Czech Technical University in Prague
Faculty of Civil Engineering
Department of Mechanics

Deep Learning-Based Modeling and Simulation of Heat Conduction

Master's thesis

Ondřej Šperl

Study programme: Civil Engineering
Branch of study: Structural Engineering of Buildings
Supervisor: doc. Ing. Jan Sýkora, Ph.D.

Prague, January 2025

Thesis Supervisor:
doc. Ing. Jan Sýkora, Ph.D.
Department of Mechanics
Faculty of Civil Engineering
Czech Technical University in Prague
Thákurova 7
160 00 Prague 6
Czech Republic

Declaration

I hereby declare I have written this master's thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

In Prague, January 2025

.....
Ondřej Šperl

Master's Thesis Assignment

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Šperl** Jméno: **Ondřej** Osobní číslo: **495069**

Fakulta/ústav: **Fakulta stavební**

Zadávající katedra/ústav: **Katedra mechaniky**

Studijní program: **Stavební inženýrství - pozemní stavby**

Specializace: **Statika pozemních staveb**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Hluboké učení jako nástroj pro modelování a simulaci vedení tepla

Název diplomové práce anglicky:

Deep Learning-Based Modeling and Simulation of Heat Conduction

Pokyny pro vypracování:

Cílem magisterské práce je konstrukce náhradní modelu problému vedení tepla pomocí hlubokých neuronových sítí. Hluboké učení je podoblast strojového učení, která využívá umělé neuronové sítě k tomu, aby se získávání složitých vzorců a vztahů z dat. Tyto sítě
se skládají z několika vrstev vzájemně propojených uzlů, které umožňují
automatické učení hierarchických reprezentací dat. Student se v rámci svojí práce zaměří na testování různých architektur neuronové sítě. V práci by se měly rovněž zkoumat metody, které kombinují velmi malé soubory dat s fyzikálními rovnicemi. Práce bude napsána v anglickém jazyku.

Seznam doporučené literatury:

[1] Karniadakis, George Em, et al. "Physics-informed machine learning." Nature Reviews Physics 3.6 (2021): 422-440.
[2] Lu, Lu, et al. "DeepXDE: A deep learning library for solving differential equations." SIAM review 63.1 (2021): 208-228.
[3] Chollet, Francois. Deep learning with Python. Simon and Schuster, 2021.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Jan Sýkora, Ph.D. katedra mechaniky FSv

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **01.10.2024** Termín odevzdání diplomové práce: **06.01.2025**

Platnost zadání diplomové práce: _____

doc. Ing. Jan Sýkora, Ph.D. podpis vedoucí(ho) práce	prof. Ing. Jiří Máca, CSc. podpis vedoucí(ho) ústavu/katedry	prof. Ing. Jiří Máca, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání Podpis studenta

Abstract

This Master's thesis deals with implementing deep neural networks for the simulation and modeling of heat conduction. It studies the application of deep neural networks in searching for approximate solutions of partial differential equations and their use in constructing surrogate models. To avoid the relatively common problem of insufficient dataset size for model training, a physics-informed neural network is introduced in this work, which exploits the physical laws represented by partial differential equations in the evaluation of the loss function. This approach leads to a reduction in the size of the training data. The proposed methods and models are assessed with a classical solution based on the finite element method. In a particular case, the comparison is made with the traditional method using polynomial chaos to construct the surrogate model. The effectiveness and limitations of this approach are investigated using various examples simulating the heat conduction problem with varying boundary conditions and thermal properties of the domain under study.

Keywords: Deep neural networks; Physics-informed neural network; Surrogate modeling; Finite element method; Heat conduction

Abstrakt

Tato diplomová práce se zabývá využitím hlubokých neuronových sítí pro simulaci a modelování vedení tepla. Práce studuje aplikaci hlubokých neuronových sítí při hledání přibližného řešení parciálních diferenciálních rovnic a jejich využití ve stavbě náhradních modelů. Aby se předešlo poměrně běžnému problému, kterým je nedostatečná velikost datového souboru pro trénování modelu, je v této práci implementována fyzikálně informovaná neuronová síť, která využívá fyzikální zákony reprezentované parciálními diferenciálními rovnicemi ve vyhodnocení ztrátové funkce. Tento přístup vede ke snížení velikost trénovacích dat. Navržené metody a modely jsou vyhodnoceny s klasickým řešením, které je založené na metodě konečných prvků. V konkrétním případu je porovnání provedeno s tradiční metodou využívající ke stavbě náhradního modelu polynomiální chaos. Efektivita a limity tohoto konceptu jsou zkoumány na různých příkladech simulujících problém vedení tepla s měnícími se okrajovými podmínky a tepelnými vlastnostmi zkoumané domény.

Klíčová slova: Hluboká neuronová síť; Fyzikálně informovaná neuronová síť; Náhradní model; Metoda konečných prvků; Vedení tepla

Acknowledgments

I would like to thank my supervisor doc. Ing. Jan Sýkora, Ph.D. for his guidance, support, and also patience. When we started working together two years ago, he put his trust in me and supported me all the time, and in the process, I learned things that I thought were way beyond my capabilities. Our collaboration had a great influence on my life and I'll always be grateful to him for everything.

I would like to express my gratitude to prof. Pierre Feissel, my French supervisor. I would like to thank him for his valuable advice, inspiring insights, and last but not least for the opportunity to participate in the scientific internship at the University of Technology of Compiègne. That half a year there gave me a lot.

I would also like to thank my girlfriend Kate and my family, for their support and trust in me. Without them, I would never have finished my studies or this work.

This work was supported by the Student Grant Competition of the Czech Technical University in Prague, project No. SGS23/152/OHK1/3T/11, and the research project DEEMA: Design and Optimisation Open Innovation Hub for Composites Modeling and Design, TACR M-ERA.NET 2, project no. TH75020002.

List of Tables

List of Figures

Contents

Chapter 1

Introduction

With the modeling of physical processes, we try to solve problems described by partial differential equation (PDE). Due to the complicated nature (shape of domain, load, nonlinear response of material) in most engineering applications, finding an exact solution of PDE is impossible. In engineering practice, we usually solve such a problem by finding an approximate solution. There are many methods for finding an approximate solution, e.g. the finite element method (FEM) [1], the finite difference method [2], the finite volume method [2], etc. In recent years, however, a considerable amount of research has also been devoted to the deployment of neural networks to solve engineering problems [3]–[5]. It was demonstrated not only that neural networks can be used to obtain an approximate solution of PDE´s [6], but also that they can solve inverse problems [7], be used as surrogate models [8], and even as surrogate model without any additional data [9], i.e data-free approach. This makes neural networks promising for the future and in any case a subject worthy of further research.

In the presented master's thesis we are using deep learning and neural networks for modeling of physical processes. To investigate these, the neural network is used to find an approximate solution of the heat equation, use it as a surrogate model both with and without additional data, and the prediction is compared with traditional established approaches, such as finite elements method and polynomial chaos approximation.

Deep learning models in general usually require a lot of data to be trained. Nevertheless, in practice, we often do not have such a dataset, due to expensive measurements and computationally exhaustive FEM simulations. This issue can be resolved by implementing the scheme called physics-informed neural network (PINN), naturally reducing the required size of the labeled dataset to even a zero. The main idea behind this concept is to incorporate known physics represented by PDEs directly into a loss function of the network. This could be done in several ways: 1) Loss function is assembled based on the unbalanced forces obtained from FD/FEM scheme. 2) We can use automatic

differentiation (AD) for direct evaluation of PDE.

The first approach with applying numerical approximation methods is feasible, and has been done [5], but for complicated problems, it can be demanding. Automatic differentiation however can be applied to almost any problem, is fast, and is already implemented and used for neural networks training in most modern machine learning libraries, such as TensorFlow [10], PyTorch [11] or Theano [12]. Therefore, using AD is quite straightforward and can be applied to different problems with minimal changes to the code.

This thesis is structured as follows: Section 2 describes the methodology and concepts. First, we address the heat conduction modeling, with emphasis on the derivation of the heat balance equation and then we describe the common method used for its solving - FEM. Subsequently, we briefly introduce an artificial neural network, and in more detail, we describe the PINN concept on top of it. Section 3 deals with the practical examples of heat conduction using PINNs and explains their performances. The last section 4 summarizes conclusions from the presented examples and discusses the benefits and drawbacks.

The neural networks and PINNs have significantly grown in popularity in recent years. Therefore the main objective of this work is: i) To study and learn about this concept of incorporating the physical laws into the calculation. ii) Implement PINNs for testing their capabilities as well as shortcomings, iii) and determine whether there is a potential for utilization of PINNs in computational mechanics.

Chapter 2

Methodology

2.1 Heat conduction

Heat conduction is one of three possibilities of heat spreading through an environment, alongside convection and radiation. It is a direct result of collisions between atoms in solid materials that oscillate around their equilibrium position due to kinetic energy. Heat conduction is an important transport phenomenon for civil engineering and it is essential to be able to calculate the related quantities, such as diffusion and condensation of water vapor. We use the modeling of this phenomenon to calculate the thermal performance of the civil engineering structures, and the thermal efficiency of construction details, but also to calculate diffusion phenomena that are directly related to the temperature inside the structure. Heat conduction is usually modeled in three dimensions, but within this thesis, we will only model conduction in two dimensions (2D).

When modeling the 2D heat conduction, we consider the real physical body as a 2D continuum (domain) Ω with its boundary Γ, which, in terms of thermal conductivity, has the properties described by a $\boldsymbol{\lambda}\left[\frac{W}{mK}\right]$ matrix. In the continuum, there is a heat flux vector $\boldsymbol{q}\left[\frac{W}{m^2}\right]$ expressing with its components the direction of the energy flow, and magnitude of this vector is determined by gradient of temperature $\nabla T\left[\frac{^\circ K}{m}\right]$. This behavior is described by Fourier´s law, which postulates that heat flux is equal to the product of the thermal conductivity of the environment and negative temperature gradient:

$$\boldsymbol{q} = -\boldsymbol{\lambda}\nabla T. \tag{2.1}$$

Please note that matrices are denoted by bold upright letters ($\mathbf{A}$) and vectors by bold italic letters ($\boldsymbol{a}$). In practice, we divide heat conduction modeling into two types, depending on the state of the structure. The first type is steady-state heat conduction, where the temperature does not change throughout the structure. In this state, we don´t need to

consider the time dependence of the temperature and therefore it is easier to solve. When we deal with real-world scenarios, it is quite rare for buildings to appear in this state, since it requires temperatures to not change for a period of a couple hours. Nevertheless, it is still frequently used to design the thermal envelope of buildings. We consider unfavorable conditions and calculate the results as a steady state because it is easy, fast, and in most cases provides a sufficient reserve.

The second type is non-stationary heat conduction. In this state, the temperature changes throughout the structure in time, so we need to calculate with the time dependence of the temperature. Solving such problems is more difficult, because to calculate a particular state of construction, we need to know all the previous ones. On the other hand, this approach is more accurate according to reality and in some cases, it is necessary to calculate with this approach.

2.1.1 Steady-state heat conduction

We start with the expression for the steady state and then expand it to a non-stationary state. For the 2D continuum, Fourier´s law is described as:

$$\begin{bmatrix} q_x(\boldsymbol{x}) \\ q_y(\boldsymbol{y}) \end{bmatrix} = - \begin{bmatrix} \lambda_{xx}(\boldsymbol{x}) & \lambda_{xy}(\boldsymbol{x}) \\ \lambda_{xx}(\boldsymbol{x}) & \lambda_{xy}(\boldsymbol{x}) \end{bmatrix} \begin{bmatrix} \frac{\partial T(\boldsymbol{x})}{\partial x} \\ \frac{\partial T(\boldsymbol{x})}{\partial y} \end{bmatrix}. \tag{2.2}$$
$$\boldsymbol{q}(x) = -\boldsymbol{\lambda} \nabla T(\boldsymbol{x})$$

where $\boldsymbol{x}$ is a coordinate vector (x, y).

