}^{(i)}, 0)\right|^2. \tag{3.56}$$

PI-DeepONet Loss Function

Let R_θ be the heat PDE residual which is given by

$$\begin{aligned} R_\theta^{(i)}(\hat{x}, \hat{y}, t) = \frac{dG_\theta(\beta^{(i)})(\hat{x}, \hat{y}, \hat{t})}{d\hat{t}} &- \frac{1}{x}\left(\hat{x} \cdot \frac{d^2G_\theta(\beta^{(i)})(\hat{x}, \hat{y}, \hat{t})}{d\hat{x}^2} + \frac{dG_\theta(\beta^{(i)})(\hat{x}, \hat{y}, \hat{t})}{d\hat{x}}\right) \\ &- \frac{d^2G_\theta(\beta^{(i)})(\hat{x}, \hat{y}, \hat{t})}{d\hat{y}^2} - \beta^{(i)}, \end{aligned} \tag{3.57}$$

and let N_θ be the residual for the Neumann BC given by

$$N_\theta^{(i)}(\hat{x}, \hat{y}, \hat{t}) = \hat{n} \cdot \hat{k}\left(\frac{d\hat{u}^{(i)}(\hat{x}, \hat{y}, \hat{t})}{d\hat{x}} + \frac{d\hat{u}^{(i)}(\hat{x}, \hat{y}, \hat{t})}{d\hat{y}}\right), \tag{3.58}$$

where $\hat{k}$ is a scaling factor chose to be $\hat{k} = 1$.

The loss function for the PI-DeepONet can then be defined as

$$L(\theta) = L_{data}(\theta) + L_{physics}(\theta),$$

where $L_{physics}(\theta)$ is given by

$$L_{physics}(\theta) = L_{n,physics}(\theta) + L_{r,physics}(\theta), \tag{3.59}$$

with each term given as,

$$L_{n,physics}(\theta) = \frac{1}{nq}\sum_{i=1}^{n}\sum_{j=1}^{q}\left|N_{\theta}^{(i)}(\hat{x}_{n,j}^{(i)}, \hat{y}_{n,j}^{(i)}, \hat{t}_{n,j}^{(i)})\right|^2, \tag{3.60}$$

$$L_{r,physics}(\theta) = \frac{1}{nq}\sum_{i=1}^{n}\sum_{j=1}^{q}\left|R_{\theta}^{(i)}(\hat{x}_{r,j}^{(i)}, \hat{y}_{r,j}^{(i)}, \hat{t}_{r,j}^{(i)})\right|^2. \tag{3.61}$$

$L_{data}(\theta)$ composed of the collocation points for the ICs and Dirichlet boundaries given by

$$L_{data}(\theta) = L_{dc,data}(\theta) + L_{ic,data}(\theta). \tag{3.62}$$

HPI-DeepONet Loss Function

The loss function for the HPI-DeepONet is given by

$$L(\theta) = L_{data}(\theta) + L_{physics}(\theta),$$

where $L_{physics}(\theta)$ is defined only for the heat PDE residual, i.e.

$$L_{physics}(\theta) = L_{r,physics}(\theta). \tag{3.63}$$

$L_{data}(\theta)$ is composed of the collocation points for the ICs, as well as, Dirichlet and Neumann boundaries given by

$$L_{data}(\theta) = L_{dc,data}(\theta) + L_{n,data}(\theta) + L_{ic,data}(\theta). \tag{3.64}$$

Defining the heat PDE Loss, $L_{r,physics}(\theta)$, in Modulus

In listing 15, we define the heat PDE loss $L_{r,physics}(\theta)$ in Modulus, using a torch.nn.Module, and a class called HeatPDE, which uses the input and output tensors of our DeepONet model to compute the residual R_θ within its forward method. This module is then incorporated into the computational graph created by Modulus using a Node as shown in listing 16. The PDE loss $L_{r,physics}(\theta)$ is then added as a constraint using Modulus's DeepONetConstraint as shown in listing 17.

```python
class HeatPDE(torch.nn.Module):
    "Custom 2-dimensional Heat PDE definition for HPI-DeepONet"

    def __init__(self):
        super().__init__()

    def forward(self, input_var: Dict[str, torch.Tensor])
        -> Dict[str, torch.Tensor]:
        # Get inputs (non-dimensional)
        x = input_var["x"]
        hq = input_var["hq"]
        # Compute gradients (non-dimensional)
        dudt = input_var["u__t"]
        dudx = input_var["u__x"]
        dduddx = input_var["u__x__x"]
        dduddy = input_var["u__y__y"]

        # Compute (non-dimensional) Heat equation
        heat_pde = ((dudt) - (1/x)*(x*dduddx + dudx) - dduddy - hq)

        # Return heat
        output_var = {
            "heat": heat_pde,
        }
        return output_var
```

Listing 15 Defining the Heat PDE in Modulus.

```python
# Incorporate Heat module into Modulus using a Node
heat_node = Node(
    inputs=["x", "hq", "u__t", "u__x", "u__x__x", "u__y__y"],
    outputs=["heat"],
    evaluate=HeatPDE(),
    name="Heat Node",
)
```

Listing 16 Adding the Heat Node into the computational graph.

```python
# Residual
residual = DeepONetConstraint.from_numpy(
    nodes=nodes,
    invar={
        "hq": hq_pi_train_nd,
        "x": xx_pi_train_nd,
        "y": yy_pi_train_nd,
        "t": tt_pi_train_nd,
    },
    outvar={"heat": np.zeros_like(hq_pi_train_nd)},
    batch_size=cfg.batch_size.interior,
    lambda_weighting={
        "heat": 1.0*(np.ones_like(hq_pi_train_nd)),
    },
)
domain.add_constraint(residual, "residual")
```

Listing 17 Adding the Heat PDE loss $L_{r,physics}(\theta)$ as an additional constraint.

Defining the Neumann BC for the PI-DeepONet in Modulus

In listing 18, we define the Neumann BC in Modulus, using a torch.nn.Module, and a class called NeumannBC, which uses the input and output tensors of our DeepONet model to compute N_θ, within the forward method. In order to obtain the unit normal vector, $\hat{n}$, for the derivatives of u on the left boundary, $\frac{du}{dx}$ and $\frac{du}{dy}$, we use the 'sample boundary' method from the 'rectangular geometry' instance representing our domain, created using Modulus' geometry module. This module is then incorporated into the computational graph created by Modulus using a Node as shown in listing 19. NeumannBC is then added as a constraint using Modulus's DeepONetConstraint for computing the loss, $L_{n,physics}(\theta)$ as shown in listing 20.

```
class NeumannBC(torch.nn.Module):
    "Custom Neumann BC for 2D axisymmetric Heat Equation"

    def __init__(self, geo, surf, K):
        super().__init__()
        self.geo = geo
        self.surf = surf
        self.K = K

    def forward(self, input_var: Dict[str, torch.Tensor])
        -> Dict[str, torch.Tensor]:
        # Compute gradients (non-dimensional)
        dudx = input_var["u__x"]
        dudy = input_var["u__y"]
        # Get normals
        xx = Symbol("x")
        samples = self.geo.sample_boundary(10, Eq(xx, self.surf))
        normal_x = samples["normal_x"][0, 0]
        normal_y = samples["normal_y"][0, 0]

        # Compute Neumann BC
        normal_gradient_u = -(self.K)*(normal_x*dudx + normal_y*dudy)

        # Return Neumann BC
        output_var = {
            "normal_gradient_u": normal_gradient_u,
        }
        return output_var
```

Listing 18 Defining the Neumann BC in Modulus.

```
# Neumann BC
NBC_node = Node(
    inputs=["u__x", "u__y"],
    outputs=["normal_gradient_u"],
    evaluate=NeumannBC(
        geo=geo,
        surf=xx_range_scaled[0],
        K=k_nd,
    ),
    name="Neumann Node",
)
```

Listing 19 Adding the Neumann Node into the computational graph.

```python
# Neumann boundary condition, x=0.2, left wall
NBC_1 = DeepONetConstraint.from_numpy(
    nodes=nodes,
    invar={
        "hq": hq_train_nm_nd,
        "x": xx_train_nm_nd,
        "y": yy_train_nm_nd,
        "t": tt_train_nm_nd,
    },
    outvar={"normal_gradient_u": np.zeros_like(hq_train_nm_nd)},
    batch_size=cfg.batch_size.NBC,
    lambda_weighting={
        "normal_gradient_u": 1.0*np.ones_like(hq_train_nm_nd),
    },
)
domain.add_constraint(NBC_1, "NBC_1")
```

Listing 20 Adding the Neumann loss $L_{n,physics}(\theta)$ as an additional constraint.

Defining the Dirichlet BCs in Modulus

The Dirichlet BCs for the top, bottom and right boundaries are easily added as constraints using Modulus's DeepONetConstraint for each DeepONet. Note that the Neumann BC for the right boundary is similarly enforced for the DI-DeepONet and HPI-DeepONet, as well as, the heat PDE residual for the DI-DeepONet. An example is shown in listing 21 for the right boundary.

```python
# Dirichlet boundary condition 1, x=0.3, right wall
DBC_1 = DeepONetConstraint.from_numpy(
    nodes=nodes,
    invar={
        "hq": hq_train_dbR_nd,
        "x": xx_train_dbR_nd,
        "y": yy_train_dbR_nd,
        "t": tt_train_dbR_nd,
    },
    outvar={"u": np.ones_like(hq_train_dbR_nd)*T_dbc_scaled},
    batch_size=cfg.batch_size.NBC,
    lambda_weighting={
        "u": 1.0*(np.ones_like(hq_train_dbR_nd)),
    },
)
domain.add_constraint(DBC_1, "DBC_1")
```

Listing 21 Adding the Dirichlet BC constraint $L_{dc,data}(\theta)$ in Modulus.

3.4.5 Training

Recall (refer to section 2.2.1):

- The branch network takes $\dot{q}$ as input and returns a features embedding vector br as output.
- The trunk network takes (x, y, t) as input and returns a features embedding vector tr as output.
- These outputs are then merged together by a dot product to produce the final prediction of the heat operator by each DeepONet, i.e. $G_\theta = br \cdot tr$.

The network architecture and training parameters used are summarized in table 3.14.

Table 3.14 Network architecture and training parameters used by each DeepONet.

Name	Value
Architecture	Fully-connected Neural Network
Depth	10
Width	128
Optimiser	Adam
Learning Rate	1.0×10^{-3} with decay
Training Steps	50000
Activation Function	SILU

DI-DeepONet Training

The DI-DeepONet loss function is computed as a sum of the point-wise differences (2.2) for:

- Predicting the temperature u for the Dirichlet BCs as given by the data loss $L_{dc,data}$,
- Predicting the temperature u for the Neumann BC as given by the data loss $L_{n,data}$,
- Predicting the temperature u for the ICs as given by the data loss $L_{ic,data}$,
- Predicting the PDE residual for the heat equation as given by the data loss $L_{r,data}$.

As a result, we apply lambda weightings to the loss function for each loss term, i.e.

$$L(\theta) = \lambda_{dc,data} L_{dc,data}(\theta) + \lambda_{n,data} L_{n,data}(\theta) + \lambda_{ic,data} L_{ic,data}(\theta) + \lambda_{r,data} L_{r,data}(\theta), \tag{3.65}$$

where $\lambda_{dc,data}$, $\lambda_{n,data}$, $\lambda_{ic,data}$, and $\lambda_{r,data}$ are given in table 3.15.

Table 3.15 Lambda weightings used by the DI-DeepONet.

Name	Value
$\lambda_{dc,data}$	2.0
$\lambda_{n,data}$	1.0
$\lambda_{ic,data}$	1.0
$\lambda_{r,data}$	$1.0 + 2\hat{x}, \hat{x} \in [0.5, 0.75]$

PI-DeepONet Training

The PI-DeepONet loss function is computed as a sum of the point-wise differences for:

- Predicting the temperature u for the Dirichlet BCs as given by the data loss $L_{dc,data}$,
- Predicting the temperature u for the Neumann BC, N_θ, as given by the physics loss $L_{n,physics}$,
- Predicting the temperature u for the ICs as given by the data loss $L_{ic,data}$,
- Predicting the PDE residual for the heat equation, R_θ, as given by the physics loss $L_{r,physics}$.

As a result, we apply lambda weightings to the loss function for each loss term, i.e.

$$L(\theta) = \lambda_{dc,data} L_{dc,data}(\theta) + \lambda_{n,physics} L_{n,physics}(\theta) + \lambda_{ic,data} L_{ic,data}(\theta) + \lambda_{r,physics} L_{r,physics}(\theta), \tag{3.66}$$

where $\lambda_{dc,data}$, $\lambda_{n,physics}$, $\lambda_{ic,data}$, and $\lambda_{r,physics}$ are given in table 3.16.

Table 3.16 Lambda weightings used by the PI-DeepONet.

Name	Value
$\lambda_{dc,data}$	1.0
$\lambda_{n,physics}$	1.0
$\lambda_{ic,data}$	1.0
$\lambda_{r,physics}$	1.0×10^{-3}

HPI-DeepONet Training

The HPI-DeepONet loss function is computed as a sum of the point-wise differences for:

- Predicting the temperature u for the Dirichlet BCs as given by the data loss $L_{dc,data}$,
- Predicting the temperature u for the Neumann BC as given by the data loss $L_{n,data}$,
- Predicting the temperature u for the ICs as given by the data loss $L_{ic,data}$,
- Predicting the PDE residual for the heat equation, R_θ as given by the physics loss $L_{r,physics}$.

As a result, we apply lambda weightings to the loss function for each loss term, i.e.

$$L(\theta) = \lambda_{dc,data} L_{dc,data}(\theta) + \lambda_{n,data} L_{n,data}(\theta) + \lambda_{ic,data} L_{ic,data}(\theta) + \lambda_{r,physics} L_{r,physics}(\theta), \quad (3.67)$$

where $\lambda_{dc,data}$, $\lambda_{n,data}$, $\lambda_{ic,data}$, and $\lambda_{r,physics}$ are given in table 3.17.

Table 3.17 Lambda weightings used by the HPI-DeepONet.

Name	Value
$\lambda_{dc,data}$	1.0
$\lambda_{n,data}$	1.0
$\lambda_{ic,data}$	1.0
$\lambda_{r,physics}$	1.0×10^{-3}

These lambda values were selected by training the network with uniform lambda weightings and observing the loss obtained from each loss term on the validation data-set. For the DI-DeepONet, we observed higher loss along the Dirichlet boundaries, as well as, within the domain with a drop in performance as we move towards the right boundary. $\lambda_{dc,data}$ and $\lambda_{r,data}$ (table 3.15) are chosen to ensure that the parameter updates, θ, during training weighed $L_{dc,data}$ and $L_{r,data}$ more for the DI-DeepONet, as compared to the other loss terms. $\lambda_{r,data}$ represents a monotonically increasing function with output ranging between $[2.0, 2.5]$ as we move towards the right boundary.

Similarly for the PI-DeepONet and HPI-DeepONet, we observed that the residual dominated the predictions made by the network, with high losses observed along the boundaries and the ICs. $\lambda_{r,physics}$ (tables 3.16 and 3.17) is chosen to ensure that the parameter updates, θ, during training weighed $L_{r,physics}$ less for the PI-DeepONet and HPI-DeepONet, as compared to the other loss terms. These lambda weightings were obtained by trial and error.

3.4.6 Results

The temperature field $\tilde{u}$ predicted by the DI-DeepONet, PI-DeepONet and HPI-DeepONet are compared to the true u, given by the FEM solution using Modulus' Validators. The results are shown for 1 source $\dot{q} = 6.940 \times 10^6$, at 3 different time-steps $t = [0, 100, 200]s$ in the figures 3.10, 3.11, and 3.12 for the

DI-DeepONet, PI-DeepONet and HPI-DeepONet respectively. We also determine the RRMSE between $\tilde{u}$ and the ground truth temperature u throughout the domain at 6 different sources $\dot{q}$, at 3 different time-steps $t = [0, 100, 200]s$ as shown in tables 3.18, 3.19, and 3.20 for the DI-DeepONet, PI-DeepONet and HPI-DeepONet respectively.

We observe:

- DI-DeepONet (3.10) achieved reasonable accuracy throughout the domain, which degrades as we approach the boundaries of the domain. The accuracy also degrades as $\dot{q}$ increases for each time-step in $t = [0, 100, 200]$. The accuracy also degrades as t increases.
- PI-DeepONet (3.11) achieved reasonable accuracy throughout the domain, which degrades as we approach the right boundary. The accuracy also degrades as $\dot{q}$ increases for each time-step in $t = [0, 100, 200]$. The accuracy improves as t increases.
- HPI-DeepONet (3.12) achieved reasonable accuracy throughout the domain, which degrades as we approach the boundaries of the domain. The accuracy also degrades as $\dot{q}$ increases for each time-step in $t = [0, 100, 200]$. The accuracy improves as t increases.

Table 3.18 RRMSE for DI-DeepONet at $t = [0, 100, 200]s$ for 6 different $\dot{q}$.

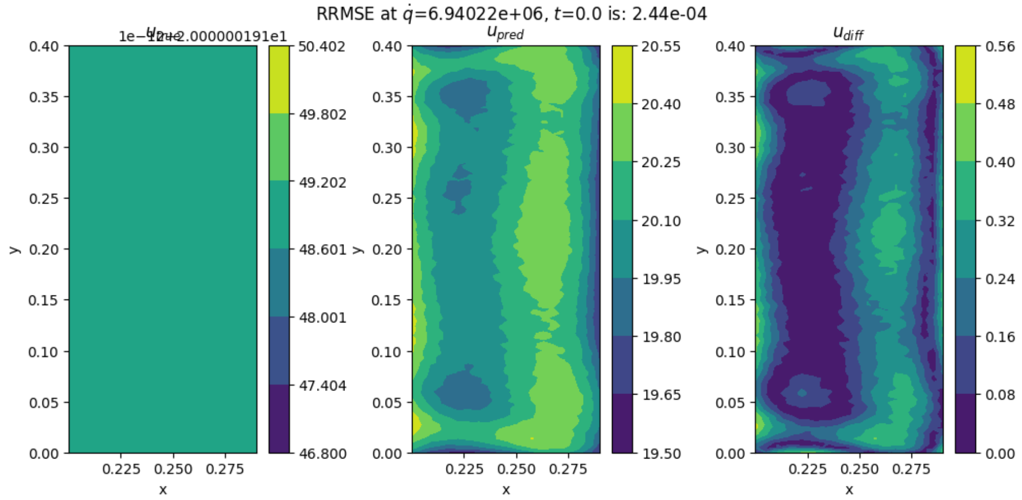
$\dot{q} \times 10^6$	$t = 0$	$t = 100$	$t = 200$
2.138	5.305×10^{-3}	9.572×10^{-3}	15.46×10^{-3}
3.544	5.758×10^{-3}	12.98×10^{-3}	20.36×10^{-3}
3.841	6.007×10^{-3}	13.62×10^{-3}	21.21×10^{-3}
6.073	8.778×10^{-3}	16.53×10^{-3}	25.05×10^{-3}
6.940	10.06×10^{-3}	17.32×10^{-3}	25.97×10^{-3}
7.193	10.53×10^{-3}	17.54×10^{-3}	26.21×10^{-3}

Table 3.19 RRMSE for PI-DeepONet at $t = [0, 100, 200]s$ for 6 different $\dot{q}$.