In thermal analysis, our goal is to calculate temperature distribution on the whole domain. To achieve this, we need to solve a differential equation based on the energy balance, that must be satisfied at each point of it. To derive such an equation, we start from the balance on a square extracted from it, see figure 2.1. And then we scale it down to infinitesimal dimensions, which will provide the equilibrium condition at each point of the domain. The continuum can also have internal heat source $\overline{Q}(\boldsymbol{x})$[1], which is used in engineering practice to model heat increments from heating systems inside constructions, chemical reactions, etc. If we denote the thickness of the continuum by $b\,[m]$, the energy

[1]Please note, that $\overline{Q}$ is marked by $\bar{}$ symbol. It is because the internal heat source is a prescribed quantity and we are viewing it as a load. Everything denoted with $\bar{}$ symbol is also viewed as a prescribed quantity

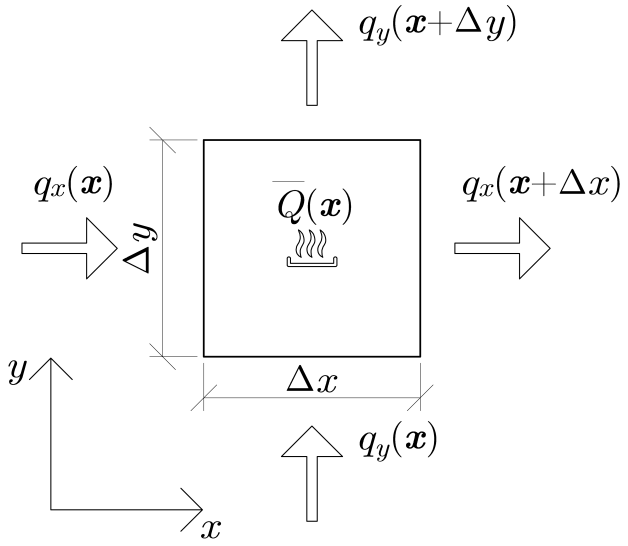


Figure 2.1: Infinitesimal square
The heat symbol labeled by $\overline{Q}(x, y)$ symbolizes the internal heat source. The depicted quantities are functions of coordinate vector $\boldsymbol{x}$, but the increments are only in particular coordinate Δx and Δy.

balance of the extracted square can be written as:

$$\begin{aligned} &q_x(\boldsymbol{x})b\Delta y + q_y(\boldsymbol{x})b\Delta x + \overline{Q}(\boldsymbol{x})b\Delta x\Delta y = q_x(\boldsymbol{x}+\Delta x)b\Delta y + q_y(\boldsymbol{x}+\Delta y)b\Delta x \\ &\frac{q_x(\boldsymbol{x})}{\Delta x} + \frac{q_y(\boldsymbol{x})}{\Delta y} + \overline{Q}(\boldsymbol{x}) = \frac{q_x(\boldsymbol{x}+\Delta x)}{\Delta x} + \frac{q_y(\boldsymbol{x}+\Delta y)}{\Delta y} \\ &-\frac{q_x(\boldsymbol{x}+\Delta x) - q_x(\boldsymbol{x})}{\Delta x} - \frac{q_y(\boldsymbol{x}+\Delta y) - q_y(\boldsymbol{x})}{\Delta y} + \overline{Q}(\boldsymbol{x}) = 0 \end{aligned} \quad . \tag{2.3}$$

After the infinitesimal transition, i.e. Δx and Δy approach zero in the limit, we convert this condition for a point in the domain Ω as:

$$\begin{aligned} -\frac{\partial q_x(\boldsymbol{x})}{\partial x} - \frac{\partial q_y(\boldsymbol{x})}{\partial y} + \overline{Q}(\boldsymbol{x}) &= 0 \\ -\begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{bmatrix} \begin{bmatrix} q_x(\boldsymbol{x}) \\ q_y(\boldsymbol{x}) \end{bmatrix} + \overline{Q}(\boldsymbol{x}) &= 0, \boldsymbol{x} \in \Omega. \\ -\nabla^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x}) + \overline{Q}(\boldsymbol{x}) &= 0 \end{aligned} \tag{2.4}$$

Since $\boldsymbol{q}(\boldsymbol{x}) = -\boldsymbol{\lambda}(\boldsymbol{x})\nabla T(\boldsymbol{x})$, we can see that energy balance condition leads to a differential equation, where the solution is the temperature distribution on the domain $T(\boldsymbol{x})$. To obtain the unique solution, each point of the domain boundary Γ must have a prescribed boundary condition. For the purpose of this thesis, we consider three types of boundary

conditions, that are typically applied.

The first type is the Dirichlet boundary condition. This condition applied to heat conduction has the form of a prescribed temperature:

$$T(\boldsymbol{x}) - \overline{T}(\boldsymbol{x}) = 0, \boldsymbol{x} \in \Gamma_D. \tag{2.5}$$

The second is the Neumann boundary condition or the second type boundary condition. For heat conduction it can have a form of prescribed heat flux in the normal direction:

$$\begin{bmatrix} n_x(\boldsymbol{x}) & n_y(\boldsymbol{x}) \end{bmatrix} \begin{bmatrix} q_x(\boldsymbol{x}) \\ q_y(\boldsymbol{y}) \end{bmatrix} - \overline{q}(\boldsymbol{x}) = 0 \\ \boldsymbol{n}^{\mathsf{T}}(\boldsymbol{x})\boldsymbol{q}(\boldsymbol{x}) - \overline{q}(\boldsymbol{x}) = 0 \end{bmatrix}, \boldsymbol{x} \in \Gamma_N. \tag{2.6}$$

And the third is the Robin boundary condition, which can be applied as heat transfer due to laminar airflow in the boundary layer around the domain as:

$$\overline{q}(\boldsymbol{x}) = \alpha(\boldsymbol{x}) \left(T(\boldsymbol{x}) - \overline{T}_0(\boldsymbol{x}) \right), \boldsymbol{x} \in \Gamma_R, \tag{2.7}$$

where $\alpha \left[\frac{J}{m^2 K s} \right]$ is the heat transfer coefficient, and $\overline{T}_0(\boldsymbol{x})$ is the temperature in the boundary layer. Another typical Robin condition could be the radiation boundary condition, where the heat is emitted to the environment due to the radiation. However, this BC is not used in this thesis.

2.1.2 Non-stationary heat conduction

When the temperature of the domain is changing in time, we have to consider the temperature as its function:

$$T = T(x, y, t) = T(\boldsymbol{x}, t). \tag{2.8}$$

Given this, the constitutive equation defined by Fourier´s law is expressed as:

$$\begin{bmatrix} q_x(\boldsymbol{x}, t) \\ q_y(\boldsymbol{y}, t) \end{bmatrix} = - \begin{bmatrix} \lambda_{xx}(\boldsymbol{x}) & \lambda_{xy}(\boldsymbol{x}) \\ \lambda_{xx}(\boldsymbol{x}) & \lambda_{xy}(\boldsymbol{x}) \end{bmatrix} \begin{bmatrix} \frac{\partial T(\boldsymbol{x}),t}{\partial x} \\ \frac{\partial T(\boldsymbol{x}),t}{\partial y} \end{bmatrix}. \\ \boldsymbol{q}(\boldsymbol{x}, t) = -\boldsymbol{\lambda} \nabla T(\boldsymbol{x}, t) \tag{2.9}$$

To compute the temperature distribution, we again need to solve a differential equation based on the energy balance of the infinitesimal square of the domain. We will use the same approach as in the case of steady-state conduction, but since the temperature is changing in time, we need to take into account energy accumulation inside the matter of the domain Ω. Each physical material has its thermal capacity $c_v \left[\frac{J}{kgK} \right]$, which indicates

how much heat we must supply to one kilogram of material to raise its temperature by one temperature degree. With this quantity, we can create a term that will include heat accumulation in the energy balance:

$$Q_c = \Delta m c_v(\boldsymbol{x}) \Delta T, \tag{2.10}$$

where $\Delta m\,[kg]$ is the weight of the extracted square from the domain and ΔT is the average temperature change of this square. If the domain has thickness $b\,[m]$, then the average energy balance on it during time $\Delta t\,[s]$ has the form:

$$\begin{aligned}
&q_x(\boldsymbol{x},t)b\Delta y\Delta t + q_y(\boldsymbol{x},t)b\Delta x\Delta t + \overline{Q}(\boldsymbol{x},t)b\Delta x\Delta y\Delta t = \Delta m c_v(\boldsymbol{x})\Delta T(\boldsymbol{x},t)\Delta t \\
&+ q_x(\boldsymbol{x}+\Delta x,t)b\Delta y\Delta t + q_y(\boldsymbol{x}+\Delta y,t)b\Delta x\Delta t \\
&\frac{q_x(\boldsymbol{x},t)}{\Delta x} + \frac{q_y(\boldsymbol{x},t)}{\Delta y} + \overline{Q}(\boldsymbol{x},t) = \frac{\Delta m}{b\Delta x\Delta y} c_v(\boldsymbol{x}) \frac{\Delta T(\boldsymbol{x},t)}{\Delta t} + \frac{q_x(\boldsymbol{x}+\Delta x,t)}{\Delta x} + \frac{q_y(\boldsymbol{x}+\Delta y,t)}{\Delta y} \\
&-\frac{q_x(\boldsymbol{x}+\Delta x,t) - q_x(\boldsymbol{x},t)}{\Delta x} - \frac{q_y(\boldsymbol{x}+\Delta y,t) - q_y(\boldsymbol{x},t)}{\Delta y} - \frac{\Delta m}{b\Delta x\Delta y} c_v(\boldsymbol{x}) \frac{\Delta T(\boldsymbol{x},t)}{\Delta t} + \overline{Q}(\boldsymbol{x},t) = 0
\end{aligned} \tag{2.11}$$

After the limit transition $\Delta x \to 0$, $\Delta y \to 0$, $\Delta t \to 0$, we get the energy balance for a point in domain Ω:

$$\begin{aligned}
-\frac{\partial q_x(\boldsymbol{x},t)}{\partial x} - \frac{\partial q_y(\boldsymbol{x},t)}{\partial} - \frac{\hat{\rho}(\boldsymbol{x})}{b} c_v(\boldsymbol{x}) \frac{\partial T(\boldsymbol{x},t)}{\partial t} + \overline{Q}(\boldsymbol{x},t) = 0 \\
-\nabla^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x},t) - \frac{\hat{\rho}(\boldsymbol{x})}{b} c_v(\boldsymbol{x}) \frac{\partial T(\boldsymbol{x},t)}{\partial t} + \overline{Q}(\boldsymbol{x},t) = 0
\end{aligned} , \boldsymbol{x} \in \Omega. \tag{2.12}$$

Due to the limit transition, the term $\frac{\Delta m}{\Delta x \Delta y}$ was transformed to $\hat{\rho}(\boldsymbol{x}) \left[\frac{kg}{m^2}\right]$, which has physical meaning of areal density. And since it applies that $\rho = \frac{\hat{\rho}}{b}$, we can write:

$$-\nabla^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x},t) + \overline{Q}(\boldsymbol{x},t) = \rho(\boldsymbol{x}) c_v(\boldsymbol{x}) \frac{\partial T(\boldsymbol{x},t)}{\partial t}, \boldsymbol{x} \in \Omega. \tag{2.13}$$

The solution to this equation is a function of temperature distribution in space and time $T(\boldsymbol{x},t)$. To obtain a unique solution, each point of domain boundary Γ at each time t must have a prescribed boundary condition:

$$\begin{aligned}
T(\boldsymbol{x},t) - \overline{T}(\boldsymbol{x},t) = 0, \boldsymbol{x} \in \Gamma_D, \\
\boldsymbol{n}^{\mathsf{T}}(\boldsymbol{x}) \boldsymbol{q}(\boldsymbol{x},t) - \overline{q}(\boldsymbol{x},t) = 0, \boldsymbol{x} \in \Gamma_N, \\
\overline{q}(\boldsymbol{x},t) = \alpha(\boldsymbol{x}) \left(T(\boldsymbol{x},t) - \overline{T}_0(\boldsymbol{x},t)\right), \boldsymbol{x} \in \Gamma_R,
\end{aligned} \tag{2.14}$$

and the whole domain Ω must have prescribed initial condition at time $t = 0$:

$$T(\boldsymbol{x}, t) - \overline{T}_{in}(\boldsymbol{x}, t) = 0, x \in \Omega. \tag{2.15}$$

In the general case, for both steady-heat state and non-stationary heat conduction we are not able to obtain a closed-form solution. We are therefore using numerical methods to obtain an approximate solution. One of the most used methods is the finite element method, which will be also used for this thesis. The results obtained by it will be used for the training of physics-informed neural networks and for evaluation of the results.

2.2 Finite element method

The finite element method (FEM) is perhaps the most popular method for numerically solving PDE´s. The name of this method comes from the methodology itself when the domain on which a PDE is solved is divided into small parts called elements by a generated appropriate mesh. This way the continuum is discretized and converted to a system, that has a finite number of points. The solution is then approximated by assembling individual functions, that model the behavior of particular elements. All of this is done in such a way, that converts the PDE into a system of algebraic equations.

2.2.1 FEM for steady-state heat conduction

The typical first step in finite element analysis involves creating a so-called weak formulation of the original PDE. The weak formulation allows us to reduce demands for the solution. The equilibrium enforced in the PDE does not need to be maintained absolutely, but instead applies for only certain test functions satisfying the equation 2.16. Such a solution is called a weak solution. The original PDE is called strong formulation and its solution is strong solution. To obtain a weak formulation, we multiply both sides of the original equation by the test function, and do the integral over the entire domain:

$$\int_{\Omega} \delta T(\boldsymbol{x})(-\nabla^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x}) + \overline{Q}(\boldsymbol{x}))d\boldsymbol{x} = 0, \tag{2.16}$$

, where δT is the test function. If we are able to satisfy this equation for any test function, it means that we have a strong solution. But if we are only able to satisfy it for a certain set of test functions, we have a weak solution. So the operation performed in equation 2.16 does not itself make the original equation a weak solution, but the fact that we do not require that this equality be satisfied for every test function does.

Now we can apply the Gauss theorem to convert this to a more favorable form:

$$\begin{aligned} 0 = &- \int_{\Gamma} \delta T(\boldsymbol{x}) \boldsymbol{n}(\boldsymbol{x})^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x}) d\boldsymbol{x} + \int_{\Omega} (\nabla \delta T(\boldsymbol{x}))^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x}) d\boldsymbol{x} \\ &+ \int_{\Omega} \delta T(\boldsymbol{x}) \overline{Q}(\boldsymbol{x}) d\boldsymbol{x} = - \int_{\Gamma_D} \delta T(\boldsymbol{x}) \boldsymbol{n}(\boldsymbol{x})^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x}) d\boldsymbol{x} \\ - &\int_{\Gamma_N} \delta T(\boldsymbol{x}) \boldsymbol{n}(\boldsymbol{x})^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x}) d\boldsymbol{x} - \int_{\Gamma_R} \delta T(\boldsymbol{x}) \boldsymbol{n}(\boldsymbol{x})^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x}) d\boldsymbol{x} \\ &+ \int_{\Omega} (\nabla \delta T(\boldsymbol{x}))^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x}) d\boldsymbol{x} + \int_{\Omega} \delta T(\boldsymbol{x}) \overline{Q}(\boldsymbol{x}) d\boldsymbol{x} \end{aligned} \tag{2.17}$$

We specify the weak solution in such a way, that $\delta T = 0$ on the boundary Γ_D. This way we don´t have to consider the term $\int_{\Gamma_D} \delta T(\boldsymbol{x}) \boldsymbol{n}(\boldsymbol{x})^{\mathsf{T}} \boldsymbol{q}(\boldsymbol{x}) d\boldsymbol{x}$, since it equals zero. After inserting the constitutive equation and the corresponding boundary conditions, we get:

$$\begin{aligned} \int_{\Omega} (\nabla \delta T(\boldsymbol{x}))^{\mathsf{T}} \boldsymbol{\lambda}(\boldsymbol{x}) \nabla T(\boldsymbol{x}) d\boldsymbol{x} &+ \int_{\Gamma_R} \delta T(\boldsymbol{x}) \alpha(\boldsymbol{x}) T(\boldsymbol{x}) d\boldsymbol{x} \\ = - \int_{\Gamma_N} \delta T(\boldsymbol{x}) \overline{q}(\boldsymbol{x}) d\boldsymbol{x} &+ \int_{\Gamma_R} \delta T(\boldsymbol{x}) \alpha(\boldsymbol{x}) \overline{T}_0(\boldsymbol{x}) d\boldsymbol{x}. \\ &+ \int_{\Omega} \delta T(\boldsymbol{x}) \overline{Q}(\boldsymbol{x}) d\boldsymbol{x} \end{aligned} \tag{2.18}$$

Now we can discretize the domain to create finite elements using the Galerkin method. The Galerkin method is based on the principle of weighted residuals, where the residual is orthogonal to the chosen finite-dimensional subspace. This method ensures that the error between the exact solution and the approximate solution is minimized in a well-defined mathematical sense, see [1]. Using the Galerkin method, we approximate the solution as a linear combination of n basis function:

$$T(\boldsymbol{x}) \approx \sum_{i=1}^{n} N_i(\boldsymbol{x}) r_i = \begin{bmatrix} N_1(\boldsymbol{x}) & N_2(\boldsymbol{x}) & \dots & N_n(\boldsymbol{x}) \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix} = \boldsymbol{N}(\boldsymbol{x}) \boldsymbol{r}, \tag{2.19}$$

where $\boldsymbol{r}$ is a vector of unknown coefficients. And the test functions are chosen with the same shape as the basis functions:

$$\delta T(\boldsymbol{x}) \approx \boldsymbol{N}(\boldsymbol{x}) \boldsymbol{\delta r}, \tag{2.20}$$

where $\boldsymbol{\delta r}$ is again a vector of coefficients independent of $\boldsymbol{r}$. The number of basis functions n corresponds to the number of degrees of freedom, or the number of nodes in the network discretising the domain. For the derivative of the temperature function and the test

function, we can write:

$$\begin{aligned} \nabla T(\boldsymbol{x}) &= \nabla \boldsymbol{N}(\boldsymbol{x})\boldsymbol{r} = \mathbf{B}(\boldsymbol{x})\boldsymbol{r}, \\ \nabla \delta T(\boldsymbol{x}) &= \nabla \boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r} = \mathbf{B}(\boldsymbol{x})\boldsymbol{\delta r}. \end{aligned} \tag{2.21}$$

After substituting these approximations to the equation 2.18 we get:

$$\begin{aligned} &\int_\Omega (\mathbf{B}(\boldsymbol{x})\boldsymbol{\delta r})^\mathsf{T} \boldsymbol{\lambda}(\boldsymbol{x})\mathbf{B}(\boldsymbol{x})\boldsymbol{r}d\boldsymbol{x} + \int_{\Gamma_R} (\boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r})^\mathsf{T} \alpha(\boldsymbol{x})\boldsymbol{N}(\boldsymbol{x})\boldsymbol{r}d\boldsymbol{x} \\ &= -\int_{\Gamma_N} (\boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r})^\mathsf{T} \overline{q}(\boldsymbol{x})d\boldsymbol{x} + \int_{\Gamma_R} (\boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r})^\mathsf{T} \alpha(\boldsymbol{x})\overline{T}_0(\boldsymbol{x})d\boldsymbol{x}. \\ &\qquad + \int_\Omega (\boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r})^\mathsf{T} \overline{Q}(\boldsymbol{x})d\boldsymbol{x} \end{aligned} \tag{2.22}$$

The vector $\boldsymbol{\delta r}$ can be put in front of the integral since it is not a function of x. And because it appear in every term, the equation 2.22 will be automatically satisfied for every $\boldsymbol{\delta r}$, if:

$$\begin{aligned} \mathbf{K}\boldsymbol{r} &= \boldsymbol{f}_N + \boldsymbol{f}_R + \boldsymbol{f}_Q = \boldsymbol{f}, \\ \mathbf{K} &= \int_\Omega \mathbf{B}(\boldsymbol{x})^\mathsf{T} \boldsymbol{\lambda}(\boldsymbol{x})\mathbf{B}(\boldsymbol{x})\boldsymbol{r}d\boldsymbol{x} + \int_{\Gamma_R} \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \alpha(\boldsymbol{x})\boldsymbol{N}(\boldsymbol{x})\boldsymbol{r}d\boldsymbol{x}, \\ \boldsymbol{f}_N &= -\int_{\Gamma_N} \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \overline{q}(\boldsymbol{x})d\boldsymbol{x}, \\ \boldsymbol{f}_R &= \int_{\Gamma_R} \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \alpha(\boldsymbol{x})\overline{T}_0(\boldsymbol{x})d\boldsymbol{x}, \\ \boldsymbol{f}_Q &= \int_\Omega \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \overline{Q}(\boldsymbol{x})d\boldsymbol{x}. \end{aligned} \tag{2.23}$$

$\mathbf{K}$ is the conductivity matrix and $\boldsymbol{f}$ is the load vector. In the finite element method, these terms are determined by localization from the individual elements. It is a common practice to choose the basis functions as continuous piece-wise differentiable in such a way that they are equal to one at the corresponding node, decrease to zero at neighboring nodes on the common elements, and are zero everywhere else. Choosing basis functions with these properties allows us to calculate the increments from individual elements simply. We calculate the local conductivity matrix and local load vector on each element and then localize this increment in $\mathbf{K}$ and $\boldsymbol{f}$. In this work, only a simple mesh composed of rectangular bilinear elements will be used, where basis functions on each element are described by the following equations:

$$\begin{aligned} N_1(\boldsymbol{x}) &= \frac{1}{4ab}(x-a)(y-b),\ N_2(\boldsymbol{x}) = \frac{1}{4ab}(x+a)(b-y), \\ N_3(\boldsymbol{x}) &= \frac{1}{4ab}(x+a)(y+b),\ N_4(\boldsymbol{x}) = \frac{1}{4ab}(a-x)(y+b), \end{aligned} \tag{2.24}$$

where a and b are the dimensions of the rectangle. Local basis function and its derivative corresponding to the particular element are in matrix form written as:

$$\begin{aligned}\boldsymbol{N}(\boldsymbol{x}) &= \frac{1}{4ab}\begin{bmatrix}(x-a)(y-b) & (x+a)(b-y) & (x+a)(y+b) & (a-x)(y+b)\end{bmatrix}, \\ \mathbf{B}(\boldsymbol{x}) &= \frac{1}{4ab}\begin{bmatrix}y-b & b-y & y+b & y+b \\ x-a & x+a & x+a & a-x\end{bmatrix}.\end{aligned} \tag{2.25}$$

Since we approximate the solution as a linear combination of basis functions, after evaluating the integrals obtained by localizing the contributions from the individual elements, the expression $\mathbf{K}\boldsymbol{r} = \boldsymbol{f}$ leads to a system of linear equations. Solving this system we obtain an approximate solution of the original differential equation.