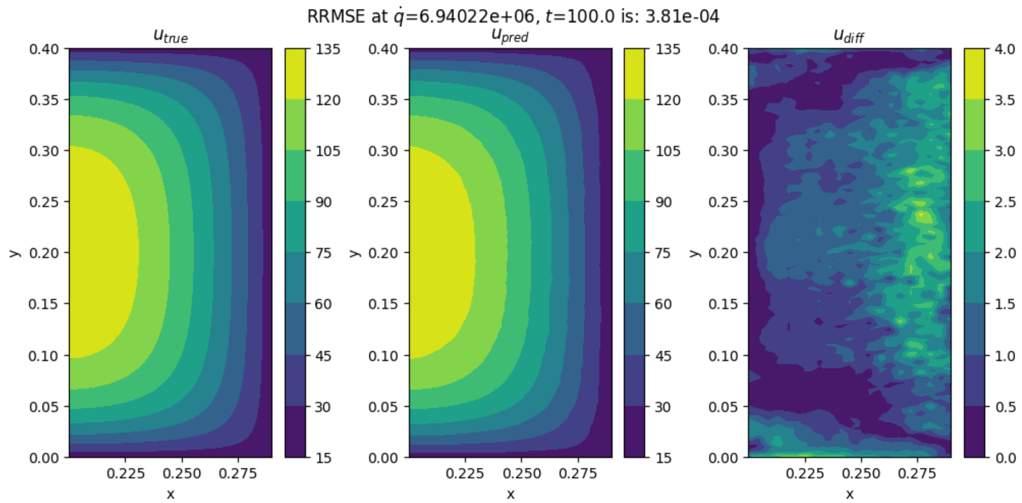
$\dot{q} \times 10^6$	$t = 0$	$t = 100$	$t = 200$
2.138	7.211×10^{-3}	2.580×10^{-3}	1.616×10^{-3}
3.544	11.78×10^{-3}	1.575×10^{-3}	1.192×10^{-3}
3.841	12.66×10^{-3}	1.658×10^{-3}	1.209×10^{-3}
6.073	19.26×10^{-3}	1.916×10^{-3}	1.616×10^{-3}
6.940	21.80×10^{-3}	2.031×10^{-3}	1.634×10^{-3}
7.193	22.47×10^{-3}	2.069×10^{-3}	1.647×10^{-3}

Table 3.20 RRMSE for HPI-DeepONet at $t = [0, 100, 200]s$ for 6 different $\dot{q}$.

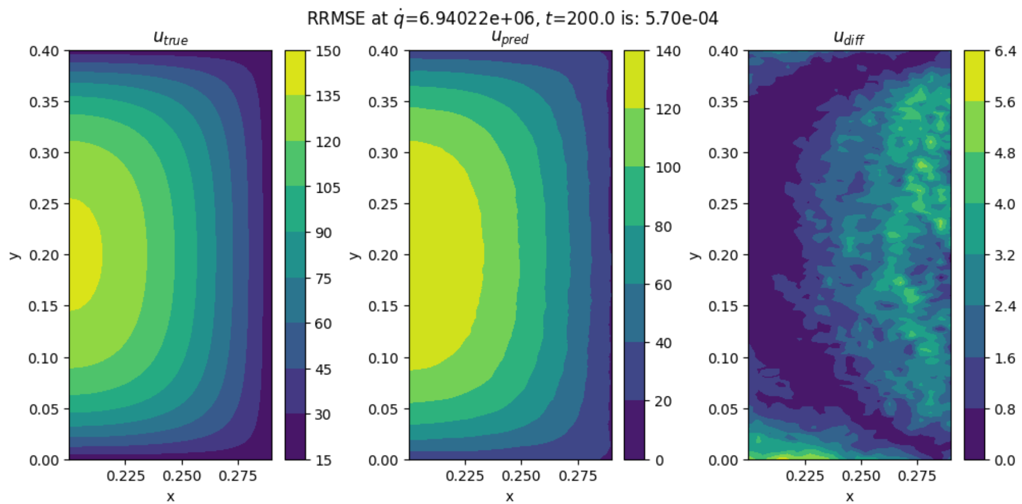
$\dot{q} \times 10^6$	$t = 0$	$t = 100$	$t = 200$
2.138	9.065×10^{-3}	1.600×10^{-3}	2.723×10^{-3}
3.544	14.62×10^{-3}	1.326×10^{-3}	1.630×10^{-3}
3.841	15.81×10^{-3}	1.489×10^{-3}	1.435×10^{-3}
6.073	24.45×10^{-3}	2.228×10^{-3}	1.386×10^{-3}
6.940	27.49×10^{-3}	2.627×10^{-3}	1.517×10^{-3}
7.193	28.40×10^{-3}	2.831×10^{-3}	1.612×10^{-3}



(a) $\dot{q} = 6.940 \times 10^6$ and $t = 0$

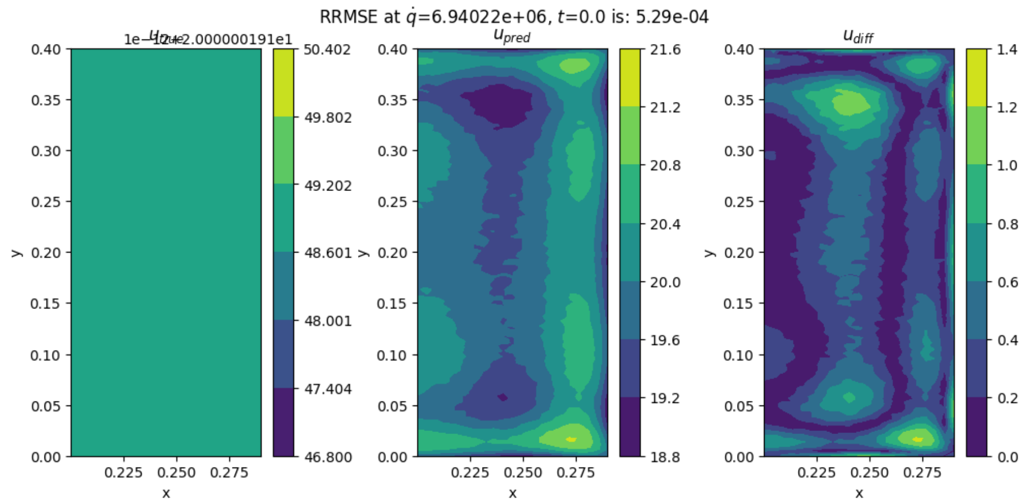


(b) $\dot{q} = 6.940 \times 10^6$ and $t = 100$

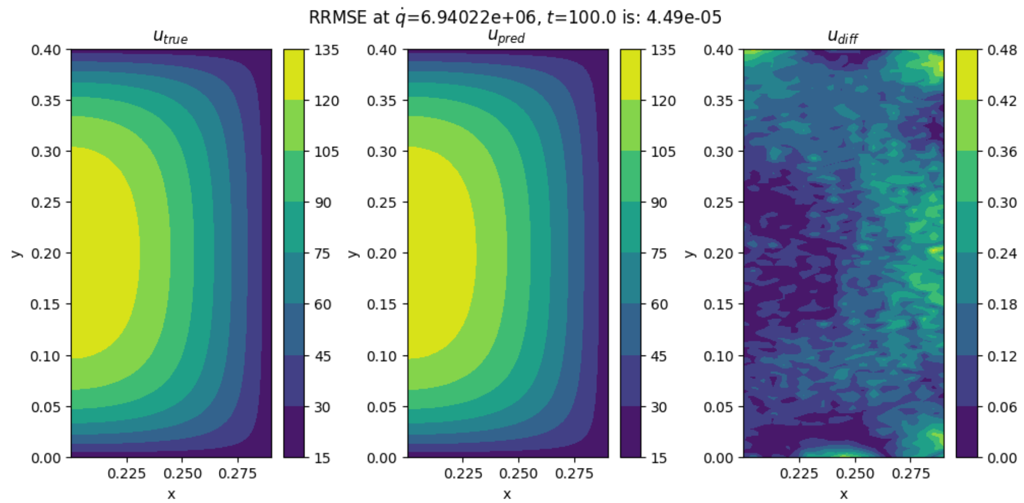


(c) $\dot{q} = 6.940 \times 10^6$ and $t = 200$

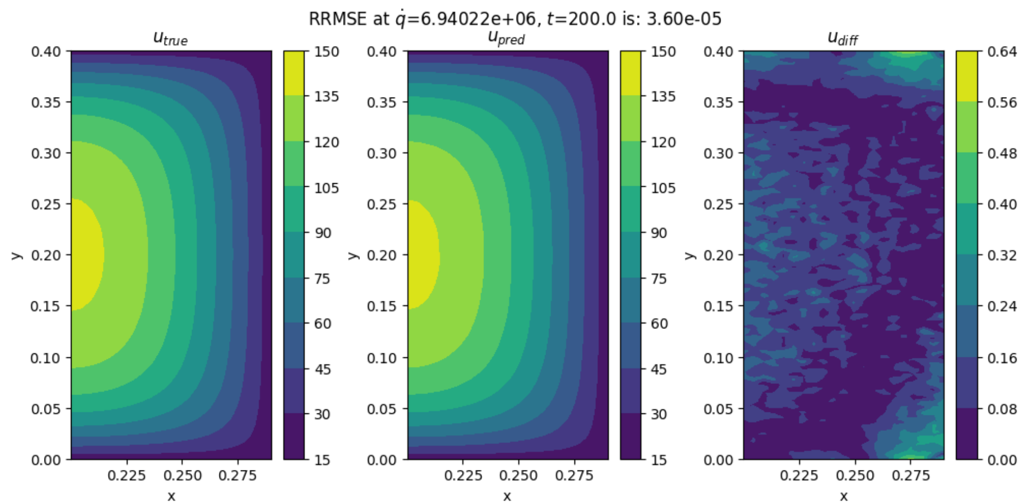
Figure 3.10 Temperature Field Predicted by DI-DeepONet at $t = [0, 100, 200]s$, with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right)



(a) $\dot{q} = 6.940 \times 10^6$ and $t = 0$

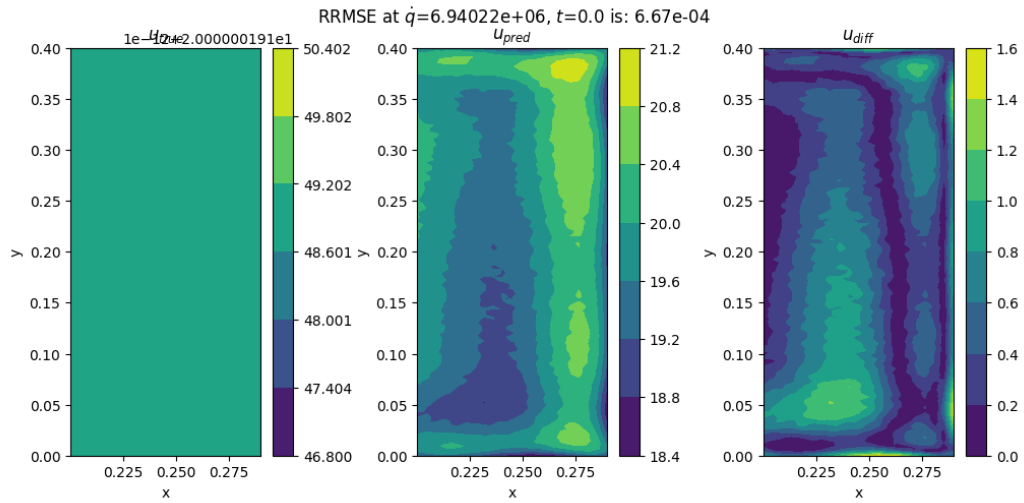


(b) $\dot{q} = 6.940 \times 10^6$ and $t = 100$

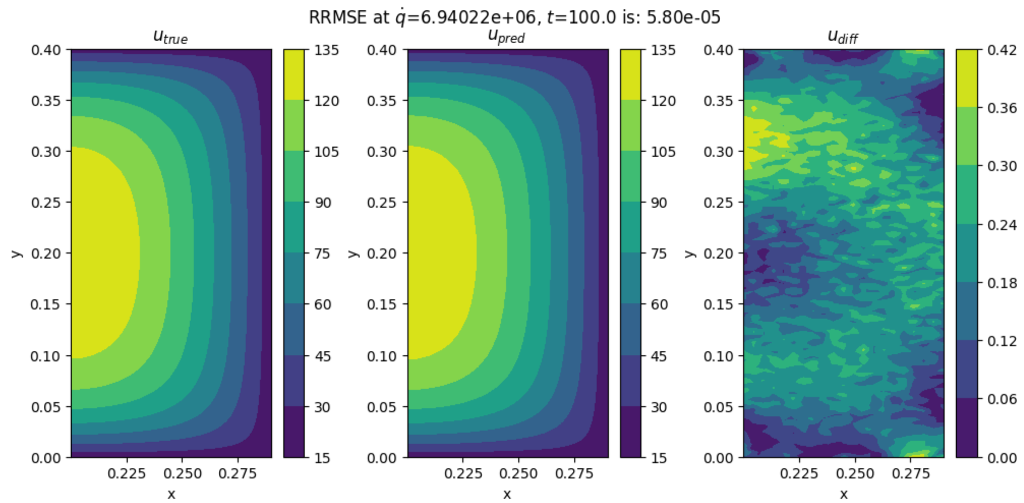


(c) $\dot{q} = 6.940 \times 10^6$ and $t = 200$

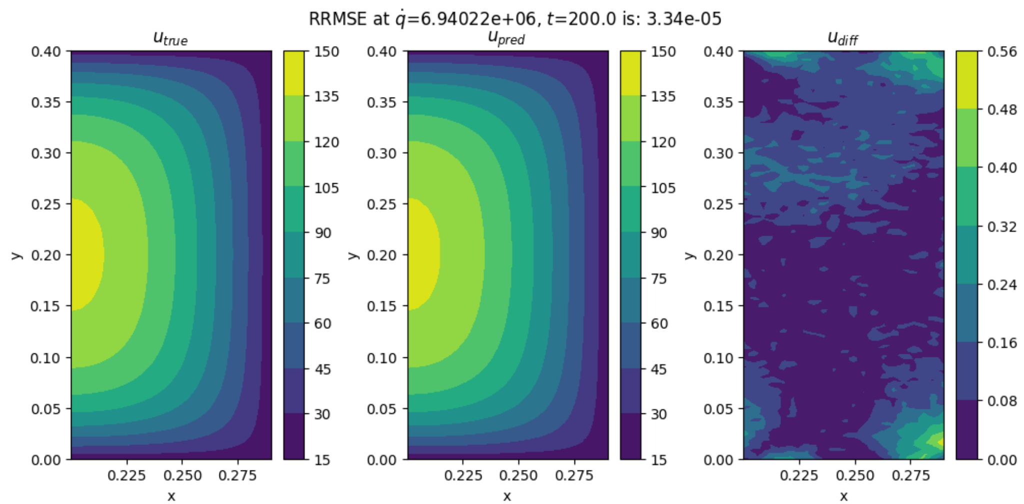
Figure 3.11 Temperature Field Predicted by PI-DeepONet at $t = [0, 100, 200]s$, with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right)



(a) $\dot{q} = 6.940 \times 10^6$ and $t = 0$



(b) $\dot{q} = 6.940 \times 10^6$ and $t = 100$



(c) $\dot{q} = 6.940 \times 10^6$ and $t = 200$

Figure 3.12 Temperature Field Predicted by HPI-DeepONet at $t = [0, 100, 200]s$, with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right)

Each DeepONet achieved reasonable accuracy on the validation data-set, however, the PI-DeepONet performed best and closely followed by the HPI-DeepONet. This indicates that the heat operator predicted by the networks, G_θ, is better satisfied by composing the loss function, L, with objectives for satisfying the physics of the axisymmetric transient heat conduction equation defined in section 3.4.1. L (section 3.4.4) for the PI-DeepONet is composed of almost 50% from physics for the heat PDE and the Neumann BC, while the remaining is supplied by data for the ICs and Dirichlet BCs. L (section 3.4.4) for the HPI-DeepONet is composed of 33% from physics for the heat PDE and the remainder supplied by data for the ICs, Neumann and Dirichlet BCs. L (section 3.4.4) for the DI-DeepONet has 0% from physics and is composed only of data for satisfying the heat PDE, ICs, Neumann and Dirichlet BCs. This may explain the reduced performance compared to the PI-DeepONet and HPI-DeepONet.

The accuracy for all DeepONets degrades as $\dot{q}$ increases, which may be due to larger temperature gradients generated within the domain u_x, u_y, u_{xx}, u_{yy}, and across time u_t. These large gradients could pose a challenge for the DeepONet models to learn when trying to satisfy the transient heat equation. In particular, $L_{physics}$ (3.59), for the PI-DeepONet is composed of these gradient terms for satisfying the heat PDE residual, R_θ (3.57), and the Neumann residual, N_θ (3.58), while $L_{physics}$ (3.63) for the HPI-DeepONet only has to satisfy the heat PDE residual, R_θ. Note that the heat PDE residual is satisfied by $L_{r,data}$ (3.52) for the DI-DeepONet which composed of training data, and not directly related to these gradients.