2.2.2 FEM for non-stationary heat conduction

To obtain a finite element solution for the non-stationary problem, we will follow the same steps as in the case of steady-state conduction. We begin by multiplying the equation with the test function and integrating it over the entire domain:

$$\int_\Omega \delta T(\boldsymbol{x},t)\left(-\nabla^\mathsf{T}\boldsymbol{q}(\boldsymbol{x},t)+\overline{Q}(\boldsymbol{x},t)\right)d\boldsymbol{x} = \int_\Omega \delta T(\boldsymbol{x},t)\left(\rho(\boldsymbol{x})c_v(\boldsymbol{x})\frac{\partial T(\boldsymbol{x},t)}{\partial t}\right)d\boldsymbol{x}. \tag{2.26}$$

Applying Gauss theorem, we get:

$$\begin{aligned}0 =& -\int_\Gamma \delta T(\boldsymbol{x},t)\boldsymbol{n}^\mathsf{T}(\boldsymbol{x})\boldsymbol{q}(\boldsymbol{x},t)d\boldsymbol{x} + \int_\Omega (\nabla\delta T(\boldsymbol{x},t))^\mathsf{T}\boldsymbol{q}(\boldsymbol{x},t)d\boldsymbol{x} \\ &+ \int_\Omega \delta T(\boldsymbol{x},t)\overline{Q}(\boldsymbol{x},t)d\boldsymbol{x} - \int_\Omega \delta T(\boldsymbol{x},t)\left(\rho(\boldsymbol{x})c_v(\boldsymbol{x})\frac{\partial T(\boldsymbol{x},t)}{\partial t}\right)d\boldsymbol{x} \\ =& -\int_{\Gamma_D} \delta T(\boldsymbol{x},t)\boldsymbol{n}(\boldsymbol{x})^\mathsf{T}\boldsymbol{q}(\boldsymbol{x},t)d\boldsymbol{x} - \int_{\Gamma_N} \delta T(\boldsymbol{x},t)\overline{\boldsymbol{q}}(\boldsymbol{x},t)d\boldsymbol{x} \\ &- \int_{\Gamma_R} \delta T(\boldsymbol{x},t)\alpha(\boldsymbol{x})\left(T(\boldsymbol{x},t)-\overline{T}_0(\boldsymbol{x},t)\right)d\boldsymbol{x} + \int_\Omega (\nabla\delta T(\boldsymbol{x},t))^\mathsf{T}\boldsymbol{q}(\boldsymbol{x},t)d\boldsymbol{x} \\ &+ \int_\Omega \delta T(\boldsymbol{x},t)\overline{Q}(\boldsymbol{x},t)d\boldsymbol{x} - \int_\Omega \delta T(\boldsymbol{x},t)\left(\rho(\boldsymbol{x})c_v(\boldsymbol{x})\frac{\partial T(\boldsymbol{x},t)}{\partial t}\right)d\boldsymbol{x}\end{aligned} \tag{2.27}$$

Further, we define a weak formulation by choosing $\delta T = 0$ on the boundary Γ_D, which ensures that the integral on this boundary is zero and we use the Galerkin method to discretize the solution in space. The discretized solution, the test function, and their

derivatives are expressed in the form:

$$\begin{aligned} T(\boldsymbol{x},t) &= \boldsymbol{N}(\boldsymbol{x})\boldsymbol{r}(t), \\ \delta T(\boldsymbol{x},t) &= \boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r}(t), \\ \nabla T(\boldsymbol{x},t) &= \nabla \boldsymbol{N}(\boldsymbol{x})\boldsymbol{r}(t) = \mathbf{B}(\boldsymbol{x})\boldsymbol{r}(t), \\ \nabla \delta T(\boldsymbol{x},t) &= \nabla \boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r}(t) = \mathbf{B}(\boldsymbol{x})\boldsymbol{\delta r}(t). \end{aligned} \tag{2.28}$$

By substituting the approximations 2.28 into the equation 2.27 we get:

$$\begin{aligned} &\int_\Omega (\mathbf{B}(\boldsymbol{x})\boldsymbol{\delta r}(t))^\mathsf{T} \boldsymbol{\lambda}(\boldsymbol{x})\mathbf{B}(\boldsymbol{x})\boldsymbol{r}(t)d\boldsymbol{x} + \int_\Omega (\boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r}(t))^\mathsf{T} \left(\rho(\boldsymbol{x})c_v(\boldsymbol{x})\boldsymbol{N}(\boldsymbol{x})\frac{\partial \boldsymbol{r}(t)}{\partial t}\right) d\boldsymbol{x} \\ &+ \int_{\Gamma_R} (\boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r}(t))^\mathsf{T} \alpha(\boldsymbol{x})\boldsymbol{N}(\boldsymbol{x})\boldsymbol{r}(t)d\boldsymbol{x} = -\int_{\Gamma_N} (\boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r}(t))^\mathsf{T} \overline{q}(\boldsymbol{x},t)d\boldsymbol{x} \\ &+ \int_{\Gamma_R} (\boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r}(t))^\mathsf{T} \alpha(\boldsymbol{x})\overline{T}_0(\boldsymbol{x},t)d\boldsymbol{x} + \int_\Omega (\boldsymbol{N}(\boldsymbol{x})\boldsymbol{\delta r}(t))^\mathsf{T} \overline{Q}(\boldsymbol{x},t)d\boldsymbol{x}. \end{aligned} \tag{2.29}$$

Since $\boldsymbol{\delta r}(t)$ is independent on $\boldsymbol{x}$ and again appears in every integral, we can factor it out, and the equation is satisfied for every $\boldsymbol{\delta r}(t)$ if:

$$\begin{aligned} &\int_\Omega \mathbf{B}(\boldsymbol{x})^\mathsf{T} \boldsymbol{\lambda}(\boldsymbol{x})\mathbf{B}(\boldsymbol{x})\boldsymbol{r}(t)d\boldsymbol{x} + \int_\Omega \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \rho(\boldsymbol{x})c_v(\boldsymbol{x})\boldsymbol{N}(\boldsymbol{x})\frac{\partial \boldsymbol{r}(t)}{\partial t}d\boldsymbol{x} \\ &+ \int_{\Gamma_R} \boldsymbol{N}^\mathsf{T} \alpha(\boldsymbol{x})\boldsymbol{N}(\boldsymbol{x})\boldsymbol{r}(t)d\boldsymbol{x} = -\int_{\Gamma_N} \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \overline{q}(\boldsymbol{x},t)d\boldsymbol{x} \\ &+ \int_{\Gamma_R} \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \alpha(\boldsymbol{x})\overline{T}_0(\boldsymbol{x},t)d\boldsymbol{x} + \int_\Omega \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \overline{Q}(\boldsymbol{x},t)d\boldsymbol{x}. \end{aligned} \tag{2.30}$$

Equation 2.30 can be further rewritten in matrix form:

$$\begin{aligned} \mathbf{K}\boldsymbol{r}(t) + \mathbf{C}\frac{\partial \boldsymbol{r}(t)}{\partial t} &= \boldsymbol{f}_N(t) + \boldsymbol{f}_R(t) + \boldsymbol{f}_Q(t) = \boldsymbol{f}(t), \\ \mathbf{K} &= \int_\Omega \mathbf{B}(\boldsymbol{x})^\mathsf{T} \boldsymbol{\lambda}(\boldsymbol{x})\mathbf{B}(\boldsymbol{x})d\boldsymbol{x} + \int_{\Gamma_R} \boldsymbol{N}^\mathsf{T} \alpha(\boldsymbol{x})\boldsymbol{N}(\boldsymbol{x})d\boldsymbol{x}, \\ \mathbf{C} &= \int_\Omega \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \rho(\boldsymbol{x})c_v(\boldsymbol{x})\boldsymbol{N}(\boldsymbol{x})d\boldsymbol{x}, \\ \boldsymbol{f}_N &= -\int_{\Gamma_N} \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \overline{q}(\boldsymbol{x},t)d\boldsymbol{x}, \\ \boldsymbol{f}_R &= \int_{\Gamma_R} \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \alpha(\boldsymbol{x})\overline{T}_0(\boldsymbol{x},t)d\boldsymbol{x}, \\ \boldsymbol{f}_Q &= \int_\Omega \boldsymbol{N}(\boldsymbol{x})^\mathsf{T} \overline{Q}(\boldsymbol{x},t)d\boldsymbol{x}, \end{aligned} \tag{2.31}$$

where $\mathbf{K}$ is the conductivity matrix, $\mathbf{C}$ is the capacity matrix, and $\boldsymbol{f}$ is the load vector. We can see, that on the left-hand side, we have the derivative of unknown coefficients

with respect to time and moreover both $\boldsymbol{r}$ and $\boldsymbol{f}$ are the functions of time. In general, there are two fundamental approaches to address this issue. We can apply the finite element discretization and create finite elements in both space and time or discretize the time separately using the time discretization method representing the classical approach, which is also applied here to our problem. The time discretization method starts with dividing the time interval for which we solve the equation into n intervals Δt. The solution in time $i+1$ is then linearly approximated as:

$$\begin{aligned} \boldsymbol{r}(t) &= (1-\tau)\boldsymbol{r}^i + \tau \boldsymbol{r}^{i+1}, \\ \tau &= \frac{t - t_i}{\Delta t}. \end{aligned} \tag{2.32}$$

Similarly we approximate the load vector $\boldsymbol{f}$:

$$\boldsymbol{f}(t) \approx (1-\tau)\boldsymbol{f}^i(\boldsymbol{x}) + \tau \boldsymbol{f}^{i+1}, \tag{2.33}$$

and the derivative of temperature with respect to time is approximated as:

$$\frac{\partial \boldsymbol{r}(t)}{\partial t} \approx \frac{1}{\Delta t}\left(\boldsymbol{r}^{i+1} - \boldsymbol{r}^i\right). \tag{2.34}$$

With discretized time we can rewrite the equation 2.31 as:

$$\left(\tau \mathbf{K} + \frac{1}{\Delta t}\mathbf{C}\right)\boldsymbol{r}^{i+1} = (1-\tau)\boldsymbol{f}^i + \tau \boldsymbol{f}^{i+1} + \left(\frac{1}{\Delta t}\mathbf{C} - (1-\tau)\mathbf{K}\right)\boldsymbol{r}^i, \tag{2.35}$$

which leads again to a system of linear equations (SLE). First $\boldsymbol{r}^i$ is set as the initial condition and each subsequent r^i is obtained by solving SLE composed of contributions from previous steps and boundary conditions. The choice of parameter τ affects the stability of the calculation and the accuracy of the obtained solution.

2.3 Neural network

To introduce neural networks, it is a good idea to first describe what is machine learning since neural networks are part of it. Machine learning is a paradigm of so-called 'artificial intelligence', Which is a term that first appeared in this context in the 1950s. Back then, a group of computer scientists wondered whether computers could think and replace humans in processes that require human intelligence, and more or less concluded that it is possible. Artificial intelligence (AI) therefore means everything that replaces humans in their intellectual actions and there are two main paradigms of how to achieve this intelligence [13].

The First paradigm is symbolic artificial intelligence. Here we are trying to create a computer program by manually defining the instructions and logical rules for the computer to follow, and based on that, it should provide us demanded results. However it turned out that there are many activities that the human brain can perform on a daily basis, but at the same time is not able to create a sufficient amount of explicit rules that a computer could follow to execute the same thing, and thus replace it. This applies for example to language translation, converting speech to text, or computer vision.

For such tasks, the second paradigm has proven to be much more effective, and it is none other than machine learning. In machine learning, we are trying to develop some kind of statistical algorithm that is capable of finding its own parameters and recognizing patterns based on labeled data, specified metrics, or some kind of score system. This process of finding the best parameters without any explicit rules is called learning or training - therefore the name machine learning. It has many subcategories, but perhaps the most popular one today is neural networks. Their name has been inspired by neurobiology and neurons in the human nervous system. A neural network consists of artificial neurons that are in a certain way connected, and their purpose is to transform the data that are passed through them. Typical and perhaps the simplest scheme of neuron connection, is to stack neurons in layers and then connect these layers one on top of the other, see figure 2.2. The first layer of a network is called the input layer. It represents the input and this layer itself does not do any transformation of the data. The last layer is called the output layer, producing the final output of the whole network. Each layer between these two is called a hidden layer. In the previous chapter 1 we introduced the term 'deep learning'. There are many definitions, nevertheless, it can be defined as training of a neural network with 2 or more hidden layers [13]. Although the name 'artificial neuron' itself seems to be complicated, it simply denotes a mathematical function. It has an input value in the form of a scalar or tensor (convolutional neural network), predefined mathematical operations, and an output value, again in the form of a scalar or tensor.

2.3.1 Simple neural network

The simplest neural network has an architecture consisting only of densely connected sometimes called fully connected layers. In these layers, every neuron in a particular layer has a connection to every neuron from the previous layer. Regarding individual neurons, they are composed of a bias and an activation function, with both the input and the output taking the form of real scalar values. Mathematically, the behavior of a neuron located inside the neural network can be described by the following equation:

$$y = a(x - b), \tag{2.36}$$

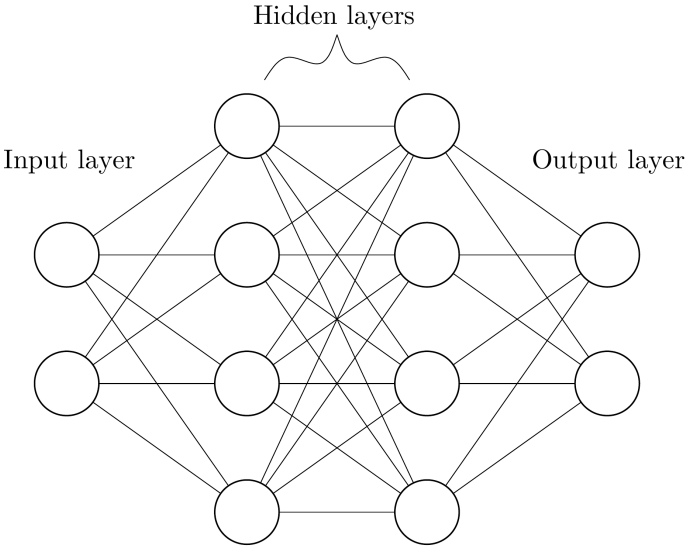


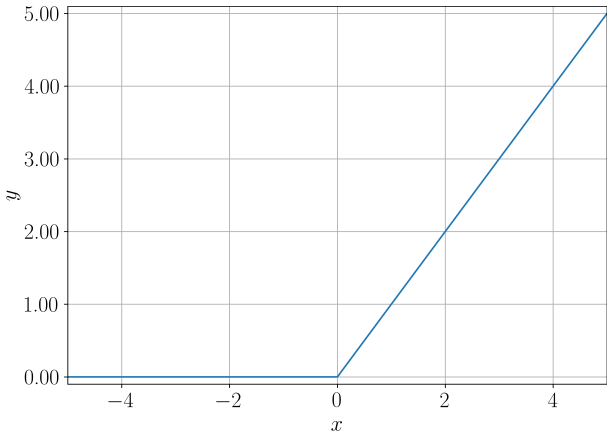
Figure 2.2: Neural network scheme

where y is the output of the neuron, x is the input of the neuron, b is the bias value and a is some non-linear activation function. Since neurons have connections with each other, the input to the neuron x is in general assembled from the output of other neurons and weights of connections between them. In general, the bias value can be zero, i.e. the neuron does not have to contain this parameter. Nevertheless, in most cases, we operate with neurons, where bias is included. The activation function on the other hand is very important because it basically defines the behavior of the neuron. Perhaps, the most common functions, see figure 2.3, are sigmoid, hyperbolic tangent, and today very popular ReLU (rectified linear unit). Although ReLU is frequently used in all kinds of models for its efficiency, it is unsuitable for usage in PINNs, as we will explain in the section 2.4.

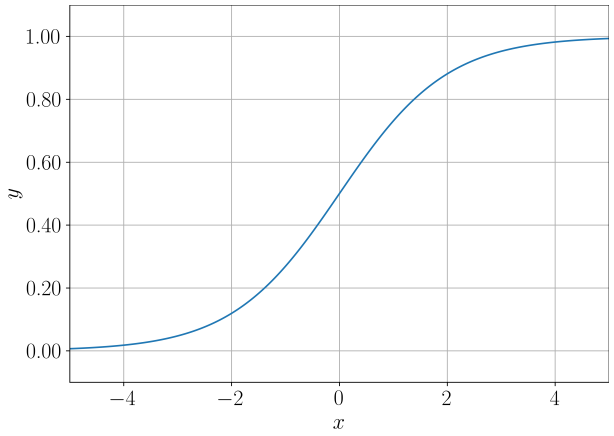
As stated above, in the densely connected layer, each neuron is connected to each neuron in the previous layer. Each connection has a certain weight scaling the increment from the particular neuron. So the output of the i-th neuron from l-th layer can be written as:

$$y_i^l = a\left(\sum_{k=1}^{K} w_k \cdot y_k^{l-1} - b_i\right), \tag{2.37}$$

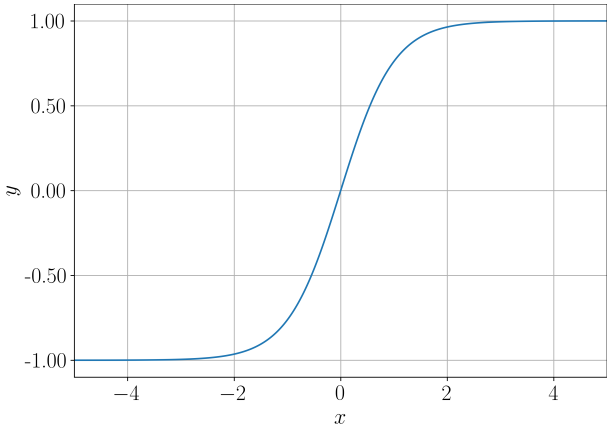
where w_k is the weight of the connection, and y_k^{l-1} the output from k-th neuron of the previous layer, b_i is the bias of i-th layer, and K is the number of neurons in the previous layer. By following this logic, if we look at the layer as a whole, the output from the l-th



(a) ReLU



(b) sigmoid



(c) hyperbolic tangent

Figure 2.3: Activation functions

layer can be written in matrix form as:

$$\boldsymbol{y}^l = \mathbf{a}\left(\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1K} \\ w_{21} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ w_{I1} & \dots & \dots & w_{IK} \end{bmatrix} \begin{bmatrix} y_1^{l-1} \\ y_2^{l-1} \\ \vdots \\ y_K^{l-1} \end{bmatrix} - \begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_I^l \end{bmatrix}\right) = \mathbf{a}\left(\mathbf{W}^l \boldsymbol{y}^{l-1} - \boldsymbol{b}^l\right). \tag{2.38}$$

Where w_{ik} is the weight of the connection between the i-th neuron in the current layer and the k-th neuron in the previous layer, y_k^{l-1} is the output of the k-th neuron in the previous layer, b_i is the i-th neuron bias, and K and I are the numbers of neurons in the previous and current layer respectively. The symbol l denoting the layer number can take the values from 2 to L, where L is the total number of layers and $l = 1$ represents the input layer. Activation function $\mathbf{a}()$ is written as a tensor, representing the piece-wise application of $a()$ to each element of the tensor. This may not seem so sophisticated, but if you consider that input and output to this simple neural network can be a vector of any length, it is clear that the network can perform a variety of different tasks related to

the classification [14] or regression [15] problems. The purpose of the network is mostly determined by the activation function and the type of the last layer. With all of this, the fully connected network can perform challenging tasks, but to achieve this we need to find proper weights and biases in the process called training. An integral part of the training process is to define the appropriate loss function.

2.3.2 Loss function

The loss function, also called the cost function, is something that is absolutely crucial for model training. With this function, we define how the weights[2] of the model should be changed to provide better performance by minimizing it during training.

When we are using a neural network as a classifier or as a regression tool, perhaps the easiest way how to measure the network´s loss, is to compare its results with the correct values that it should return. This requires a dataset of inputs together with labeled outputs, ensuring several features, such as relevance, quality, diversity, or consistency to be effective for training models. The approach utilizing a labeled dataset for model training is called supervised learning. The simplest loss functions that are used for evaluating the differences between the model prediction and labeled output are mean squared error (MSE) and mean absolute error (MAE):

$$MSE = \frac{1}{I} \cdot \sum_{i=1}^{I} \left(y_i^L - y_i^{true} \right)^2, \tag{2.39}$$

$$MAE = \frac{1}{I} \cdot \sum_{i=1}^{I} \left| y_i^L - y_i^{true} \right|. \tag{2.40}$$

Where y_i^L is the output of the i-th neuron in the last layer, y_i^{true} is a correct value that the model should return and I is the number of neurons in the last layer. Using these error metrics, one can measure the discrepancy in the model predictions and determine whether the model has been trained sufficiently.

2.3.3 Training of the model

Training of a model is a process, where one tries to find the best parameters for model predictions. During the training model makes predictions that are passed as an argument to a loss function and a particular loss value is calculated. The goal is to minimize the loss value, as a smaller loss indicates better model performance on the given task. The

[2] In many publications and articles, the weights of the model refer to all parameters of the model, that could be changed during training, i.e. also biases in our case.

most common technique today in NN training is to use some kind of optimizer based on stochastic gradient descent.

In calculus, the gradient is a vector whose components are obtained by deriving a function $f : \mathbb{R}^n \to \mathbb{R}$ at point $p = [x_1, ..., x_n]$ with respect to its variables

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f(p)}{\partial x_1} \\ \vdots \\ \frac{\partial f(p)}{\partial x_n} \end{bmatrix}. \tag{2.41}$$

The gradient is pointing to the direction of the largest increment of the function value. This is very frequently used in optimization wherever possible because if we take the negative value of the gradient, we will get the direction of the largest decrease. This capability has proven really efficient in finding the local minima of a function, and in our case, it can be used to minimize the loss function. We just need to calculate the derivative of the loss function with respect to model weights, and then adjust the weights in the direction of the gradient.

If we look at equation 2.38, we can see that the dense neural network is a composite function that consists of individual layers because the output from one layer serves as the input for the next layer. The equation of the whole network with L layers can be then written as:

$$\mathbf{Y}^L = \mathbf{a}^L \left(\mathbf{W}^L \mathbf{a}^{L-1} \left(\mathbf{W}^{L-1} \mathbf{a}^{L-2} \left(... \mathbf{a}^2 \left(\mathbf{W}^2 \mathbf{I} - \hat{\mathbf{b}}^2 \right) ... \right) - \hat{\mathbf{b}}^{L-1} \right) - \hat{\mathbf{b}}^L \right), \tag{2.42}$$

where index L denotes the last layer, index 2 denotes first hidden layer and $\mathbf{I}$ is input to the network. Notice that input $\mathbf{I}$ as well as output $\mathbf{Y}^L$ are written with the capital letter as matrices. It is contradictory to our definition of dense network input and output as vectors. And technically it is true. One training example is indeed represented by a vector, but in real-world computation, it would be inefficient to send each training example through the network individually. In practice, we instead send data in groups called batches. However, if we store each input example in a column of a matrix, we can see from Equation 2.38 that the behavior of the network remains unchanged by this representation. The only difference is that output from each layer is not a vector of size corresponding to the number of neurons I, but it is a matrix with the number of rows corresponding to the number of neurons I, and the number of columns corresponding to

the batch size b:

$$\boldsymbol{y}^l = \mathbf{a}\left(\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1K} \\ w_{21} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ w_{I1} & \dots & \dots & w_{IK} \end{bmatrix} \begin{bmatrix} y_{11}^{l-1} & & y_{1b}^{l-1} \\ y_{21}^{l-1} & \dots & y_{2b}^{l-1} \\ \vdots & & \vdots \\ y_{K1}^{l-1} & & y_{Kb}^{l-1} \end{bmatrix} - \begin{bmatrix} b_1^l & & b_1^l \\ b_2^l & \dots & b_2^l \\ \vdots & & \vdots \\ b_I^l & & b_I^l \end{bmatrix}\right) = \mathbf{a}\left(\mathbf{W}^l \boldsymbol{y}^{l-1} - \hat{\mathbf{b}}^l\right). \tag{2.43}$$

The $\hat{\mathbf{b}}^l$ is a matrix that is created by copying the same vector $\boldsymbol{b}^l$ column-wise, because each neuron only has one bias, but the dimensions must match with the batch size. The $\hat{}$ symbol represents such an operation. In modern tensor libraries such as NumPy, TensorFlow, or PyTorch this broadcasting is done automatically, but for manually expressing the derivatives of the loss function with respect to network weights we need to be aware of this fact.