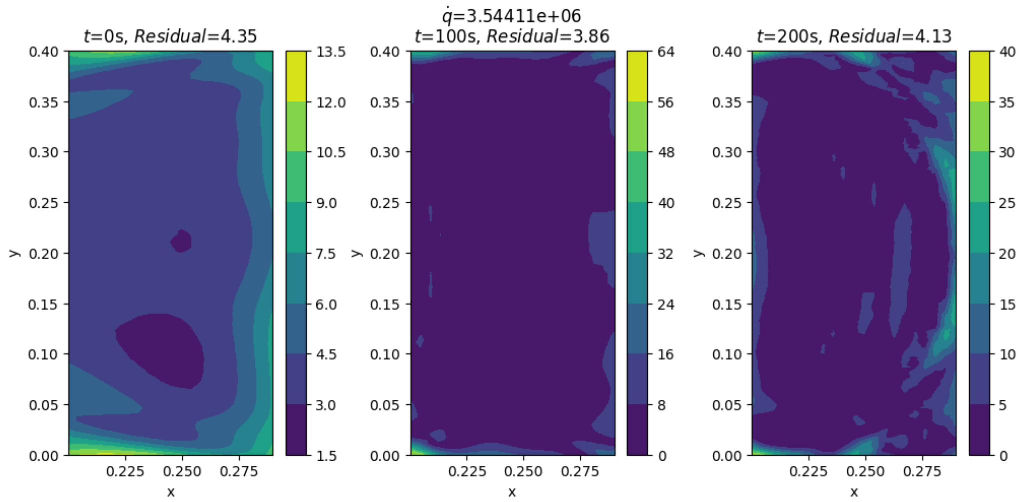
As the time, t, increases, the performance of the DI-DeepONet decreases, where as the performance of the PI-DeepONet and HPI-DeepONet improves. This indicates that u_t is better satisfied by incorporating physics (PI-DeepONet and HPI-DeepONet) into the models, as opposed to from data (DI-DeepONet). However, at $t = 0$ (ICs), the DI-DeepONet performs consistently better, which indicates that there may be a conflict with the heat PDE loss term, $L_{r,physics}$ (3.61), for PI-DeepONet and HPI-DeepONet and the IC loss term, $L_{ic,data}$ (3.56), even though $L_{r,physics}$ is supplied collocation points where $t > 0$.

The PDE residual, R_θ, within the domain, at 3 time-steps $t = [0, 100, 200]s$ for each DeepONet is shown in figure 3.13. Recall that the PI-DeepONet and HPI-DeepONet is trained with this loss term which acts as a regularization mechanism biasing our output to satisfy this heat PDE constraint, while the DI-DeepONet is not, and trained only using data. Also note that, N_θ is also an additional regularization mechanism added to the PI-DeepONet, as opposed to the HPI-DeepONet, to satisfy the Neumann constraint. After training, the PDE residual is satisfied more accurately for the PI-DeepONet and HPI-DeepONet by 2 orders of magnitude better than the DI-DeepONet at each t. An Inferencer was used to track the PDE partial derivatives, u_t, u_x, u_y, u_{xx} and u_{yy} within the domain during training, in order to produce these plots.

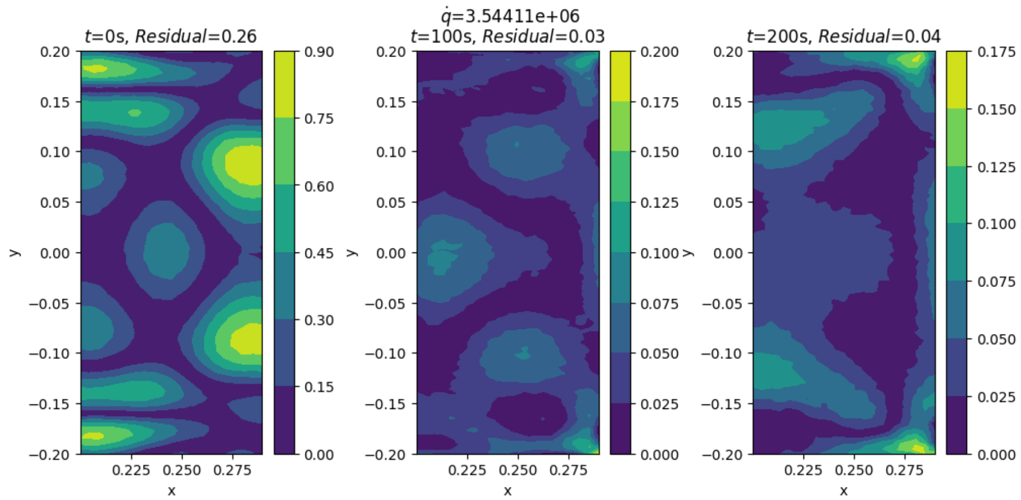
u_t, u_x, u_y, u_{xx} and u_{yy} at $t = 200$ within the domain is also provided for each DeepONet in figure 3.14. We observe that u_x, u_y, u_{xx} and u_{yy} have a more similar profiles for the PI-DeepONet and HPI-DeepONet, as compared to the DI-DeepONet which is unable to adequately predict these terms. In particular, the higher derivatives, u_{xx} and u_{yy}, are unable to be formed by the DI-DeepONet, with u_x, u_y having dull profiles. The PI-DeepONet and HPI-DeepONet are able to form sharp clear profiles for each term. As previously mentioned, $L_{physics}$ for the PI-DeepONet and HPI-DeepONet is constrained to learn these terms, where as $L_{r,data}$ for the DI-DeepONet is only required to satisfy u. As a result, the inability of the DI-DeepONet to adequately satisfy these terms is expected.

$u_x \neq 0$ for the HPI-DeepONet at the left corners of the domain as compared to the PI-DeepONet where $u_x = 0$. This term is required for the Neumann BC, which is an added constraint, N_θ, for the PI-DeepONet, as opposed to the HPI-DeepONet which enforced this term through data.

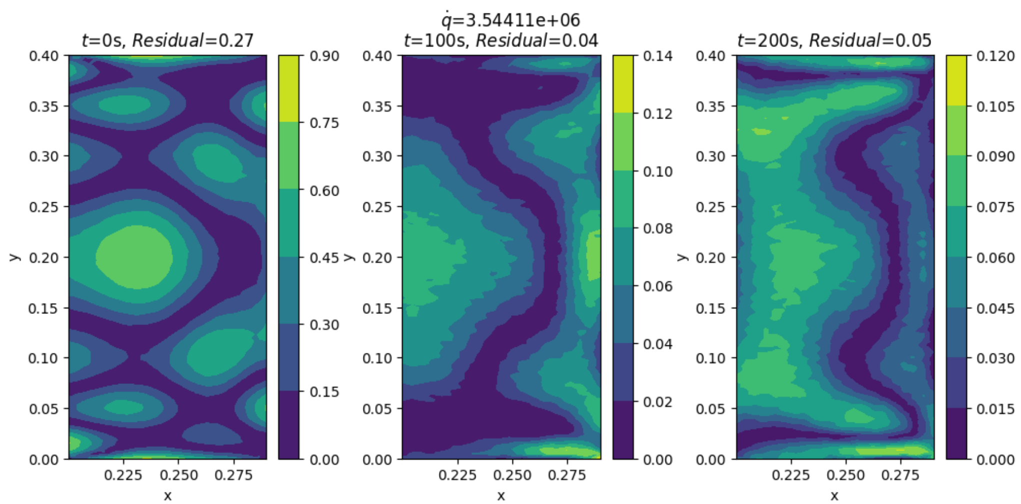
The temperature field $\tilde{u}$ predicted for 2 sources $\dot{q}$ at the extremes of the range given in table 3.12,



(a) DI-DeepONet at $\dot{q} = 3.544 \times 10^6$.

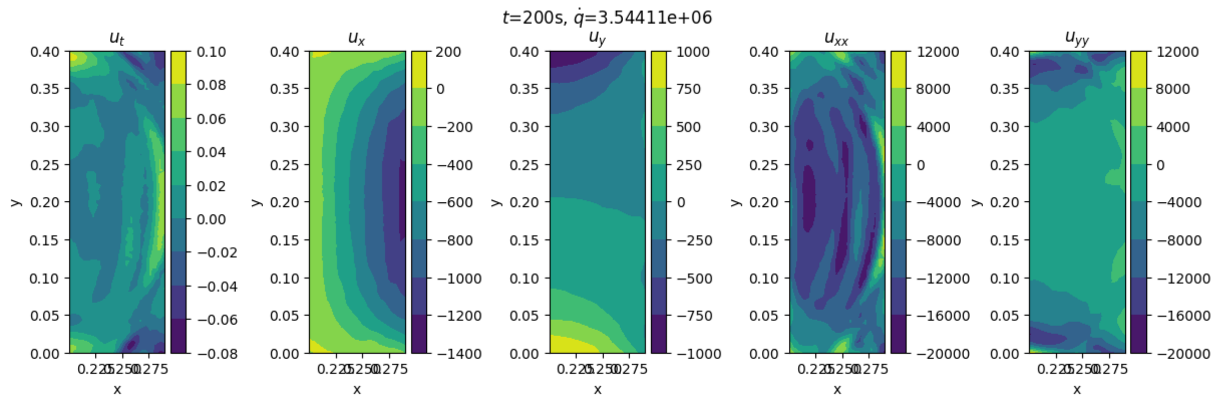


(b) PI-DeepONet at $\dot{q} = 3.544 \times 10^6$.

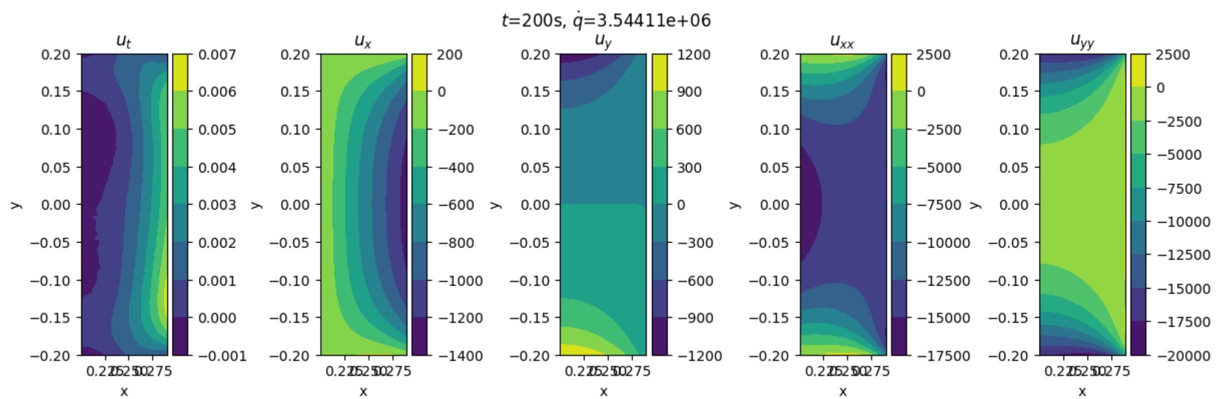


(c) HPI-DeepONet at $\dot{q} = 3.544 \times 10^6$.

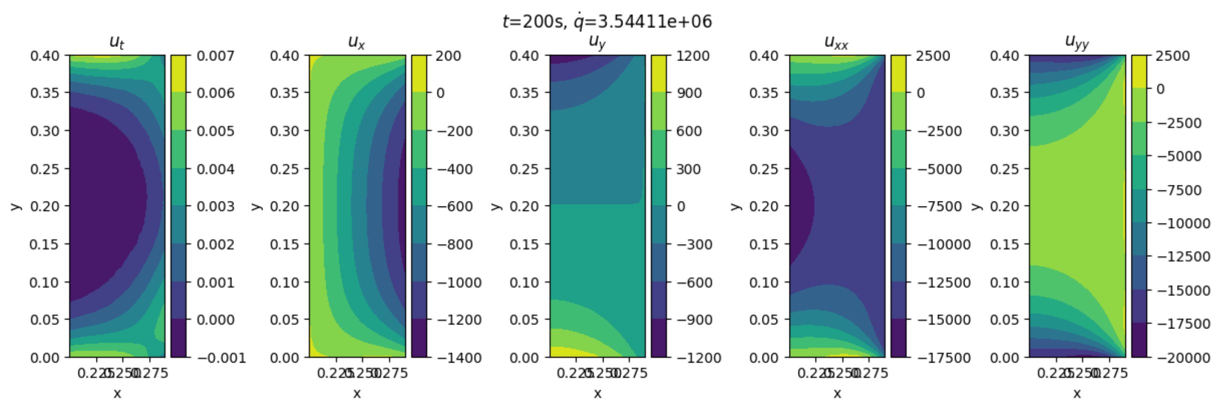
Figure 3.13 PDE Residual at $t = 0$ (left), $t = 100$ (middle), $t = 200$ (right) for the DI-DeepONet (top), PI-DeepONet (middle) and HPI-DeepONet (bottom).



(a) DI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 200$.



(b) PI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 200$.



(c) HPI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 200$.

Figure 3.14 PDE partial derivatives, u_t, u_x, u_y, u_{xx}, and u_{yy}, at $t = 200$ for the DI-DeepONet (top), PI-DeepONet (middle) and HPI-DeepONet (bottom).

at 3 different time-steps $t = [0, 100, 200]s$ for the DI-DeepONet, PI-DeepONet and HPI-DeepONet respectively, are provided in Appendix A.3. u_t, u_x, u_y, u_{xx} and u_{yy} at $t = [0, 100, 200]$ within the domain is also provided.

3.4.7 Summary of Findings

Important findings we observed for the DI-DeepONet, PI-DeepONet and HPI-DeepONet is given below.

- DI-DeepONet contains 100% training data for the ICs, BCs and within the domain.
- PI-DeepONet has only 53% training data for the ICs and Dirichlet BCs, with the remainder of the constraints obtained by physics for the heat PDE and Neumann BC.
- HPI-DeepONet is trained from 67% training data for the ICs, Dirichlet and Neumann BCs, with the remainder obtained by physics for the heat PDE.
- PI-DeepONet has a more complex learning task, when compared to the DI-DeepONet and HPI-DeepONet, as it has to satisfy the heat equation, as well as, the Neumann BC constraints without any data.
- PI-DeepONet is more consistent with the underlying physics as R_θ and N_θ are added as additional objectives, which act as a regularization mechanism biasing the network's output to satisfy the heat PDE and Neumann BC constraints.
- HPI-DeepONet is only biased to satisfy R_θ for the heat PDE constraint.
- The gradient terms, u_t, u_x, u_y, u_{xx} and u_{yy}, are clearly formed by the PI-DeepONet and HPI-DeepONet as opposed to the DI-DeepONet.
- Each variant reached reasonable accuracy in their predictions, with PI-DeepONet performing the best, followed by the HPI-DeepONet.

3.4.8 Conclusions

The DI-DeepONet, PI-DeepONet and HPI-DeepONet are able to learn the solution operator that maps $\dot{q}$ to the corresponding PDE solutions $u(x, y, t)$ for the heat equation (3.45). The PI-DeepONet performed best with 47% less training data (for the heat PDE residual and Neumann BC). This was closely followed by the HPI-DeepONet with 33% less training data (for the heat PDE residual). These constraints are defined using custom Pytorch modules and enforced by adding additional Nodes for each to the computational graph generated by Modulus. This ensures that the PI-DeepONet and HPI-DeepONet is more consistent with physics of the defined problem (3.4.1).

4 Conclusion

After discussing the DI-DeepONet and PI-DeepONet architectures (2.2.1), and their use for learning continuous operators for solving PDEs, we investigated their use for solving the heat conduction equation (3.1) for three different scenarios:

1. 1D transient heat equation (3.2) with parametric thermal diffusivity α, periodic ICs and Dirichlet BCs,
2. 2D steady-state heat equation (3.3) with parametric convective heat transfer coefficient h for the Robin BC, with additional, Dirichlet and Neumann BCs,
3. 2D axisymmetric transient heat equation (3.4) with a parametric source term $\dot{q}$, with ICs, Dirichlet and Neumann BCs.

We constructed DI-DeepONets and PI-DeepONets using NVIDIA's Modulus framework, and investigated their performance to learn the heat PDE operators, G, for the three different scenarios. We determined that each DeepONet can reasonably learn to solve the temperature field u, with the PI-DeepONet performing better in two of the three cases investigated. Recall that the DI-DeepONets require 100% training data to adequately learn G, where as the PI-DeepONets require training data only for some combination of the ICs and BCs, which in some cases (i.e. ICs and Dirichlet BCs), may be known a priori.

In scenario 1, we investigated the impact of the quality of the training data used for training the DI-DeepONet, by using two different data-sets generated from the analytical solution and the BTCS scheme. We found that the accuracy of the DI-DeepONet degrades using the numerical data, as compared to the one trained with the analytical data. We then investigated the impact of different AFs on the performance of the DI-DeepONet and PI-DeepONet, and determined a more severe impact on performance for the PI-DeepONet as compared to the DI-DeepONet.

In scenarios 2 and 3, we introduced a hybrid PI-DeepONet, HPI-DeepONet, where all BCs are defined by training data, where as the PI-DeepONet has only a portion of the BCs defined by data with the remainder supplied by physics. As a result, the amount of data required by each DeepONet drops from 100% for the DI-DeepONet, to a smaller portion for the HPI-DeepONet, and less for the PI-DeepONet. Each variant was able to adequately learn G, with the DI-DeepONet performing better in 2, while the PI-DeepONet performing better in 3.

Before closing, we must stress that not only is the PI-DeepONet trained to satisfy the heat equation with much less training data when compared to the DI-DeepONet, but is also constrained to satisfy the physics of the problem, thereby producing a solution operator $\hat{G}$ that is more consistent with the underlying physical constraints. Additionally, the PI-DeepONet training data can be assembled without first solving time-consuming FEA models or from field sensor measurements, however, these methods are required for the DI-DeepONet and HPI-DeepONet.

Despite the performance of DeepONets for solving the heat conduction problems, more work is needed to address some open questions that we noticed during our studies. The lambda weightings we applied to each objective of the DeepONet loss function were obtained manually by trial and error. The weightings were based on our observations on the localisation of the errors throughout the spatial and time domains using the validation data-set. To enhance the trainability of DeepONets, an in-depth study on the selection of these weights is required, particularly to determine if there is an adaptive method that

can be implemented for various problem domains. Additionally, the input functions used for the trunk networks were constant functions, that were easily represented using a single point, allowing for further studies using non-constant input functions.

Domain decomposition is a technique where the problem domain is divided into several subdomains that only interact along their shared boundaries, where some continuity conditions are enforced [10]. This technique has been extensively studied for heat conduction problems with composite material with PINNs [8, 29, 8, 10, 1], and is achieved by training separate sub-networks for each subdomain. A larger network is then used to combine each sub-network's total loss, with an additional interface loss, for the shared boundaries. The interface loss is then enforced, through physics, by ensuring that the temperatures and heat fluxes along the shared boundaries are similar. The use of composite material domains with added constraints for the interfaces has yet to be investigated with DeepONets, allowing for further investigations.

Bibliography

[1] Yan Gu Xiao Wang Benrong Zhang, Guozheng Wu and Fajie Wang. Multi-domain physics-informed neural network for solving forward and inverse problems of steady-state heat conduction in multilayer media. *Physics of Fluids*, 2022.

[2] Analytics by Vidhya. End-to-end introduction to evaluating regression models. https://www.analyticsvidhya.com/blog/2021/10/evaluation-metric-for-regression-models/#:~:text=In%20order%20to%20calculate%20Relative,mean%20as%20the%20predicted%20value. Accessed: 2023-02-01.

[3] Tianping Chen and Hong Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, 6(4):911–917, 1995.