Considering that any neural network can contain a large number of hidden layers, computing the derivatives of the loss function with respect to each weight individually can be computationally demanding. What we can do instead is take advantage of the chain rule properties, calculate the derivative of the loss function with respect to the last layer of the network, and then propagate it back to compute derivatives with respect to all layers. This process is called back propagation. When we calculate the derivatives with backpropagation, we are calculating the effect of each network parameter on the loss function, therefore it is often read that the network error is backpropagated.

In the context of working with tensors and calculating their derivatives, Einstein notation is applied to ensure the formulas remain simple. Einstein notation denotes tensors as lowercase letters with indices and implies that if one index appears more than once in one term, it denotes summation over all possible values of that index. For example, matrix multiplication in Einstein notation can be written as:

$$c_{ij} = a_{ik} b_{kj} = \sum_k a_{ik} b_{kj} = \mathbf{AB}. \tag{2.44}$$

As mentioned, the chain rule is used to backpropagate the derivatives of the loss function, so the first term that must be calculated is the derivative with respect to output from the last layer $\mathbf{Y}^L$. For the demonstration of this process, the mean squared error is considered as the loss metric, see equation 2.39. With Einstein notation, it can be written as:

$$\begin{aligned} l &= \frac{1}{n} 1_{ij} u_{ij}^2, \\ u_{ij} &= y_{ij}^L - y_{ij}^{true} \end{aligned} \tag{2.45}$$

The 1_{ij} is a second-order tensor full of ones with dimensions ij corresponding to the

number of neurons in the last layer and batch sizer. It is there to denote the fact that l is computed by summation of square differences on all possible elements. The derivative of l with respect to y_{ij}^L is then:

$$\frac{\partial l}{\partial y_{pq}^L} = \frac{\partial l}{\partial u_{ij}} \frac{\partial u_{ij}}{\partial y_{pq}^L} = \frac{1}{n} 2_{ij} u_{ij} \frac{\partial u_{ij}}{\partial y_{pq}^L} = \frac{1}{n} 2_{ij} u_{ij} \delta_{ip} \delta_{jq} = \frac{1}{n} 2_{pq} u_{pq}. \tag{2.46}$$

The δ_{ip} and the δ_{jq} symbols are the Kronecker delta tensors that are equal to one when their indices match and zero otherwise:

$$\delta_{ab} = \begin{cases} 1, & \text{if } a = b, \\ 0, & \text{if } a \neq b. \end{cases} \tag{2.47}$$

With Einstein notation the term $\frac{1}{n} 2_{ij} u_{ij} \delta_{ip} \delta_{jq}$ means $\frac{1}{n} \sum_i \sum_j 2_{ij} u_{ij} \delta_{ip} \delta_{jq}$, and thanks to Kronecker delta tensors, only combinations that are not equal to zero are if $i = p$ and $j = q$, therefore it can be simplified to $\frac{1}{n} 2_{pq} u_{pq}$. From this result, it is clear that the derivative of the loss with respect to the output of the last layer can be written in matrix form as:

$$\frac{\partial l}{\partial \mathbf{Y^L}} = \frac{2}{n} \left(\mathbf{Y}^L - \mathbf{Y}^{true} \right). \tag{2.48}$$

After obtaining $\frac{\partial l}{\partial \mathbf{Y^L}}$ we can apply the chain rule further to get the derivatives with respect to weights and biases of the last layer and output of the previous layer. And since the formula of each densely connected layer has the same shape, this principle is applicable to every layer of the model, not just the last one. So the labeling is changed from a capital L to a lowercase l to obtain those equations for the general layer.

Using the equation 2.43, with the 'weighted input' as $\mathbf{Z}^l = \mathbf{W}^l \mathbf{Y}^{l-1} - \hat{\mathbf{b}}^l$, the derivative with respect to layer weights can be written as:

$$\begin{aligned} \frac{\partial l}{\partial w_{pr}^l} &= \frac{\partial l}{\partial y_{ij}^l} \frac{\partial y_{ij}^l}{\partial z_{ij}^l} \frac{\partial z_{ij}^l}{\partial w_{pr}^l} \\ = \frac{\partial l}{\partial y_{ij}^l} a'^l \left(z_{ij}^l \right) \frac{\partial w_{ik}^l y_{kj}^{l-1} - \hat{b}_{ij}}{\partial w_{pr}^l} &= \frac{\partial l}{\partial y_{ij}^l} a'^l \left(z_{ij}^l \right) y_{kj}^{l-1} \delta_{ip} \delta_{kr} \\ = \frac{\partial l}{\partial y_{pj}^l} a'^l \left(z_{pj}^l \right) y_{rj}^{l-1} &= \frac{\partial l}{\partial y_{pj}^l} a'^l \left(z_{pj}^l \right) \left(y_{jr}^l \right)^\mathsf{T}, \end{aligned} \tag{2.49}$$

Where $a'^l()$ is the derivative of the activation function. This can be rewritten in matrix form as:

$$\frac{\partial l}{\partial \mathbf{W}^l} = \left(\frac{\partial l}{\partial \mathbf{Y}^l} \odot \mathbf{a}'^l \left(\mathbf{Z}^l \right) \right) \left(\mathbf{Y}^{l-1} \right)^\mathsf{T}, \tag{2.50}$$

where the $\odot$ symbol denotes the Hadamard product (element-wise multiplication).

To calculate the derivative with respect to bias, we must first express the broadcasting of the vector to the matrix. With Einstein notation, we can write it as $\hat{b}^l_{ij} = b_i 1_j$. The derivative with respect to layer bias is then:

$$\begin{aligned} \frac{\partial l}{\partial b^l_p} &= \frac{\partial l}{\partial y^l_{ij}} \frac{\partial y^l_{ij}}{\partial z^l_{ij}} \frac{\partial z^l_{ij}}{\partial \hat{b}^l_{ij}} \frac{\partial \hat{b}^l_{ij}}{\partial b^l_p} \\ = \frac{\partial l}{\partial y^l_{ij}} a'^l \left(z^l_{ij}\right) (-1)\, 1_j \delta_{ip} &= -\frac{\partial l}{\partial y^l_{pj}} a'^l \left(z^l_{pj}\right) 1_j. \end{aligned} \tag{2.51}$$

In the matrix form, the equation is expressed as:

$$\frac{\partial l}{\partial \boldsymbol{b}^l} = -\left(\frac{\partial l}{\partial \mathbf{Y}^l} \odot \mathbf{a}'^l \left(\mathbf{Z}^l\right)\right) \mathbf{1}, \tag{2.52}$$

where $\mathbf{1}$ is a column vector full of ones with the size j correspond to the batch size.

For propagating the derivatives back to previous layers, we must calculate the derivative with respect to the output from the previous layer.

$$\begin{aligned} \frac{\partial l}{\partial y^{l-1}_{rq}} &= \frac{\partial l}{\partial y^l_{ij}} \frac{\partial y^l_{ij}}{\partial z^l_{ij}} \frac{\partial z^l_{ij}}{\partial \hat{y}^{l-1}_{rq}} \\ = \frac{\partial l}{\partial y^l_{ij}} a'^l \left(z^l_{ij}\right) w^l_{ik} \delta_{kr} \delta_{jq} &= \frac{\partial l}{\partial y^l_{iq}} a'^l \left(z^l_{iq}\right) w^l_{ir} \\ &= \left(w^l_{ri}\right)^{\mathsf{T}} \frac{\partial l}{\partial y^l_{iq}} a'^l \left(z^l_{iq}\right), \end{aligned} \tag{2.53}$$

which in matrix form is rewritten as:

$$\frac{\partial l}{\partial \mathbf{Y}^{l-1}} = \left(\mathbf{W}^l\right)^{\mathsf{T}} \left(\frac{\partial l}{\partial \mathbf{Y}^l} \odot \mathbf{a}'^l \left(\mathbf{Z}^l\right)\right). \tag{2.54}$$

With those four equations, we are able to compute derivatives of loss with respect to any weights and biases for model that consist of densely connected layers. The only equation that would be case-dependent, is the derivative of the loss function with respect to the output from the last layer since it is specifically derived for the mean squared error metric. Nevertheless, the shape of the derivative remains the same—a matrix of the same size as the output from the last layer, so the remaining three equations and principles behind them are universally applicable. We simply calculate the derivative of loss with respect to

output from the last layer and then propagate it back using these three general equations:

$$\boxed{\begin{array}{rl} \frac{\partial l}{\partial \mathbf{W}^l} & = \left(\frac{\partial l}{\partial \mathbf{Y}^l} \odot \mathbf{a}'^l\left(\mathbf{Z}^l\right)\right)\left(\mathbf{Y}^{l-1}\right)^\mathsf{T} \\ \frac{\partial l}{\partial \boldsymbol{b}^l} & = -\left(\frac{\partial l}{\partial \mathbf{Y}^l} \odot \mathbf{a}'^l\left(\mathbf{Z}^l\right)\right)\mathbf{1} \\ \frac{\partial l}{\partial \mathbf{Y}^{l-1}} & = \left(\mathbf{W}^l\right)^\mathsf{T}\left(\frac{\partial l}{\partial \mathbf{Y}^l} \odot \mathbf{a}'^l\left(\mathbf{Z}^l\right)\right) \end{array}}. \tag{2.55}$$

After obtaining the necessary derivatives, we can adjust the network weights in the negative direction of the gradient, which should reduce the loss function. Therefore, individual batches are sent through the network during training, and after finishing the calculation of each batch, the weights are adjusted. The problem is, how much we should move in the way of the negative gradient direction. Because if we would just change the weights by the size of the whole gradient, i.e. $w^{t+1} = w^t - \frac{\partial l}{\partial w^t}$, for the majority of the real world problems the change in the weights would be simply too large, and it would not result in reducing the loss function. Therefore parameter called learning rate η is employed to determine how much of the gradient size is applied: $w^{t+1} = w^t - \eta\frac{\partial l}{\partial w^t}$. The learning rate parameter is used in some form in virtually all modern optimizers based on stochastic gradient descent, such as AdaGrad, RMSprop, or Adam, which is used for the purposes of this thesis. If the value is too small, convergence can be slow or the optimizer can freeze in some local minima of the loss function. Oppositely, the convergence is also problematic for large values of learning rate, because the optimizer never gets to the minima of the loss function. The default learning rate for the Adam optimizer in TensorFlow and Pytorch is $1e-3$, however, you can never predict the optimal learning rate with certainty. Sometimes it is not possible to find the learning rate for which the model would converge, since the convergence is also dependent on the model architecture, loss function, size and generality of the dataset, initialization of the weights, and batch size and composition of each batch. So in general case convergence is not guaranteed and therefore the design and the training of the network is rather a heuristic process.

To explain more why the batch size affects model convergence, we must clarify what stochastic gradient descent is. When we are training the model, we are trying to minimize the loss function defined by the properties mentioned above. This means that the shape of the loss function is affected, among other things by all the data from the training dataset, hence the true gradient we are trying to use for minimization would need to be computed on the whole dataset, not on just one batch. This is however in most cases not feasible simply because computers often do not have the memory capacity to send the whole dataset through the network at once. On the other hand, if we compute the gradient only on one training example, the probability that it will have the same direction as the true gradient defined by the whole dataset is rather low. Therefore the main idea behind

stochastic gradient descent is to randomly scramble examples in the training dataset and then create batches from it. And if those randomly scrambled batches are large enough, there is a good chance that the gradient calculated from those does not differ so much from the true gradient computed for the whole dataset. It explains why batch size also affects the convergence of the model.

During the training, an epoch refers to the process where the model passes through all the training data and adjusts its weights. This is one of the parameters that need to be set. We can either set the fixed number of epochs for which the model is trained or specify some conditions that stop the training, such as a desired value of loss function. However, the loss value on the training dataset itself is not a sufficient metric for evaluation of the model's true capabilities, since we can not determine if the training dataset is general enough. The model may exhibit good loss values on the training dataset, but predictions on unseen data can be significantly worse. This phenomenon is referred to as overfitting and takes place when the model adjusts its weights to perfectly match the training data, which are not sufficiently general. As the number of weights in the model increases, so does the risk of overfitting, due to the model's enhanced capacity to approximate more complex functions. It can be said in some sense that larger models in general require larger datasets.

In deep learning, there is therefore a common practice to divide available data into three groups. The first group is the training dataset that is used for training the model and adjusting its weights. The second group is the validation dataset, which is not used for model training (adjusting model weights), but after completing each epoch, the model will make predictions that are compared against this data. This approach allows tracking the model's prediction ability on non-training data during training to determine if overfitting/underfitting is occurring. And the third group is the testing dataset. This dataset is used post-training to evaluate the model's prediction ability on an independent dataset, as adjusting hyperparameters during training might lead to overfitting to the training and validation datasets, resulting in poorer general prediction performance. The hyperparameter of the model is basically anything that can be set before the training, except trainable parameters like weights and biases. The term hyperparameter can refer to the number of hidden layers, number of neurons in each layer, activation function, batch size, number of epochs, loss function and its possible parameters, learning rate, optimizer, etc.

Previously we derived formulas for backpropagation which we can use for training the network, but this is not how it is typically done in practice. In modern libraries like TensorFlow or PyTorch, there is no manual derivation required, instead, automatic differentiation is used during the training process. It is based on the idea that every

algorithm, no matter how complex it may be, is only performing a series of basic mathematical operations, i.e. addition, subtraction, multiplication, division, etc., and simple mathematical functions like sins, cos, exp, root, etc.. Automatic differentiation is taking advantage of this fact. These libraries are then able to record every operation that is done with a variable and use the chain rule to efficiently compute required derivatives. This is highly beneficial as it only requires creating a variable, utilizing it in a function, requesting its derivatives, and these libraries can automatically provide them, including derivatives of intermediate results and higher-order derivatives, see [16]. With automatic differentiation, we can use basically any model architecture, define custom layers and loss functions, and the derivatives are always automatically provided for us.

2.3.4 Optimizer

By optimizer, we refer to the algorithm that is updating model weights after each batch. In most cases, we use optimizers that are based on stochastic gradient descent (SGD). In this thesis, we use a SGD-based optimizer called Adam. The first paragraph is devoted to the concept of the stochastic gradient descent algorithm followed by the description of Adam optimizer.

Stochastic gradient descent

As the name suggests, this algorithm uses the gradient to optimize a given quantity. In our case, it is the loss function of a model with its weights as variables. After obtaining the necessary derivatives, we can simply adjust the weights with particular learning rate as:

$$w^{t+1} = w^t - \eta \frac{\partial l}{\partial w^t}, \tag{2.56}$$

where w^t is general weight of a model in t-th iteration, η is the learning rate, $\frac{\partial l}{\partial w^t}$ is the gradient of l with respect to weights and w^{t+1} is the new updated weight. Although this algorithm is relatively effective, it does not contain any mechanisms for dealing with local minima, other than the learning rate. An additional problem is that as the algorithm approaches the desired minimum, the gradient's magnitude decreases, resulting in an unnecessary extension of the convergence time. Adam tries to solve these shortcomings with the following measures.

Adam

The name Adam stands for adaptive moment estimation and is a combination of two optimizers, SGD with momentum and RMSprop.

Stochastic gradient descent with momentum tries to resolve the issue of freezing in the local minima by simulating the moment of inertia, similar to how a marble rolling from a hill would behave - if the marble has enough inertia, it does not stop in local valleys. The momentum is realized by using an exponential moving average:

$$\begin{aligned} w^{t+1} &= w^t - \eta m_t, \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial l}{\partial w^t} \end{aligned} \tag{2.57}$$

where m_t is momentum, m_{t-1} is momentum from previous step and β_1 is a chosen constant. The default value of β_1 is 0.9 in both TensorFlow and PyTorch. By multiplying previous momentum with parameter β_1 smaller than 1 it is assured that the influence of the previous gradients is smaller with each step while keeping the momentum.

RMSprop enriches stochastic gradient descent by adaptively changing the learning rate for each weight separately during training. It achieves this again using an exponential moving average:

$$\begin{aligned} w^{t+1} &= w^t - \frac{\eta}{\sqrt{v_t + \epsilon}}\frac{\partial l}{\partial w^t}, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial l}{\partial w^t}\right)^2, \end{aligned} \tag{2.58}$$

where v_t is the weighted sum of squares of past gradients, v_{t-1} is this sum from the previous step, β_2 is a constant with default value 0.999 in TensorFlow and PyTorch and ϵ is also a small constant added for numerical stability that is by default equal to $1 \cdot 10^{-7}$ and $1 \cdot 10^{-8}$ in TensorFlow and PyTorch, respectively. When we divide the learning rate with the square root of v_t, we normalize individual derivatives with respect to their history. If the derivative with respect to a certain weight tends to be small, it automatically increases in size because the denominator will be smaller, and similarly, it automatically decreases in size if it tends to be large. This is useful, for example, when the neighborhood of the minimum of a function in some direction is very mild compared to other directions. The derivative in this direction will be automatically scaled, making convergence faster, and the optimizer is also able to move in the direction corresponding to the minimum. Since the moving average is used, it helps to keep the stability of the convergence.

The Adam optimizer combines both those approaches and adjusts the weights as follows:

$$\begin{aligned} w^{t+1} &= w^t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}, \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t}. \end{aligned} \tag{2.59}$$

Constants β_1, β_2, and ϵ in Adam optimizer have the default values specified above and the default learning rate is 0.001. The $\hat{m}_t$ and $\hat{v}_t$ modify m_t and v_t because both of these values are initialized as zero, which in early steps causes unconditional scaling of the gradient that can eventually deflect the algorithm. Thanks to these modifications Adam in the first iteration corresponds to a SGD algorithm. With growing exponents in the calculation of the hat terms, the impact of the momentum and the weighted sum of squares of the previous gradient is increasing, and as a result of that the $\hat{m}_t$ and $\hat{v}_t$ more and more correspond to m_t and v_t. In normal use cases, the default constants β_1, β_2, and ϵ are not changed and the only modified parameter is the learning rate η. It applies also to this thesis, the only optimizer parameter changed during experimentation for this thesis, is the learning rate.

2.4 Physics-informed neural network - PINN

Neural networks have been shown to be effective tools for finding correlations between data and also as approximators. Therefore, there are tendencies to incorporate them into engineering practice in solving various physical problems. Difficulties arise due to the size of the labeled dataset, needed for successful training of the model. In civil engineering problems, we usually have access to only small datasets, which is in contrast to typical deep-learning tasks. However, compared to the standard neural network applications such as image classification, speech-to-text conversion, language translation, etc., applications in engineering tasks have one major advantage. The relations between our data and physical quantities are known to us in advance. Those relations are expressed in formulas and differential equations mostly as equilibrium equations. Physics physics-informed neural network is something built on this knowledge and it is defined as a neural network that has the physical law of the given problem incorporated into its loss function.

There are many applications of PINNs and one of them is the focus of this work, namely the use of a neural network for obtaining an approximate solution of a partial differential equation. For our purpose, the incorporation can be done in several ways. The first way is to create some kind of residuum obtained by the finite difference method, finite element method, etc. We can for example use the fact that these methods lead to a system of equations in a form $\mathbf{K}\boldsymbol{r} = \boldsymbol{f}$. Left side matrix $\mathbf{K}$ can be then multiplied with a solution predicted by the neural network and compared to a right side vector f:

$$l = \left\| \mathbf{K}\boldsymbol{y}^{\boldsymbol{L}} - \boldsymbol{f} \right\|^2 . \tag{2.60}$$

By minimizing this loss function, the neural network learns the physical laws indirectly

from the approximation defined by a particular method. This approach can then be combined with data from other simulations or experiments.

The second approach, which we also use in this thesis, is to use automatic differentiation to create residuum directly from the network itself. Since the automatic differentiation is already implemented by most modern machine learning libraries, this strategy is easy to deploy and perhaps more natural than the first approach. As mentioned, with automatic differentiation we are able to quickly obtain the derivatives based on the operations that are done with a variable. This is very useful, because if we choose network architecture in a way, so that it approximates the solution of the differential equation, i.e. it has the same input and output as the solution, we can easily calculate its derivatives, from whose the residuum is composed.

To demonstrate this strategy further, we focus on the stationary heat conduction problem defined by equation 2.4. Here the solution is a function that takes x and y coordinates as arguments and returns the temperature value T at a point determined by these coordinates. To approximate it, we set our neural network to have two neurons in the input layer and one neuron in the output layer, to be able to predict the temperature based on the spatial coordinates. This way we also retrieve the derivatives of any degree of temperature with respect to particular coordinates, after each prediction by automatic differentiation. With this ability, we can assemble the differential equation for a given point, and check if the imposed balance is maintained. We simply take the first derivative of the temperature with respect to each coordinate, multiply it with $-\boldsymbol{\lambda}$ to get the heat fluxes, differentiate them again by the appropriate coordinate, and together with the eventual heat source we can construct the original PDE, see the figure 2.4. If the term is equal to zero, the PDE at this point is satisfied. Naturally, adding more points during the training process ensures better fitted neural network model with higher capability of approaching the exact solution. Therefore, we create a dataset of points within the domain and evaluate the PDE at each of them. To measure the deviation, we can simply take the mean squared error of the residuum at each point:

$$l_{PDE} = \frac{1}{N_{PDE}} \sum_{i=1}^{N_{PDE}} \left(\frac{\partial}{\partial x_i} \left(-\boldsymbol{\lambda}_i \frac{T_i}{\partial x_i} \right) + \frac{\partial}{\partial y_i} \left(-\boldsymbol{\lambda}_i \frac{T_i}{\partial y_i} \right) + \overline{Q_i} \right)^2, \tag{2.61}$$

where T_i and $\overline{Q_i}$ are the predicted temperature and prescribed heat source at a given point, $\boldsymbol{\lambda}_i$ is thermal conductivity at a given point, and N_{PDE} is the number of points in the domain, on which we are evaluating the loss.

Evaluating the l_{PDE} however is not enough, as we must also impose the boundary condition, to be able to approach a specific desired solution. Therefore, a typical approach to training this type of PINN is to create a dataset of points located on the boundaries and

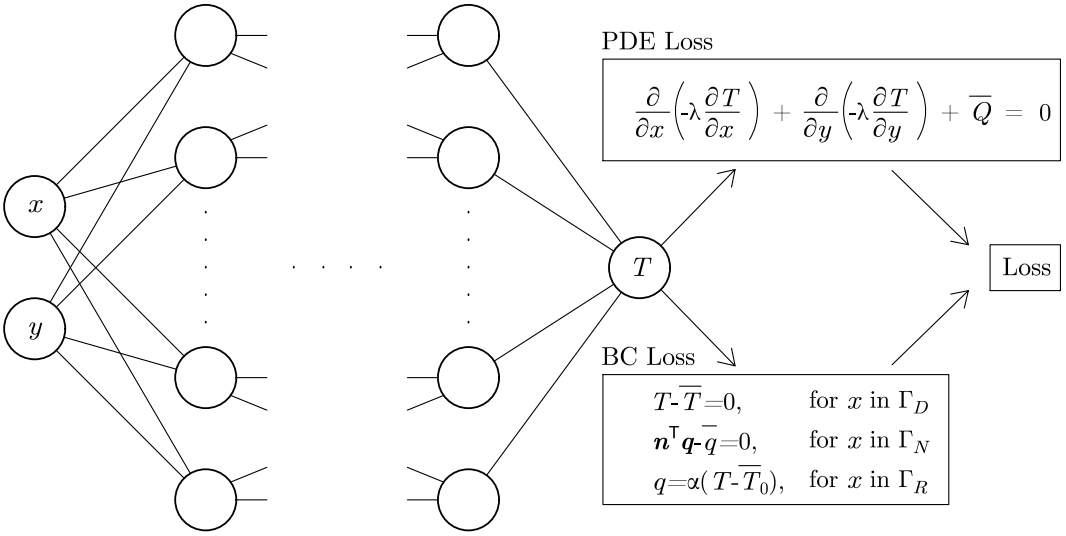


Figure 2.4: PINN scheme

then simply check whether the network prediction matches the given boundary condition. We split the boundary points in the dataset based on the type of the condition, and enforce those conditions by minimizing the following loss function:

$$\begin{aligned} l_{BC} &= l_{\Gamma_D} + l_{\Gamma_N} + l_{\Gamma_R}, \\ l_{\Gamma_D} &= \frac{1}{N_D}\sum_{i=1}^{N_D}\left(T_i - \overline{T}_i\right)^2, \boldsymbol{x} \in \Gamma_D, \\ l_{\Gamma_N} &= \frac{1}{N_N}\sum_{i=1}^{N_N}\left(\boldsymbol{n}_i^\mathsf{T}\boldsymbol{q}_i - \overline{q}_i\right)^2, \boldsymbol{x} \in \Gamma_N, \\ l_{\Gamma_R} &= \frac{1}{N_R}\sum_{i=1}^{N_R}\left(\alpha_i\left(T_i - \overline{T}_{0,i}\right) - \overline{q}_i\right)^2, \boldsymbol{x} \in \Gamma_R. \end{aligned} \tag{2.62}$$

N_D, N_N and N_R are the number of points on the boundary, where Dirichlet, Neumann and Robin BC´s are prescribed, T_i is the predicted temperature, $\boldsymbol{q}_i$ is the predicted heat-flux obtained by automatic differentiation, α_i is the heat transfer coefficient, and $\overline{T}_i$, $\overline{T}_{0,i}$ and $\overline{q}_i$ are prescribed temperature, prescribed temperature in the boundary layer, and prescribed heat-flux in the normal direction.