[4] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). In Yoshua Bengio and Yann LeCun, editors, *ICLR (Poster)*, 2016.

[5] Nick Connor. What is heat equation – heat conduction equation – definition. https://www.thermal-engineering.org/what-is-heat-equation-heat-conduction-equation-definition/, May 2019. Accessed: 2023-01-01.

[6] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks*, 107:3–11, 2018. Special issue on deep reinforcement learning.

[7] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv:Learning*, 2016.

[8] Yuhao Huang. *Parallel Physics-Informed Neural Networks with Bidirectional Balance*. PhD thesis, Beijing Jiaotong University, Beijing, China, 2021.

[9] Turing Enterprises Inc. Role of artificial neural network in artificial intelligence. https://www.turing.com/kb/importance-of-artificial-neural-%networks-in-artificial-intelligence, Jul 2022. Accessed: 2023-01-01.

[10] Ameya Jagtap and George Em Karniadakis. *Extended physics-informed neural networks (XPINNs) : A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations*. PhD thesis, Brown University, Providence, RI, USA.

[11] Jeremy Jordan. Neural networks: training with backpropagation. https://www.jeremyjordan.me/neural-networks-training/, Jul 2017. Accessed: 2023-01-01.

[12] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 972–981, Red Hook, NY, USA, 2017. Curran Associates Inc.

[13] Seid Koric and Diab Abueidda. Data-driven and physics-informed deep learning operators for solution of heat conduction equation with parametric heat source. 12 2022.

[14] Ziyue Liu, Yixing Li, Jing Hu, Xinling Yu, Shinyu Shiau, Xin Ai, Zhiyu Zeng, and Zheng Zhang. Deepoheat: Operator learning-based ultra-fast thermal simulation in 3d-ic design. 02 2023.

[15] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, 2021.

[16] Guofei Pang Zhongqiang Zhang Lu Lu, Pengzhan Jin and George Em Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 2021.

[17] Pengzhan Jin Lu Lu and George Em Karniadakis. *DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators*. PhD thesis, Brown University, Providence, RI, USA, 2020.

[18] P. Perdikaris M. Raissi and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 2018.

[19] Inc Meta Platforms. Hydra. `https://hydra.cc/`, 2022. Accessed: 2022-10-15.

[20] Diganta Misra. *Mish: A Self Regularized Non-Monotonic Activation Function*. PhD thesis, KIIT, Bhubaneswar, India, 2020.

[21] Keith D. Humfeld Navid Zobeiry. *A Physics-Informed Machine Learning Approach for Solving Heat Transfer Equation in Advanced Manufacturing and Engineering Applications*. PhD thesis, University of Washington, Seattle, WA, USA.

[22] NVIDIA. Modulus user guide. `https://docs.nvidia.com/deeplearning/modulus/index.html`, 2022. Accessed: 2022-10-15.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[24] J. Chung X. Yue R. Gnanasambandam, B. Shen and Z. Kong. *Self-scalable Tanh (Stan): Faster Convergence and Better Generalization in Physics-informed Neural Networks*. PhD thesis, Virginia Tech, Blacksburg, US, 2022.

[25] M. W. Scroggs, I. A. Baratta, C. N. Richardson, and G. N. Wells. Basix: a runtime finite element basis evaluation library. *Journal of Open Source Software*, 7(73):3982, 2022.

[26] M. W. Scroggs, J. S. Dokken, C. N. Richardson, and G. N. Wells. Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes. *ACM Transactions on Mathematical Software*, 2022. To appear.

[27] Sifan Wang Paris Perdikaris Shengze Cai, Zhicheng Wang and George Em Karniadakis. Physics-informed neural networks (pinns) for heat transfer problems. *Journal of Heat Transfer*, 2021.

[28] Hanwen Wang Sifan Wang and Paris Perdikaris. *Learning the Solution Operator of Parametric Partial Differential Equations with Physics-Informed DeepONets*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 2021.

[29] Balaji Srinivasan Sreehari Manikkan. Transfer physics informed neural network: a new framework for distributed physics informed neural networks via parameter sharing. *Engineering with Computers*, 2022.

[30] Duke University. Lecture notes on pdes, part i: The heat equation and the eigenfunction method. https://services.math.duke.edu/~jtwong/math353-2018/lectures/Notes-PDEs1.pdf, 2018. Accessed: 2023-01-01.

[31] Huang Zhu Wang Tong Sheng, Wang Zhi Heng and Xi Guang. Multi-domain physics-informed neural network for solving heat conduction and conjugate natural convection with discontinuity of temperature gradient on interface. *Science China Technological Sciences*, 2022.

[32] Wolfram. Heat transfer model verification tests. https://reference.wolfram.com/language/PDEModels/tutorial/HeatTransfer/HeatTransferVerificationTests.html. Accessed: 2022-11-01.

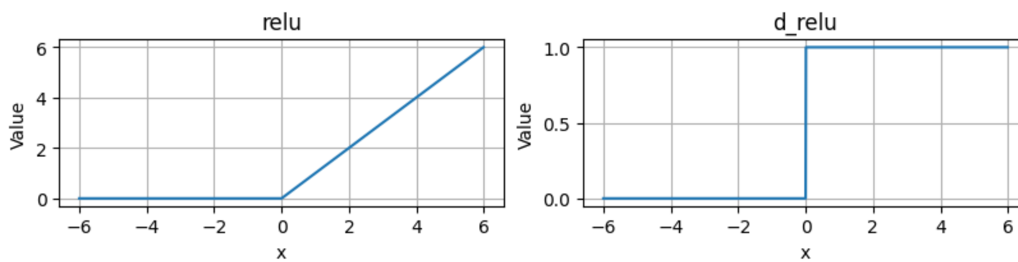
[33] Weiguo Wang Zhili He, Futao Ni and Jian Zhang. A physics-informed deep learning method for solving direct and inverse heat conduction problems of materials. *Materials Today Communications*, 2021.

A Appendix

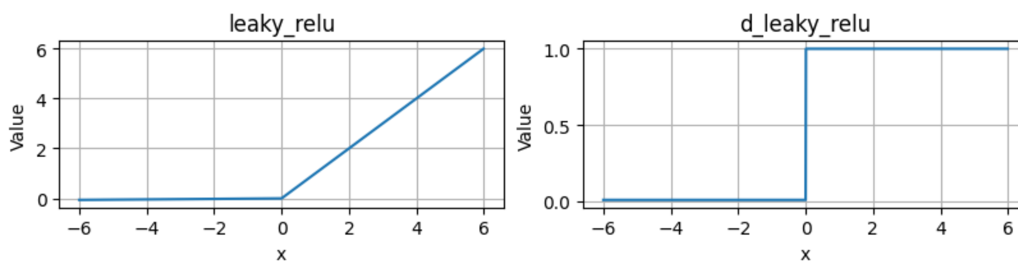
A.1 Activation Functions (AFs)

A.1.1 Activation Function Graphs

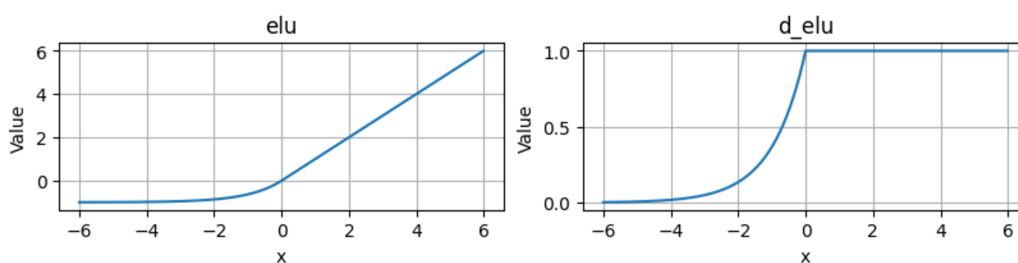
In this section, we provide graphs for the AFs and their first derivatives introduced in section 2.3.3 in figures A.1, A.2, and A.3.



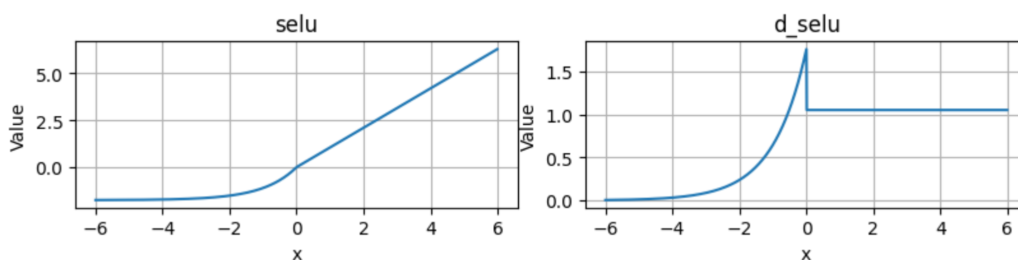
(a) ReLU and its first derivative.



(b) Leaky ReLU and its first derivative.

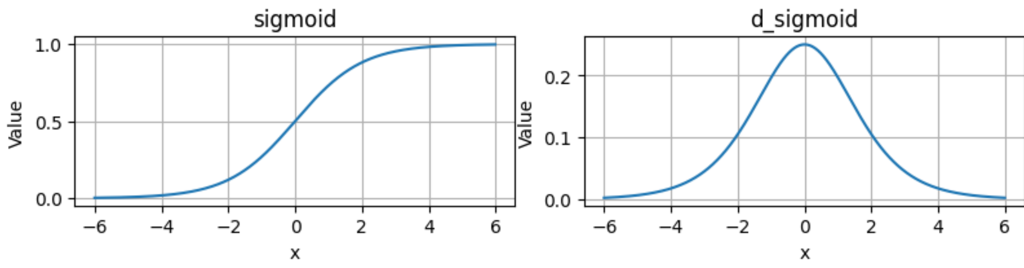


(c) ELU and its first derivative.

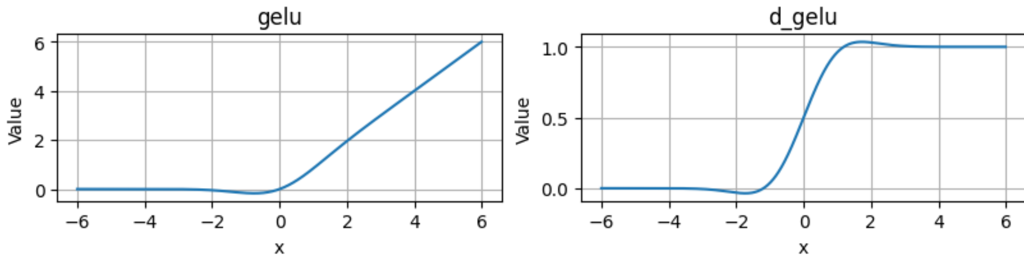


(d) SELU and its first derivative.

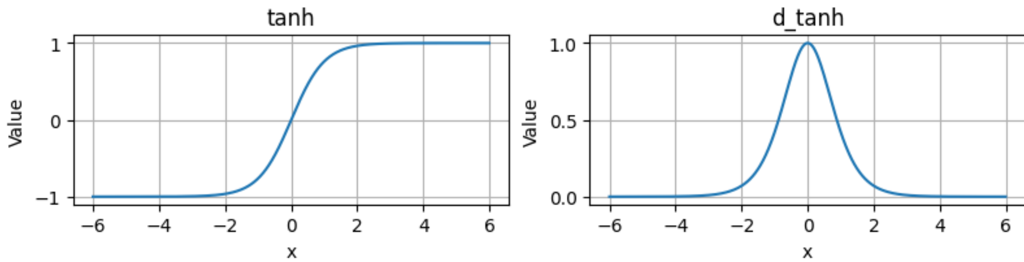
Figure A.1 Graphs of ReLU, Leaky ReLU, ELU, and SELU AFs with their first derivatives.



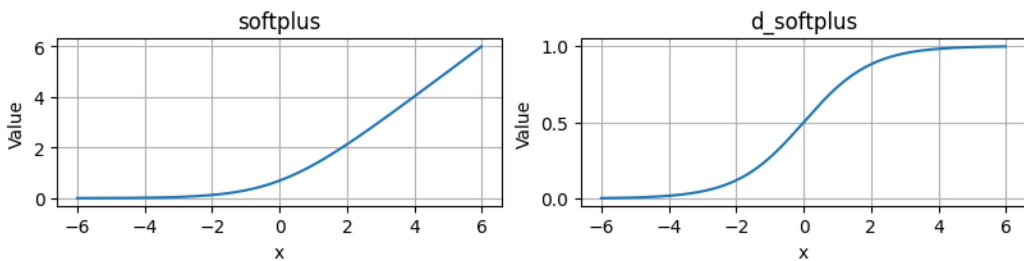
(a) Sigmoid and its first derivative.



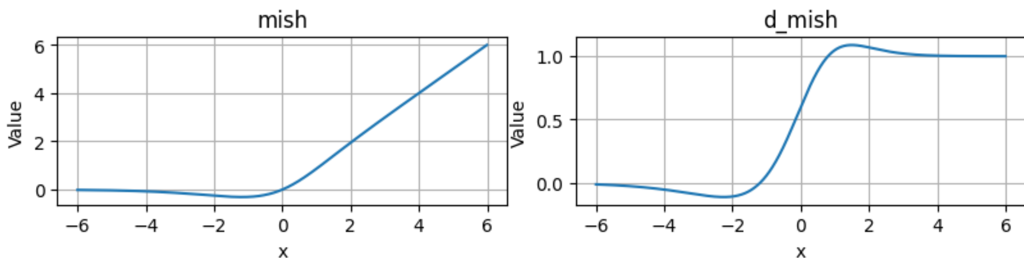
(b) GELU and its first derivative.



(c) Tanh and its first derivative.

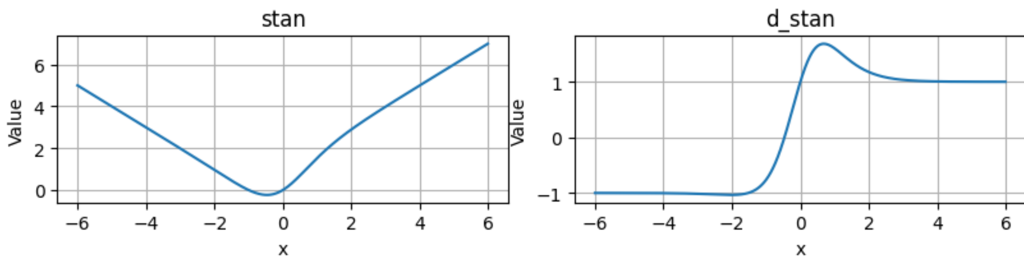


(d) Softplus and its first derivative.

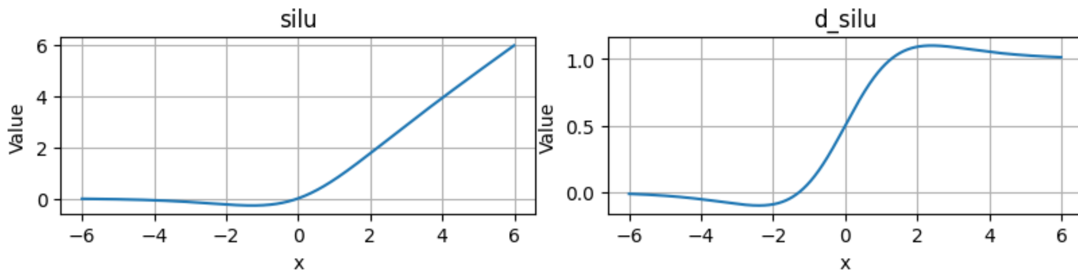


(e) Mish and its first derivative.

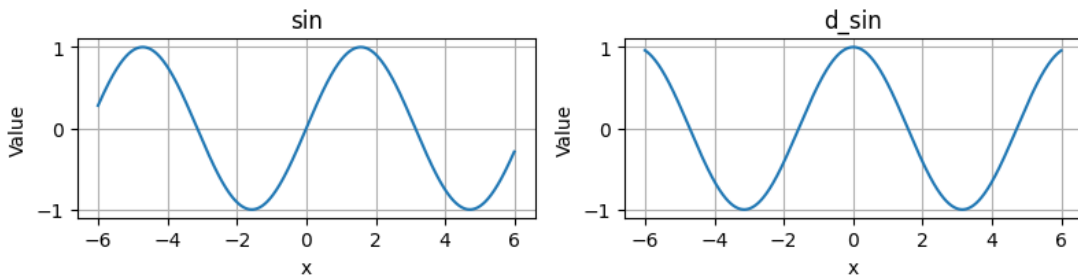
Figure A.2 Graphs of Sigmoid, GELU, Tanh, Softplus, and Mish AFs with their first derivatives.



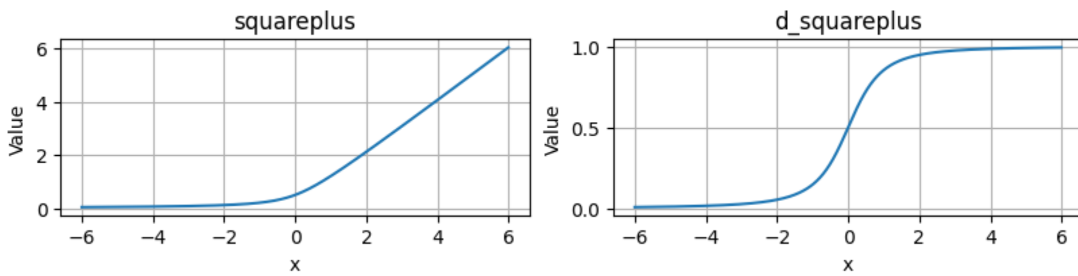
(a) Stan and its first derivative.



(b) SiLU and its first derivative.



(c) Sin and its first derivative.

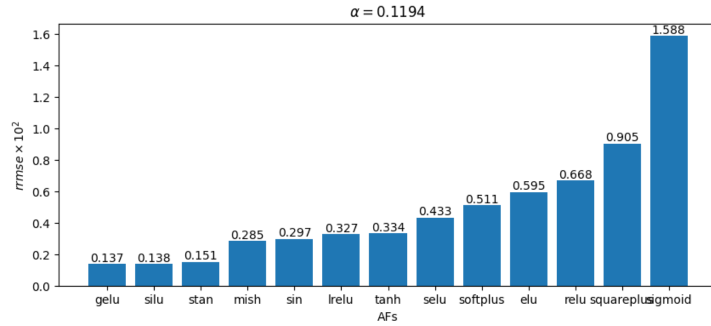


(d) Squareplus and its first derivative.

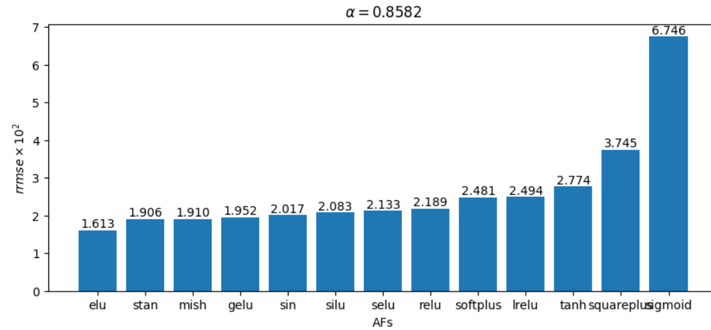
Figure A.3 Illustrations of Stan, SiLU, Sin, and Squareplus AFs with their first derivatives.

A.1.2 Activation Function Results

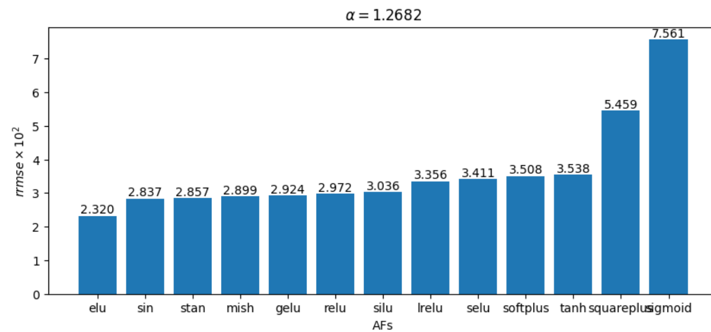
The RRMSE (3.11) and temperature differences, u_{diff}, for the DI-DeepONet and PI-DeepONet for 4 different α functions used for the 1D transient heat conduction problem (section 3.2) is provided in figures A.4, A.5, A.6, and A.7.



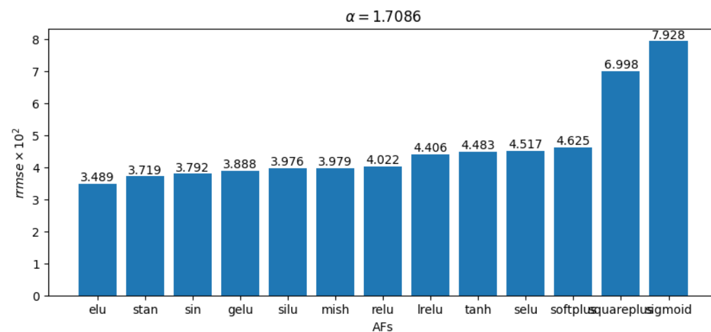
(a) RRMSE for different AFs at $\alpha = 0.1194$.



(b) RRMSE for different AFs at $\alpha = 0.8582$.

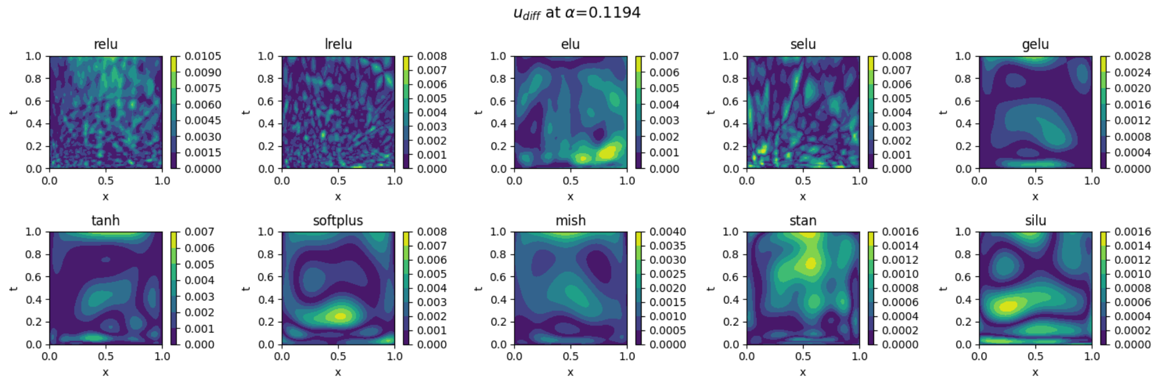


(c) RRMSE for different AFs at $\alpha = 1.2682$.

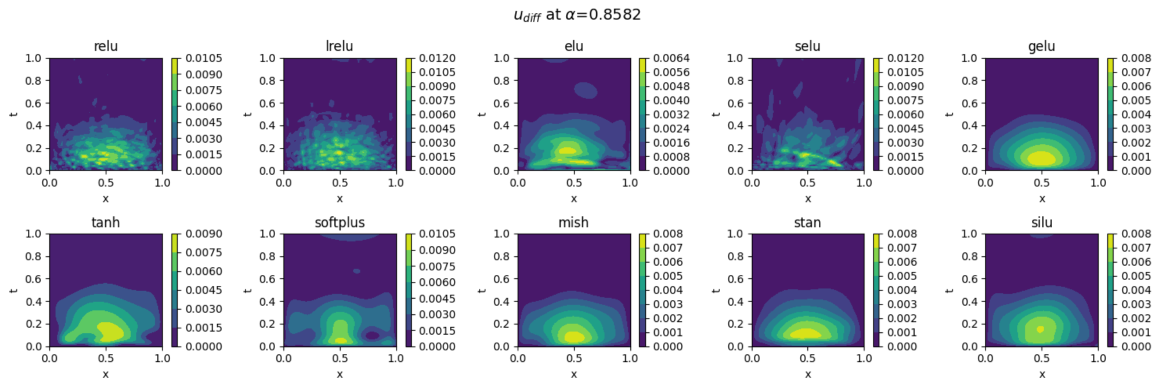


(d) RRMSE for different AFs at $\alpha = 1.7086$.

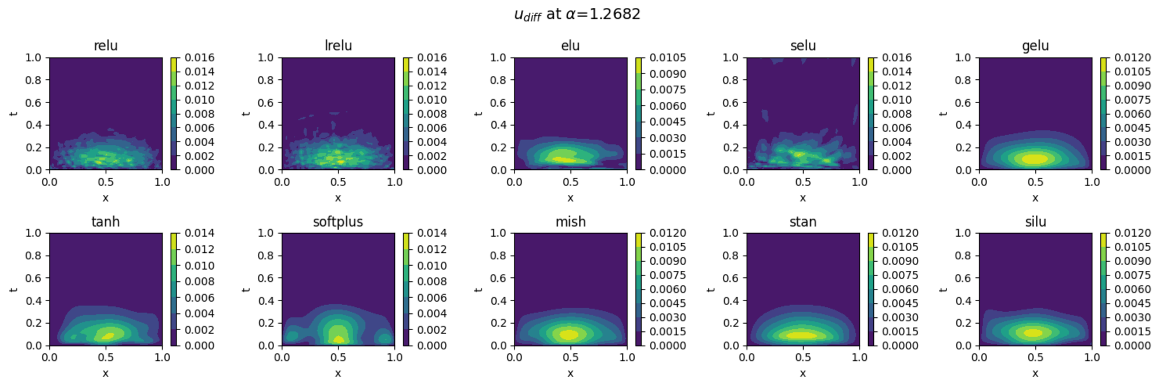
Figure A.4 DI-DeepONet: RRMSE Bar Graphs for different AFs at $\alpha = [0.1194, 0.8582, 1.2682, 1.7086]$ respectively.



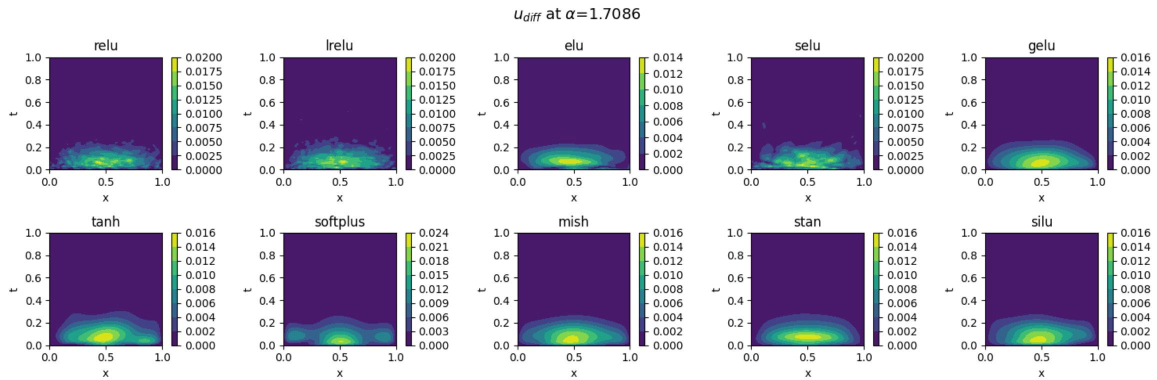
(a) u_{diff} for different AFs at $\alpha = 0.1194$.



(b) u_{diff} for different AFs at $\alpha = 0.8582$.

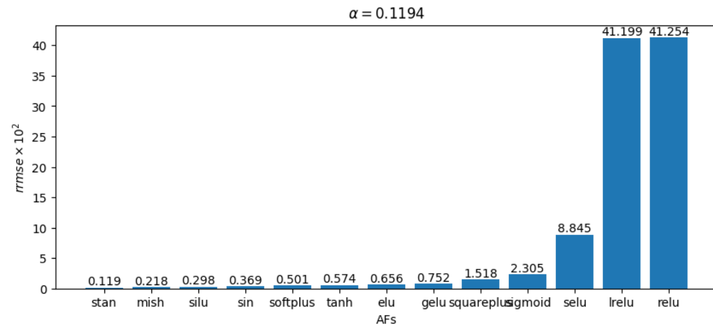


(c) u_{diff} for different AFs at $\alpha = 1.2682$.

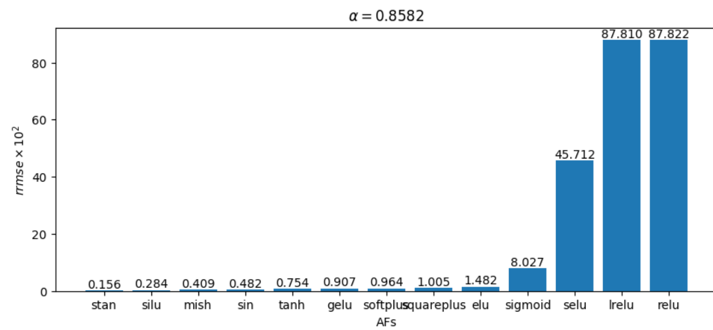


(d) u_{diff} for different AFs at $\alpha = 1.7086$.

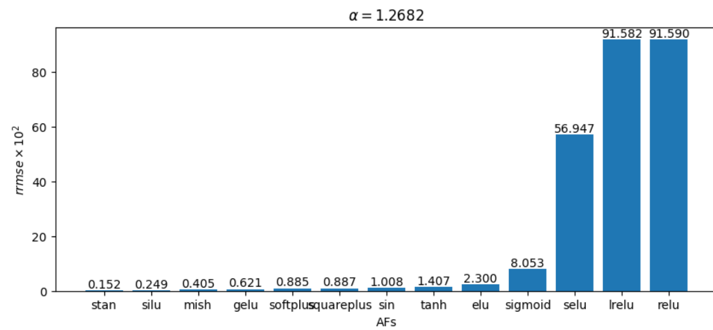
Figure A.5 DI-DeepONet: u_{diff} for different AFs at $\alpha = [0.1194, 0.8582, 1.2682, 1.7086]$ respectively.



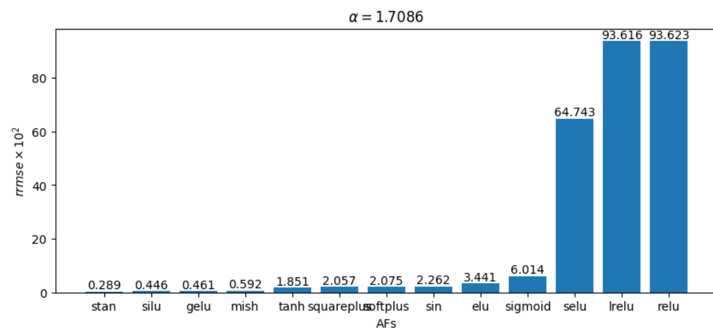
(a) RRMSE for different AFs at $\alpha = 0.1194$.



(b) RRMSE for different AFs at $\alpha = 0.8582$.

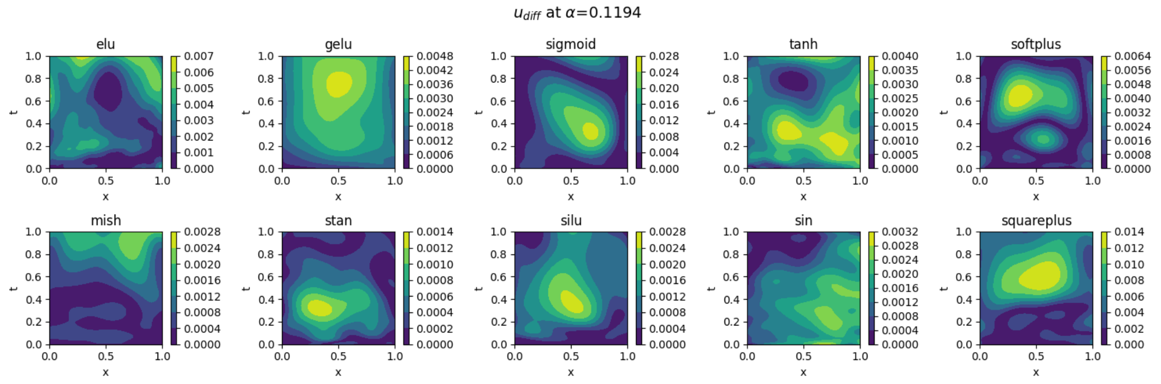


(c) RRMSE for different AFs at $\alpha = 1.2682$.

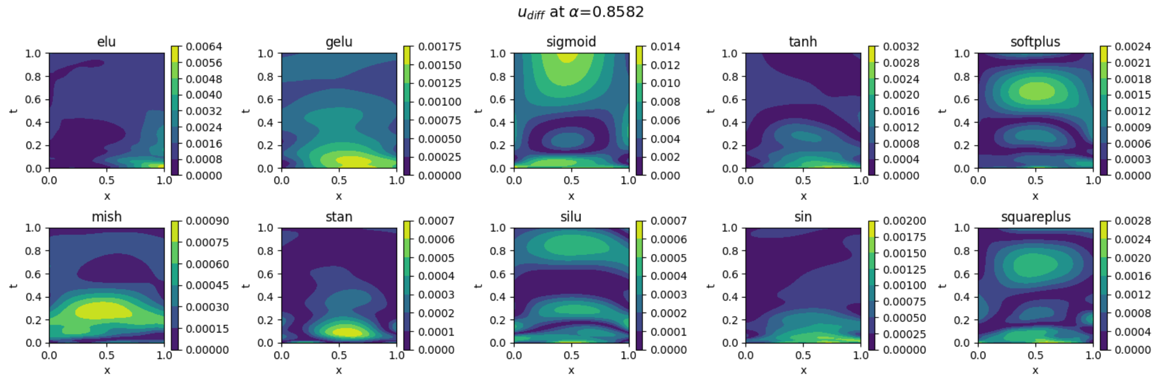


(d) RRMSE for different AFs at $\alpha = 1.7086$.

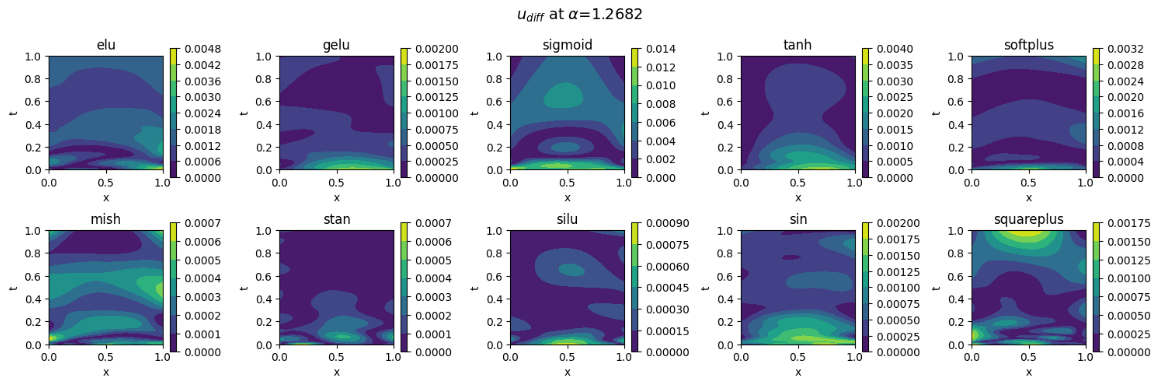
Figure A.6 PI-DeepONet: RRMSE Bar Graphs for different AFs at $\alpha = [0.1194, 0.8582, 1.2682, 1.7086]$ respectively.



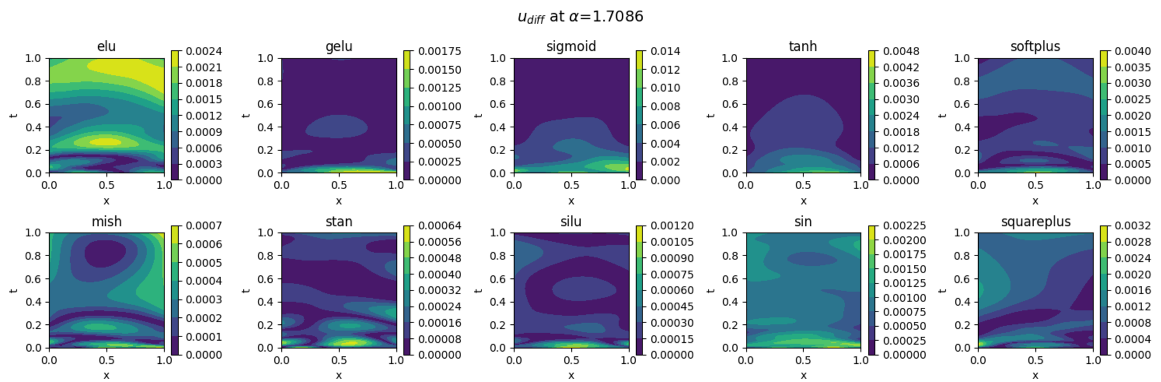
(a) u_{diff} for different AFs at $\alpha = 0.1194$.



(b) u_{diff} for different AFs at $\alpha = 0.8582$.



(c) u_{diff} for different AFs at $\alpha = 1.2682$.

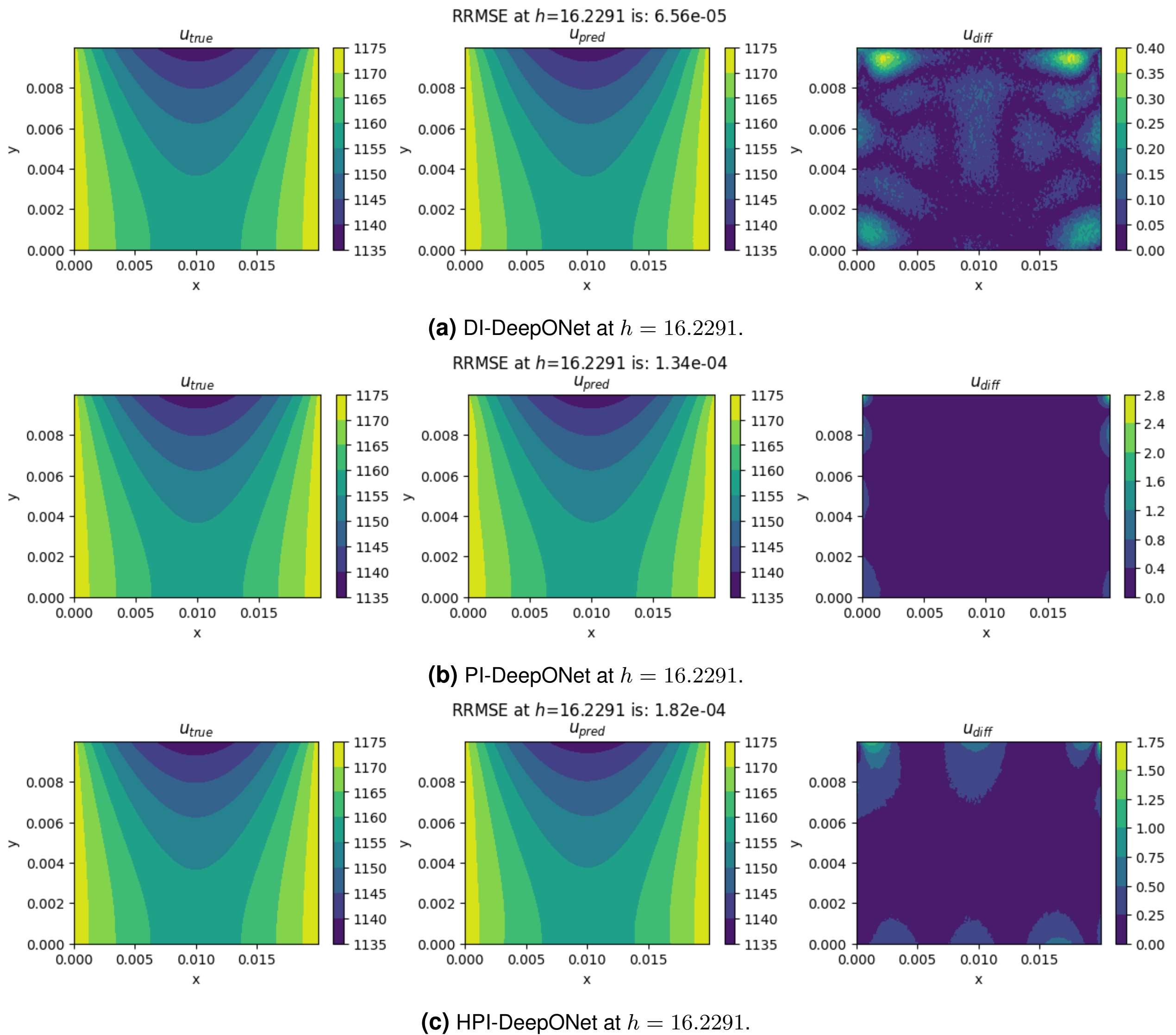


(d) u_{diff} for different AFs at $\alpha = 1.7086$.

Figure A.7 PI-DeepONet: u_{diff} for different AFs at $\alpha = [0.1194, 0.8582, 1.2682, 1.7086]$ respectively.

A.2 2D Steady-State Heat Equation Results

The temperature field $\tilde{u}$ predicted by the DI-DeepONet, PI-DeepONet and HPI-DeepONet are compared to the true u, given by the FEM solution using Modulus' Validators. The results are shown for 2 convective heat transfer coefficients h at the extremes of the range given in table 3.6 in figures A.8 and A.9.

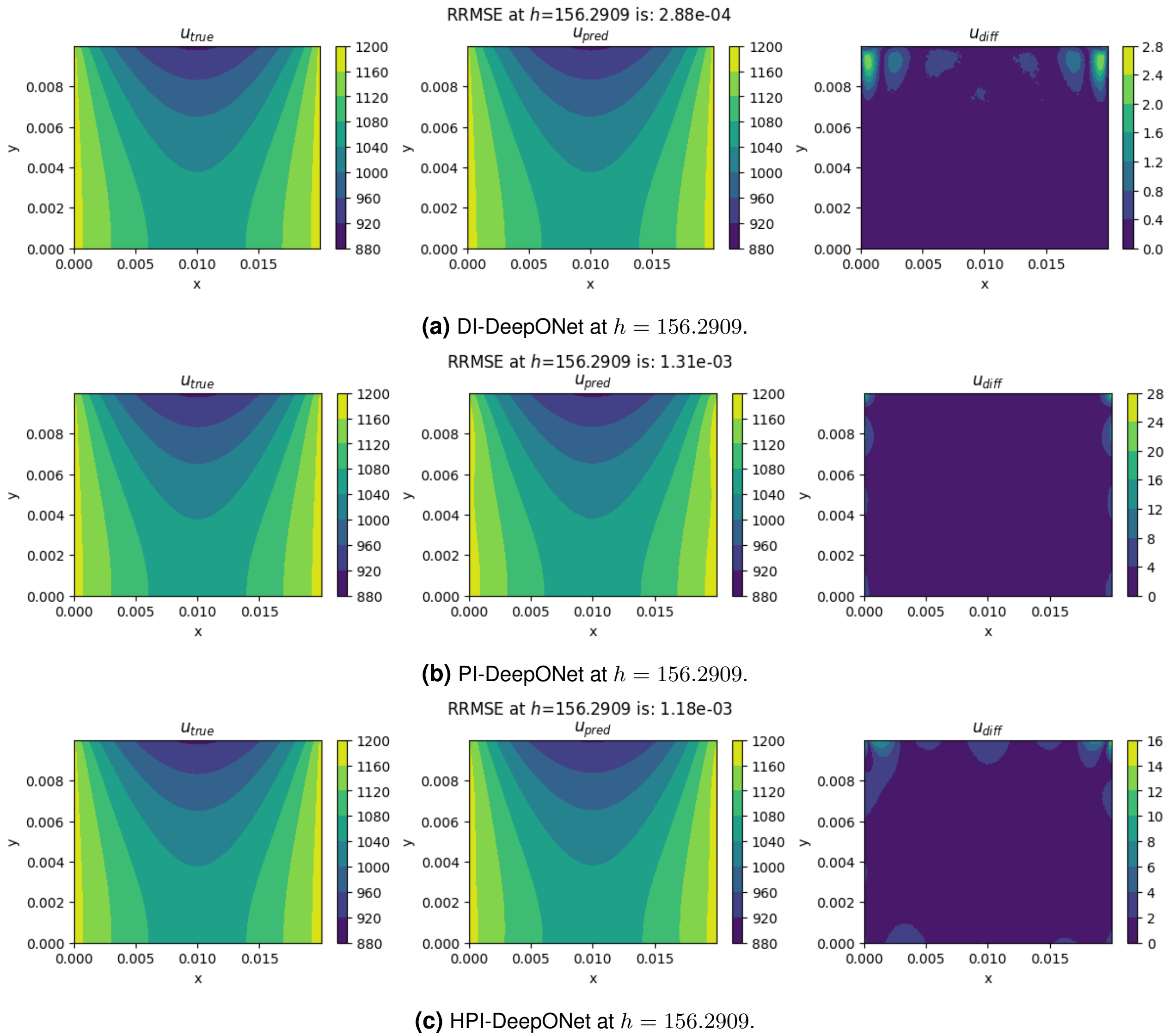


(a) DI-DeepONet at $h = 16.2291$.

(b) PI-DeepONet at $h = 16.2291$.

(c) HPI-DeepONet at $h = 16.2291$.

Figure A.8 Temperature Field Predicted by DI-DeepONet (top), PI-DeepONet (middle) and HPI-DeepONet (bottom), with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).



(a) DI-DeepONet at $h = 156.2909$.

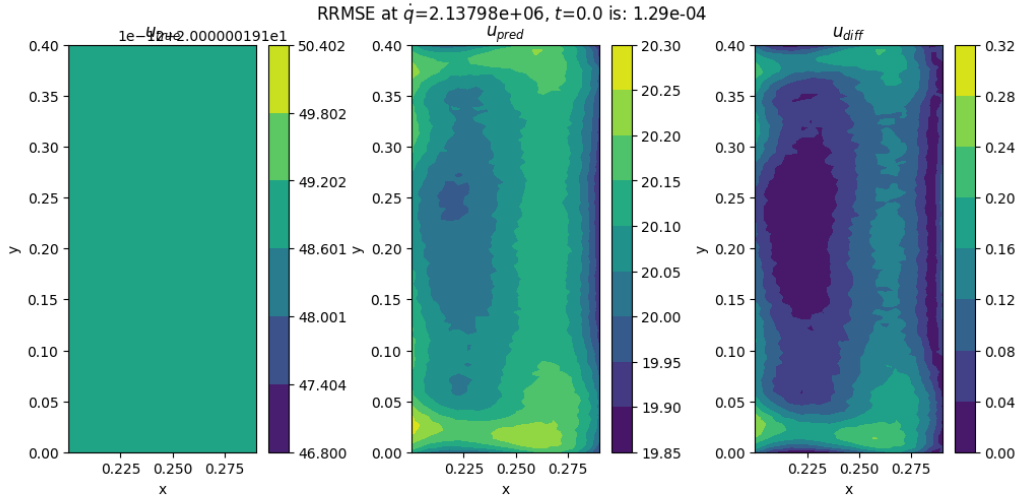
(b) PI-DeepONet at $h = 156.2909$.

(c) HPI-DeepONet at $h = 156.2909$.

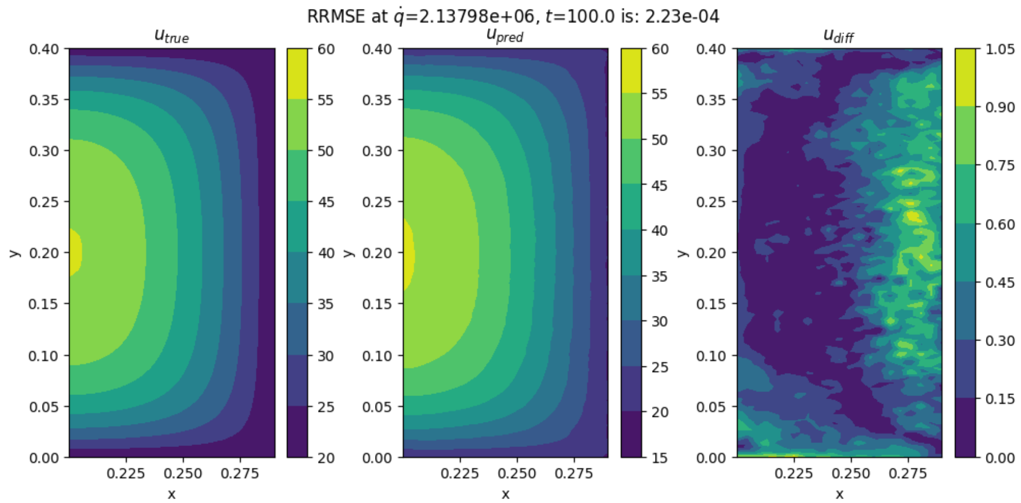
Figure A.9 Temperature Field Predicted by DI-DeepONet (top), PI-DeepONet (middle) and HPI-DeepONet (bottom), with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).

A.3 Heat Equation 2D Transient Axisymmetric Results

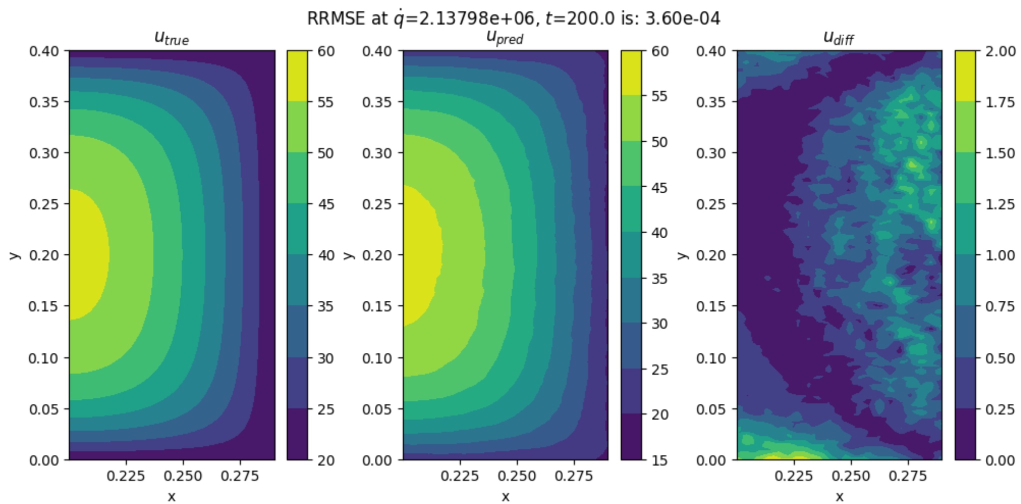
The temperature field $\tilde{u}$ predicted by the DI-DeepONet, PI-DeepONet and HPI-DeepONet are compared to the true u for 2 sources $\dot{q}$ at the extremes of the range given in table 3.12, at time-steps $t = [0, 100, 200]s$ in figures A.10 to A.15. u_t, u_x, u_y, u_{xx} and u_{yy} at $t = [0, 100, 200]$ within the domain are also shown in figures A.16 to A.16.



(a) $\dot{q} = 2.138 \times 10^6$ and $t = 0.$

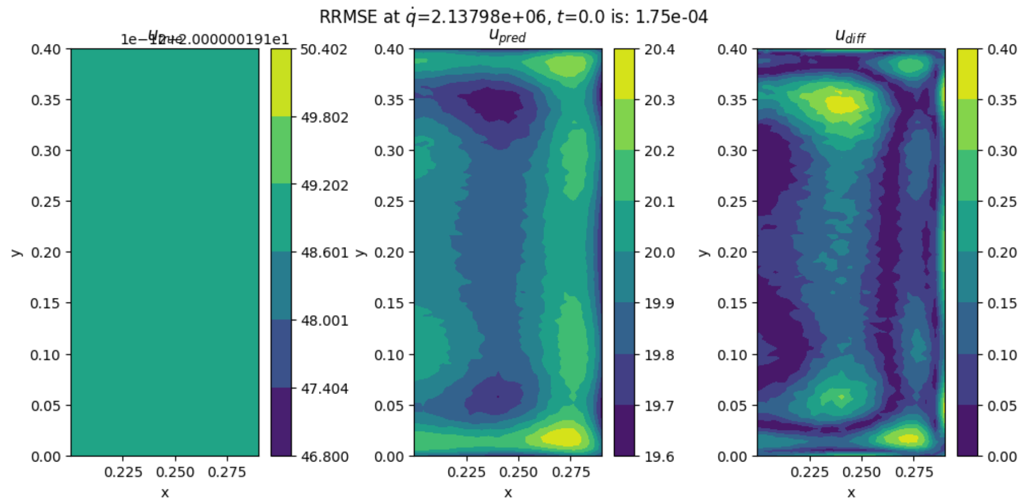


(b) $\dot{q} = 2.138 \times 10^6$ and $t = 100.$

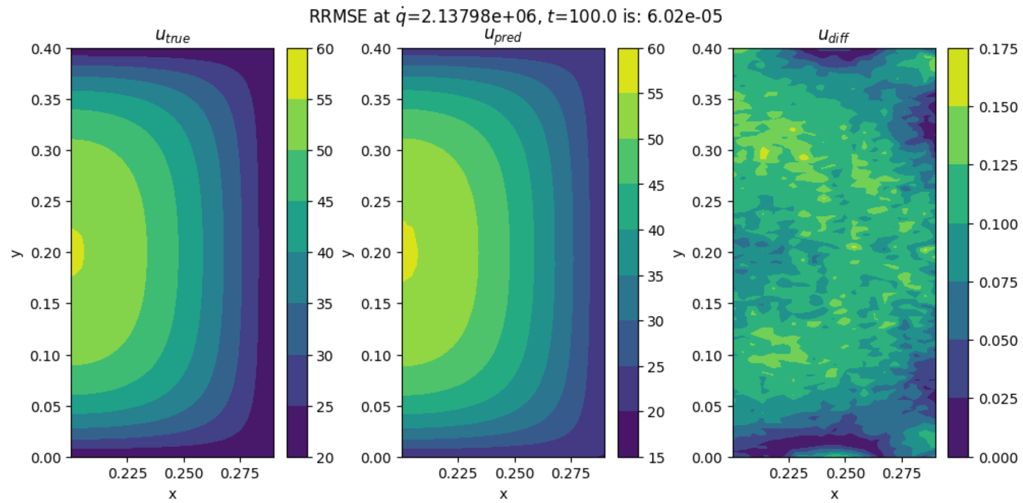


(c) $\dot{q} = 2.138 \times 10^6$ and $t = 200.$

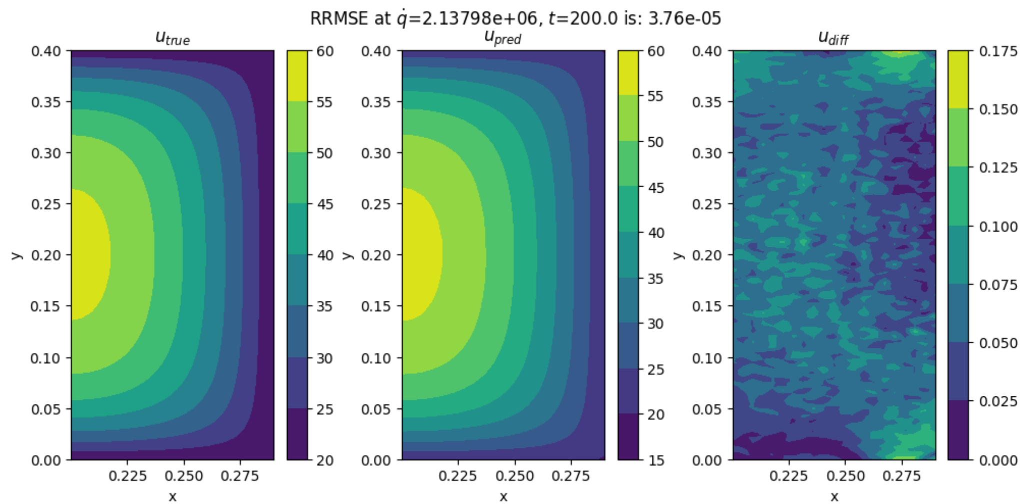
Figure A.10 Temperature Field Predicted by DI-DeepONet at $t = [0, 100, 200]s$, with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).



(a) $\dot{q} = 2.138 \times 10^6$ and $t = 0$.

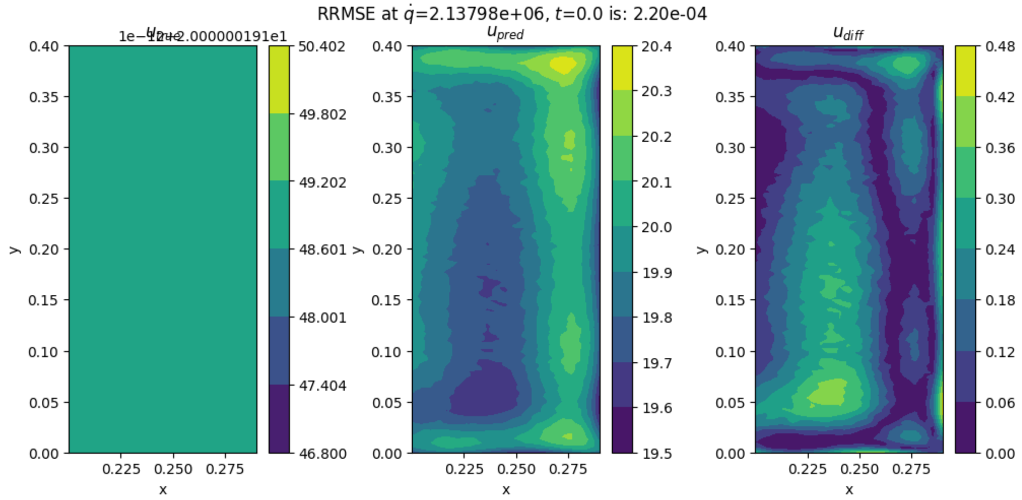


(b) $\dot{q} = 2.138 \times 10^6$ and $t = 100$.

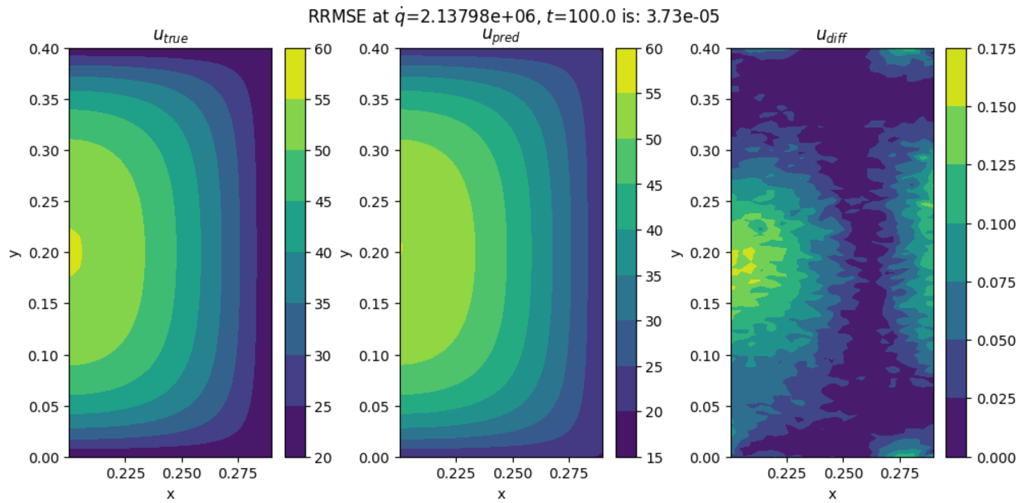


(c) $\dot{q} = 2.138 \times 10^6$ and $t = 200$.

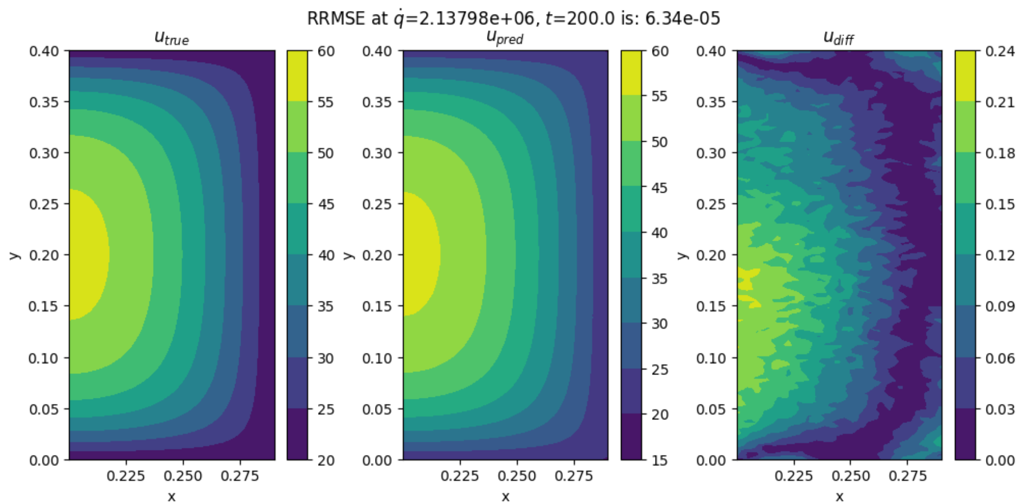
Figure A.11 Temperature Field Predicted by PI-DeepONet at $t = [0, 100, 200]s$, with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).



(a) $\dot{q} = 2.138 \times 10^6$ and $t = 0$.

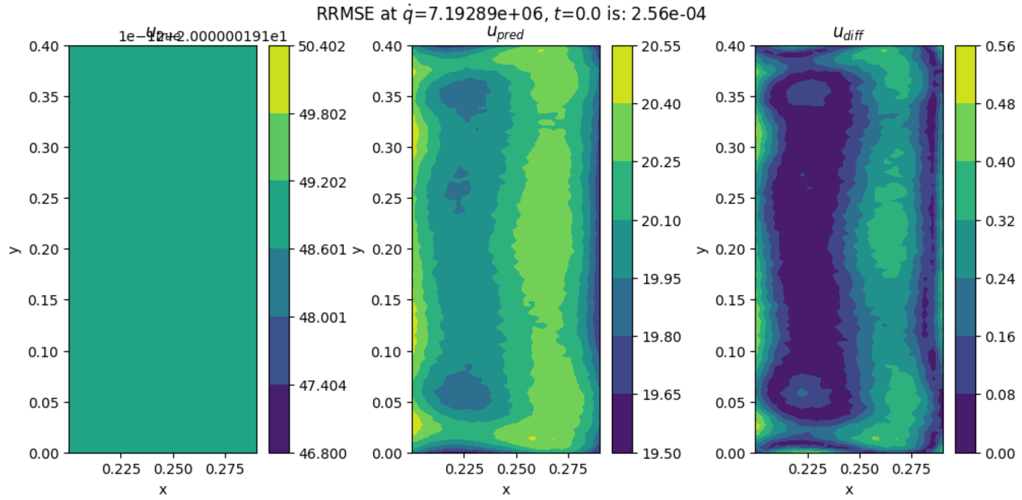


(b) $\dot{q} = 2.138 \times 10^6$ and $t = 100$.

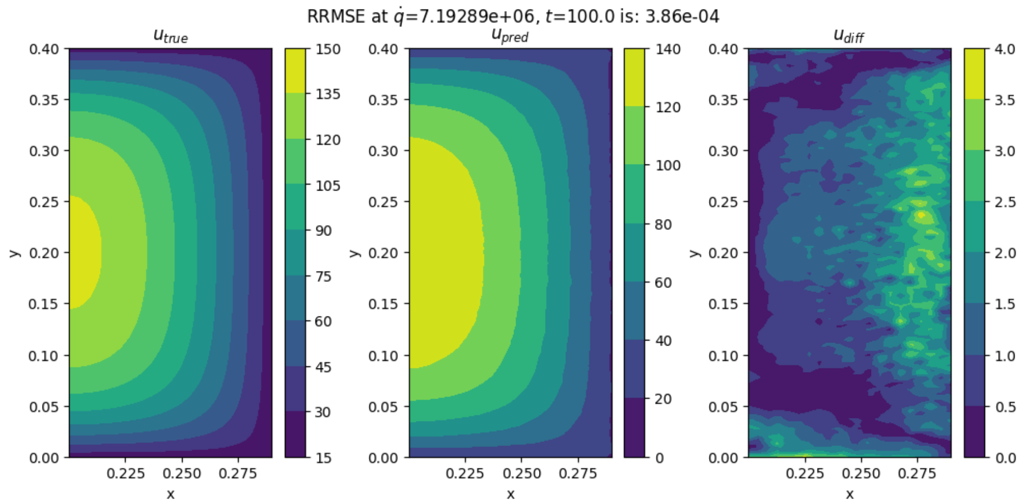


(c) $\dot{q} = 2.138 \times 10^6$ and $t = 200$.

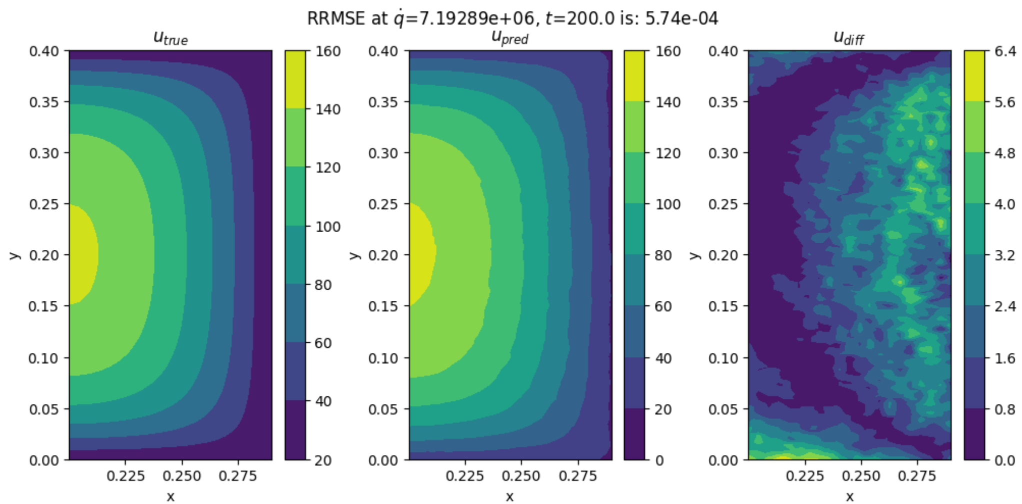
Figure A.12 Temperature Field Predicted by HPI-DeepONet at $t = [0, 100, 200]s$, with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).



(a) $\dot{q} = 7.193 \times 10^6$ and $t = 0$.

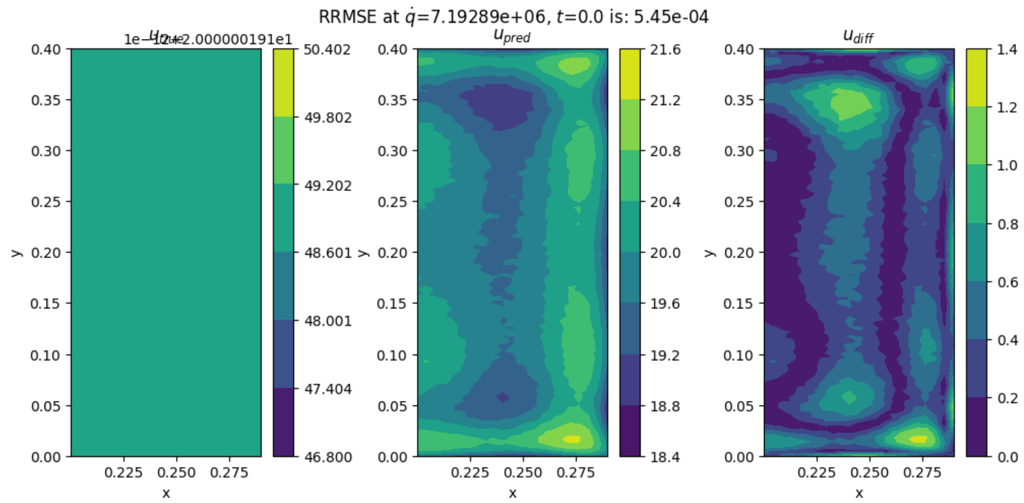


(b) $\dot{q} = 7.193 \times 10^6$ and $t = 100$.

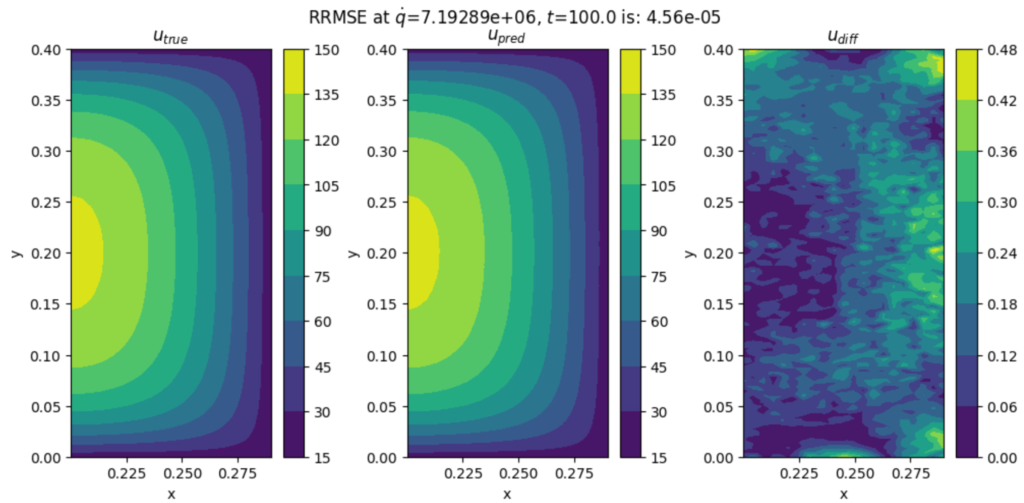


(c) $\dot{q} = 7.193 \times 10^6$ and $t = 200$.

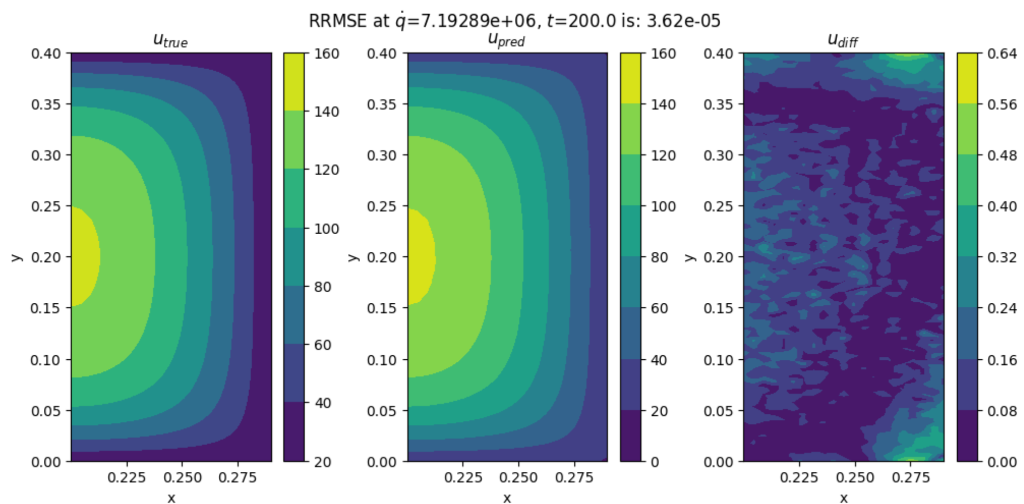
Figure A.13 Temperature Field Predicted by DI-DeepONet at $t = [0, 100, 200]s$, with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).



(a) $\dot{q} = 7.193 \times 10^6$ and $t = 0$.

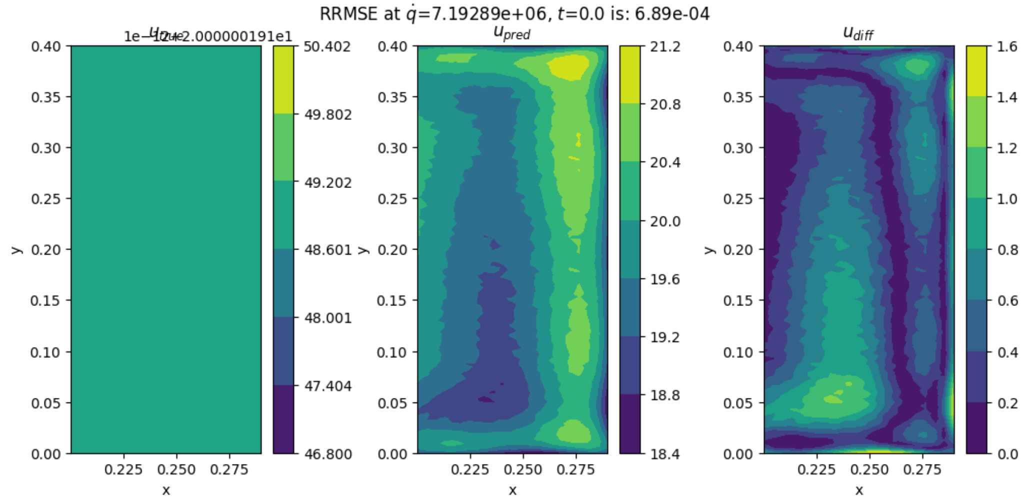


(b) $\dot{q} = 7.193 \times 10^6$ and $t = 100$.

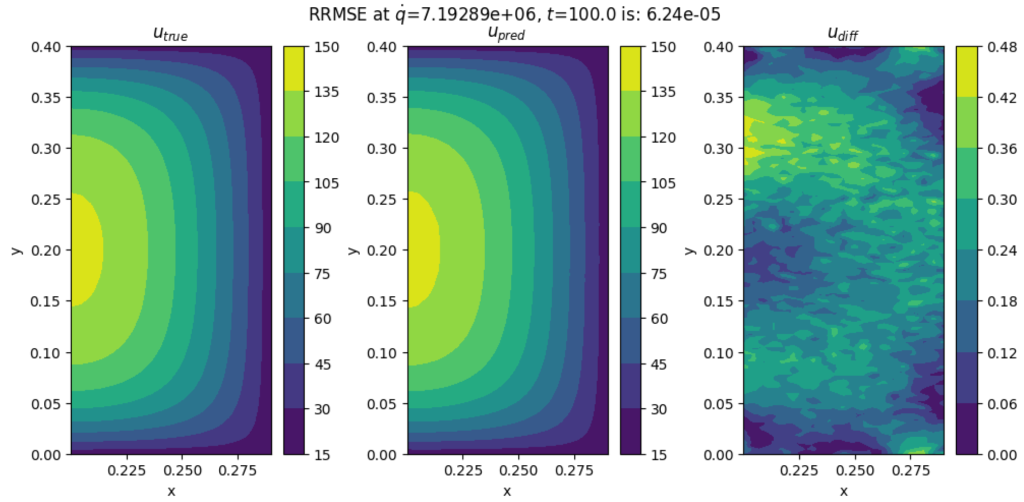


(c) $\dot{q} = 7.193 \times 10^6$ and $t = 200$.

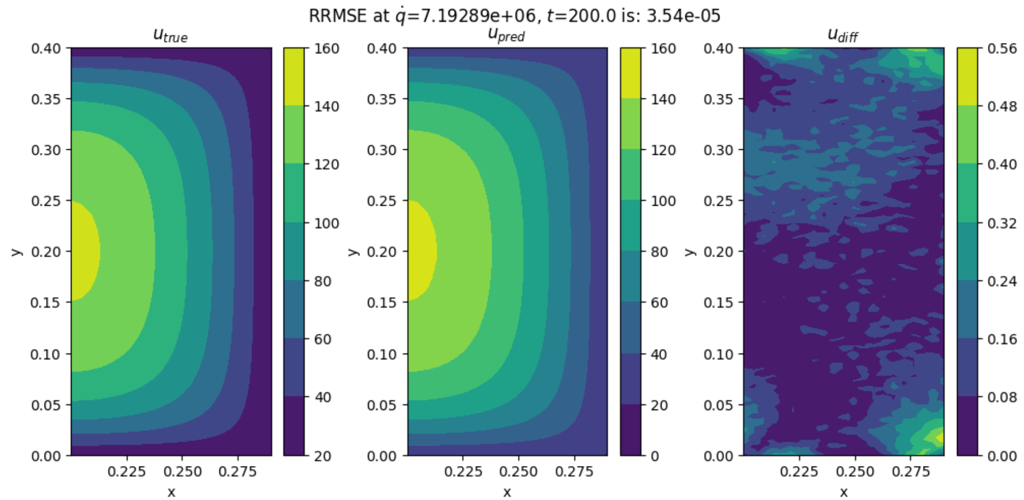
Figure A.14 Temperature Field Predicted by PI-DeepONet at $t = [0, 100, 200]s$, with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).



(a) $\dot{q} = 7.193 \times 10^6$ and $t = 0$.

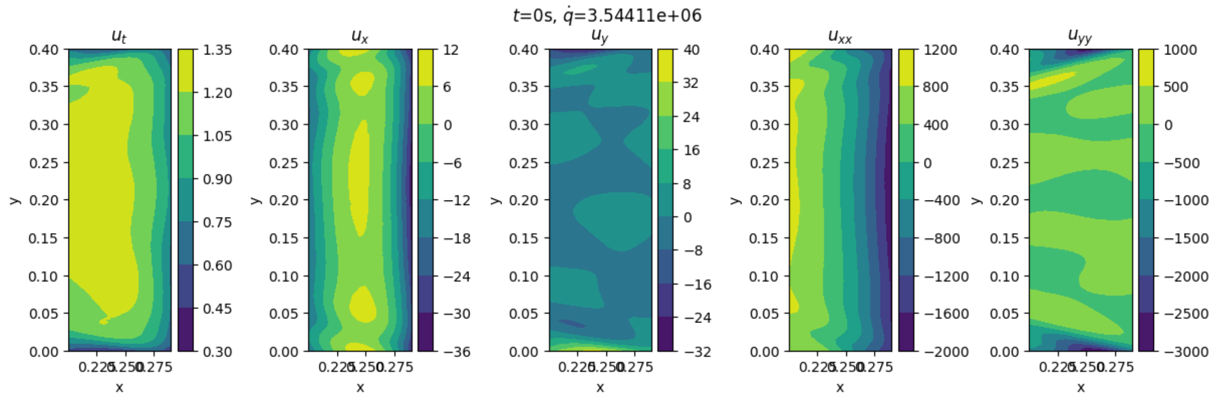


(b) $\dot{q} = 7.193 \times 10^6$ and $t = 100$.

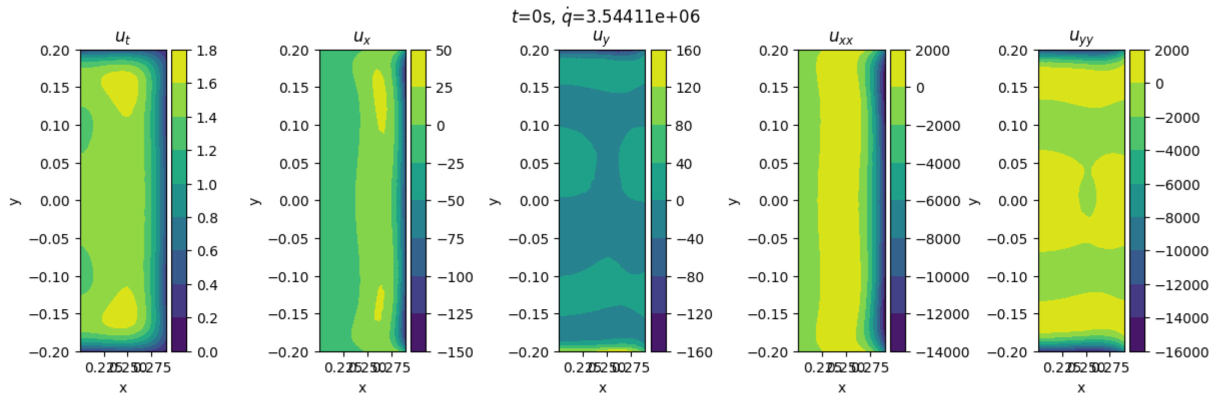


(c) $\dot{q} = 7.193 \times 10^6$ and $t = 200$.

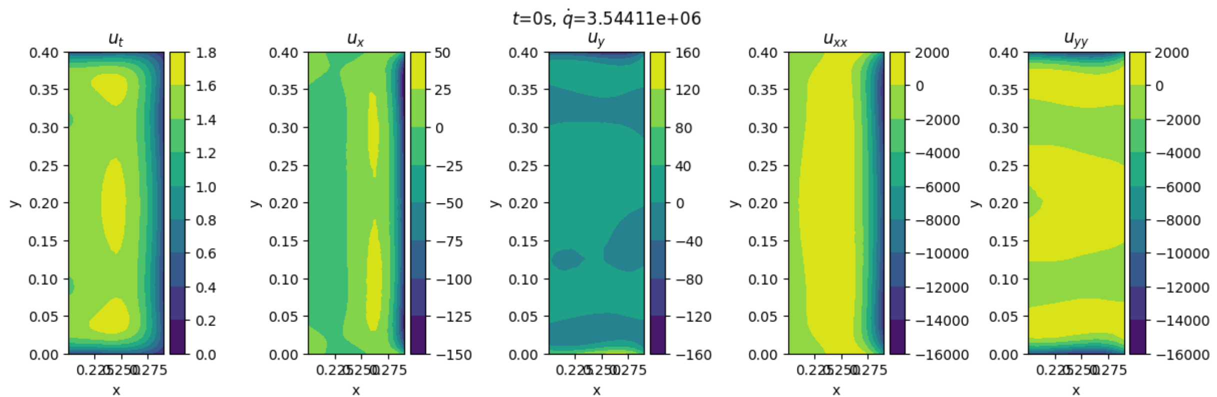
Figure A.15 Temperature Field Predicted by HPI-DeepONet at $t = [0, 100, 200]s$, with True Temperature Field u_{true} (left), Predicted Temperature Field u_{pred} (center), and Absolute Difference in Temperature u_{diff} (right).



(a) DI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 0$.

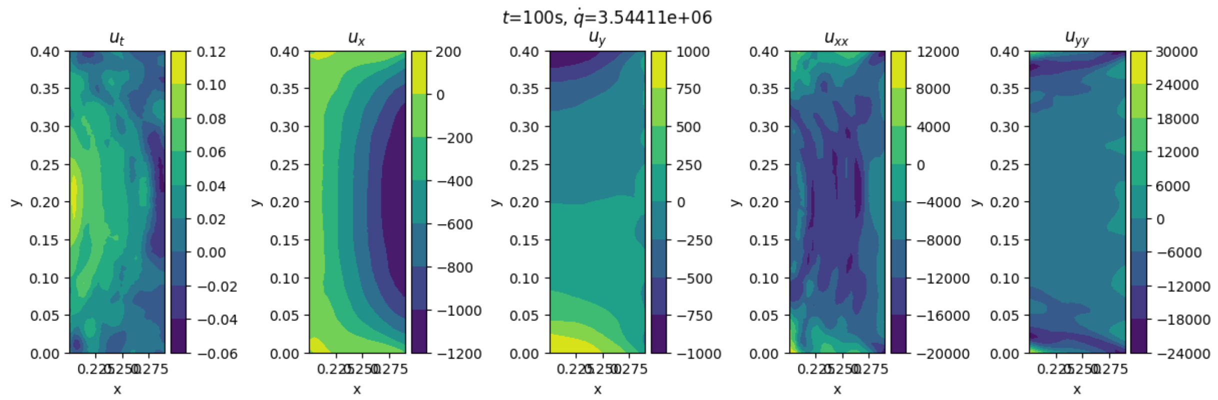


(b) PI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 0$.

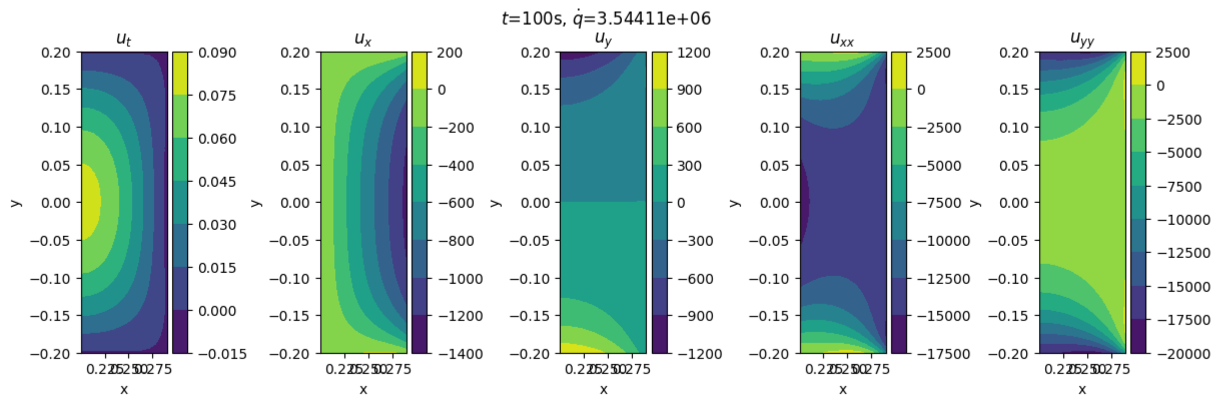


(c) HPI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 0$.

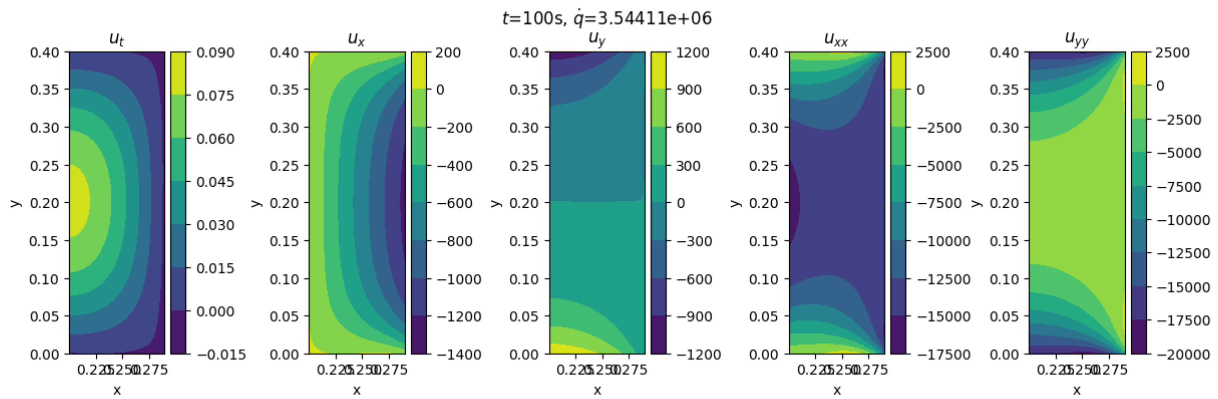
Figure A.16 PDE partial derivatives at $t = 0$, u_t, u_x, u_y, u_{xx} and u_{yy} for the DI-DeepONet (top), PI-DeepONet (middle) and HPI-DeepONet (bottom).



(a) DI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 100$.

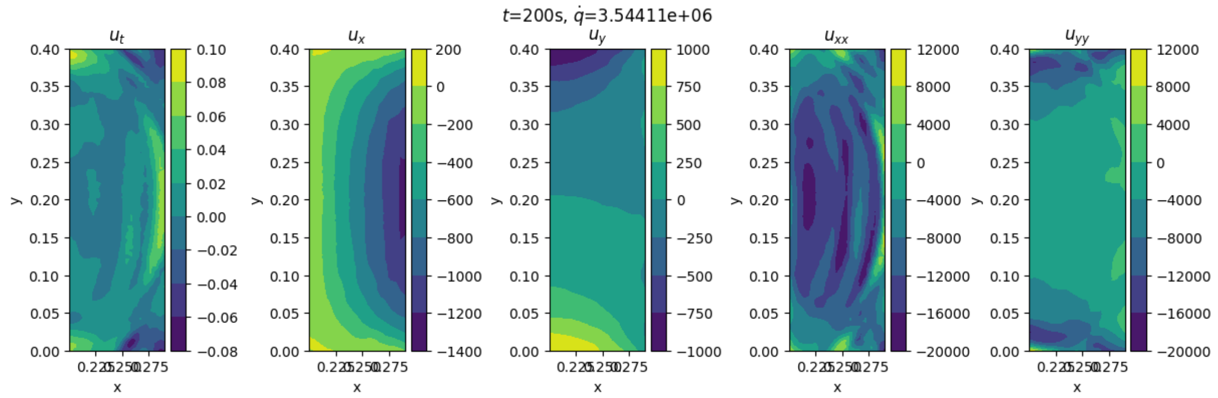


(b) PI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 100$.

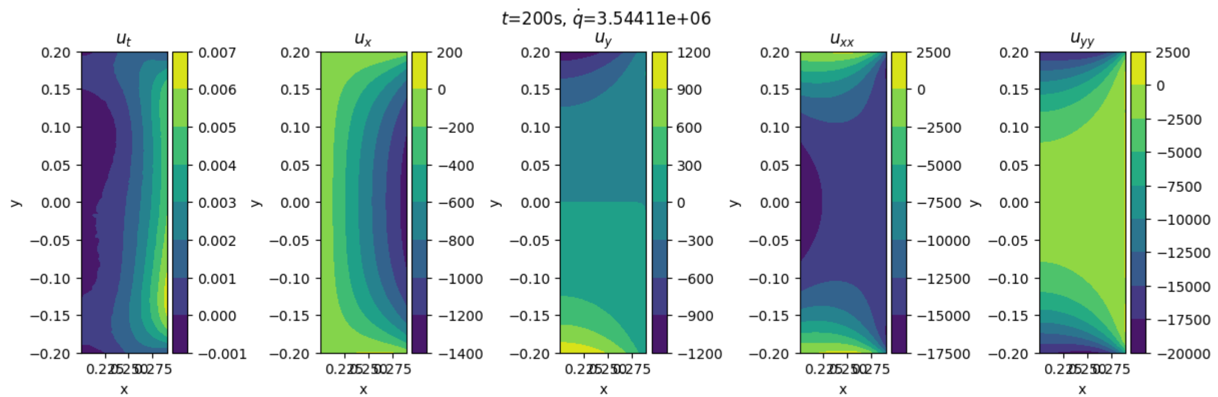


(c) HPI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 100$.

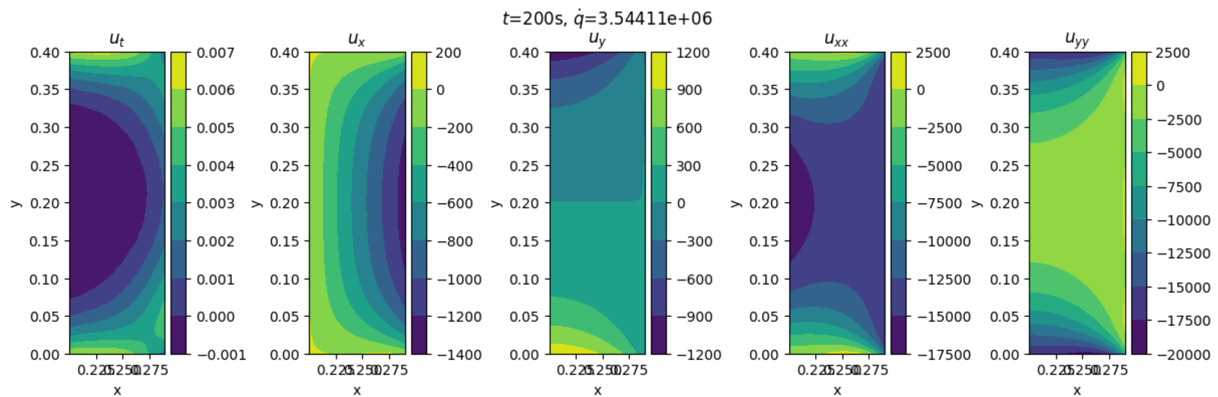
Figure A.17 PDE partial derivatives at $t = 100$, u_t, u_x, u_y, u_{xx} and u_{yy} for the DI-DeepONet (top), PI-DeepONet (middle) and HPI-DeepONet (bottom).



(a) DI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 200$.



(b) PI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 200$.



(c) HPI-DeepONet at $\dot{q} = 3.544 \times 10^6$ and $t = 200$.

Figure A.18 PDE partial derivatives, u_t, u_x, u_y, u_{xx} and u_{yy}, at $t = 200$ for the DI-DeepONet (top), PI-DeepONet (middle) and HPI-DeepONet (bottom).