Minimizing both l_{PDE} and l_{BC} ensures that the neural network approaches the exact

solution. Consequently, these loss functions are combined into a single objective:

$$l = c_{PDE} l_{PDE} + c_{BC} l_{BC}, \tag{2.63}$$

where c_{PDE} and c_{BC} are the coefficients scaling the increments from the losses if needed. In this thesis, we consider both equal to one, since the models are able to converge even if the values from the beginning of the training differ by several orders of magnitude. As already outlined, the error between the exact solution and the the model prediction is not only specified by the value of the loss function l, but also by the quality of the dataset utilized during the training. Here it is mostly affected by the number of points, which are defined by some sort of mesh similar to the FEM or randomly generated. If we provide a sufficient number of points and successfully minimize the loss function, we obtain the approximate solution of the strong form directly. We demonstrate the approach on the 2D steady-state heat conduction, but it is applicable for any PDE and again can be further combined with data from experiments or different simulations. This direct evaluation by automatic differentiation is not only limited for finding the approximate solution, but for almost any problem, where NN is predicting some physics based quantity, like inverse problems, etc.

In the subsection 2.3.1 it has been mentioned that the ReLU activation function is not suitable for usage in the PINNs. The ReLU is only differentiable at most once since it is a constant for $x < 0$ and a linear function for $x \geq 0$. The second derivative of the network output with respect to its input would always be zero and we would not be able to train the network.

Chapter 3

Examples

This section focuses on evaluating the proposed methods and models against a classical finite element method solution. The effectiveness and limitations of the PINN concept are explored through various examples that simulate heat conduction problems with different boundary conditions and thermal properties of the studied domain. Additionally, in one specific case, a comparison is made with the traditional method that employs polynomial chaos to build the surrogate model. All code used in the examples is available as a GitHub repository at [17].

In this thesis, the following examples are realized with the help of the TensorFlow library. We work with default dense layers, optimizers, and activation functions provided by the Keras framework. The Glorot uniform implemented in KERAS is a default initializer of weights for dense layers, which is here used to start the initialization from the random uniform distribution. $[-l, l]$. The limit l is calculated as:

$$l = \sqrt{\frac{6}{f_{in} + f_{out}}}, \tag{3.1}$$

where the f_{in} and f_{out} are the number of input and output units to the particular layer, respectively. For a network that consists purely of dense layers, these numbers are equal to the number of neurons in the previous and the next layer.

In the following examples, we consider a rectangle domain Ω with the boundaries defined as:

$$\begin{aligned} \Omega &= \{(x, y) \in \mathbb{R}^2 \mid 0 \leq x \leq 0.5,\ 0 \leq y \leq 0.25\}, \\ \Gamma_D &= \{(0, y) \mid y \in [0, 0.25]\} \cup \{(0.5, y) \mid y \in [0, 0.25]\}, \\ \Gamma_N &= \{(x, 0) \mid x \in [0, 0.5]\} \cup \{(x, 0.25) \mid x \in [0, 0.5]\}. \end{aligned} \tag{3.2}$$

The thermal conductivity within this geometric domain has the following shape:

$$\boldsymbol{\lambda}(\boldsymbol{x}) = \begin{bmatrix} \lambda(\boldsymbol{x}) & 0 \\ 0 & \lambda(\boldsymbol{x}) \end{bmatrix} = \lambda(\boldsymbol{x}), \tag{3.3}$$

i.e. it is a scalar function of $\boldsymbol{x}$.

3.1 PINN as an approximation of the solution

In this section, we will provide a few examples to obtain particular results and assess the limitations of the procedure explaining how to obtain an approximate solution for 2D stationary heat conduction.

For the evaluation of the prediction capability, we compare the computed results with the FEM. To achieve this, we use uniform mesh with bilinear rectangular elements described in the subsection 2.2.1. The mesh contains 100 nodes in the x direction and 50 nodes in the y direction, which in total makes 5000 nodes and 4851 finite elements. The nodes also serve as a training dataset for the neural network, so the total number of nodes in the domain N_{PDE} is 5000 and the number of nodes on boundaries related to the Dirichlet N_D and Neumann N_N boundary conditions is 100 and 200, respectively.

3.1.1 Example 1 - Thermal conductivity defined as a paraboloid function

In this example, we consider λ to be a parabolic function:

$$\lambda(\boldsymbol{x}) = 5x^2 + 5y^2 + xy + 0.05. \tag{3.4}$$

The thermal conductivity is the lowest in the left bottom corner and gradually increases in the right top corner. The constant 0.05 is here introduced since the zero thermal conductivity does not have any physical meaning. The boundary conditions are set as follows:

$$\begin{aligned} \overline{T}(0, y) &= -5, \quad y \in [0, 0.25], \\ \overline{T}(0.5, y) &= 15, \quad y \in [0, 0.25], \\ \overline{q}(x, 0) &= 0, \quad x \in [0, 0.5], \\ \overline{q}(x, 0.25) &= 0, \quad x \in [0, 0.5]. \end{aligned} \tag{3.5}$$

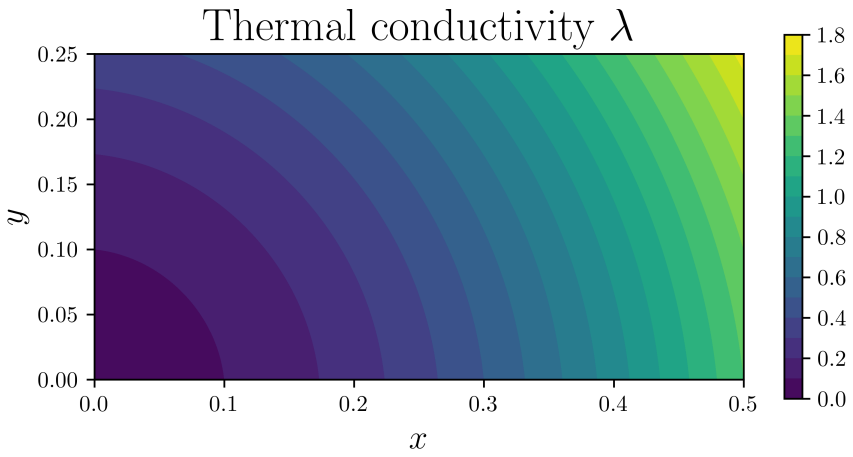
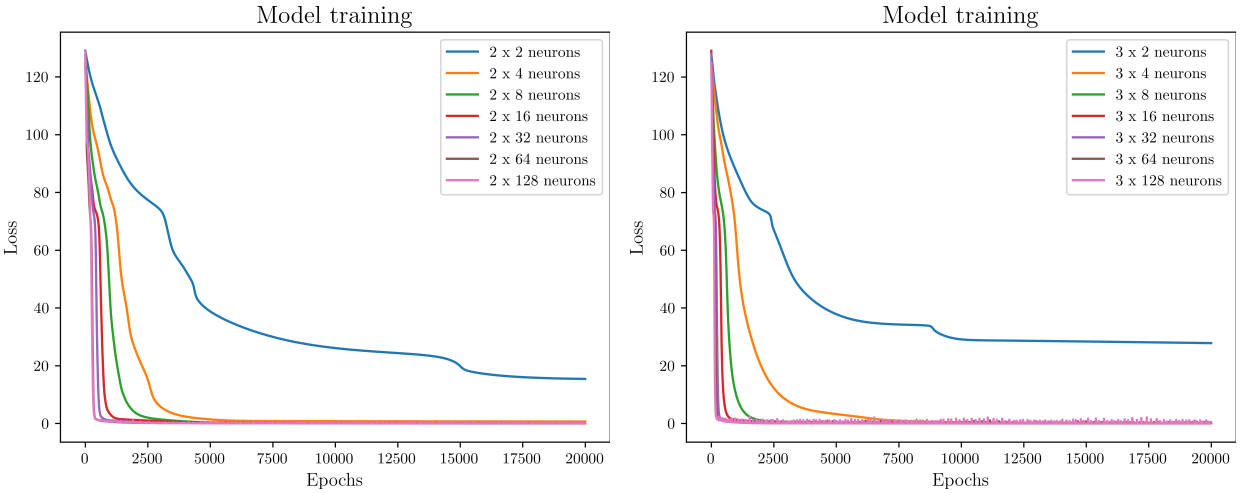


Figure 3.1: Thermal conductivity parabolic function

In accordance with these BCs and also with the fact, that we do not consider any internal heat source $\overline{Q}$, the loss function for this particular problem is expressed as:

$$
\begin{aligned}
l &= c_{PDE} l_{PDE} + c_{BC} l_{\Gamma_D} + c_{BC} l_{\Gamma_N}, \\
l_{PDE} &= \frac{1}{N_{PDE}} \sum_{i=1}^{N_{PDE}} \left(\frac{\partial}{\partial x_i} \left(-\lambda_i \frac{T_i}{\partial x_i} \right) + \frac{\partial}{\partial y_i} \left(-\lambda_i \frac{T_i}{\partial y_i} \right) \right)^2, \quad \boldsymbol{x} \in \Omega, \\
l_{\Gamma_D} &= \frac{1}{N_D} \sum_{i=1}^{N_D} \left(T_i - \overline{T}_i \right)^2, \quad \boldsymbol{x} \in \Gamma_D, \\
l_{\Gamma_N} &= \frac{1}{N_N} \sum_{i=1}^{N_N} \left(\boldsymbol{n}_i^\mathsf{T} \boldsymbol{q}_i - \overline{q}_i \right)^2, \quad \boldsymbol{x} \in \Gamma_N, \\
c_{PDE} &= 1, \\
c_{BC} &= 1.
\end{aligned}
\tag{3.6}
$$

In order to find an approximate solution, we analyze the multiple model architectures for a better understanding of the relations between the model capabilities and its weights. We test the model architecture with 2 and 3 hidden layers with the number of neurons varying from 2 to 128. For every part of the loss function l_{PDE}, l_{Γ_D}, l_{Γ_N} separate datasets of dimensions $N_{PDE} = 5000$, $N_D = 100$, and $N_N = 200$ are created according to the rules described above. The other hyperparameters of the model are set as follows: We train the model for 20000 epochs with batch sizes equal to the sizes of particular datasets, and we use the Adam optimizer with the learning rate $\eta = 0.001$. As an activation function for each layer hyperbolic tangent is used.



(a) Models with 2 hidden layers

(b) Models with 3 hidden layers

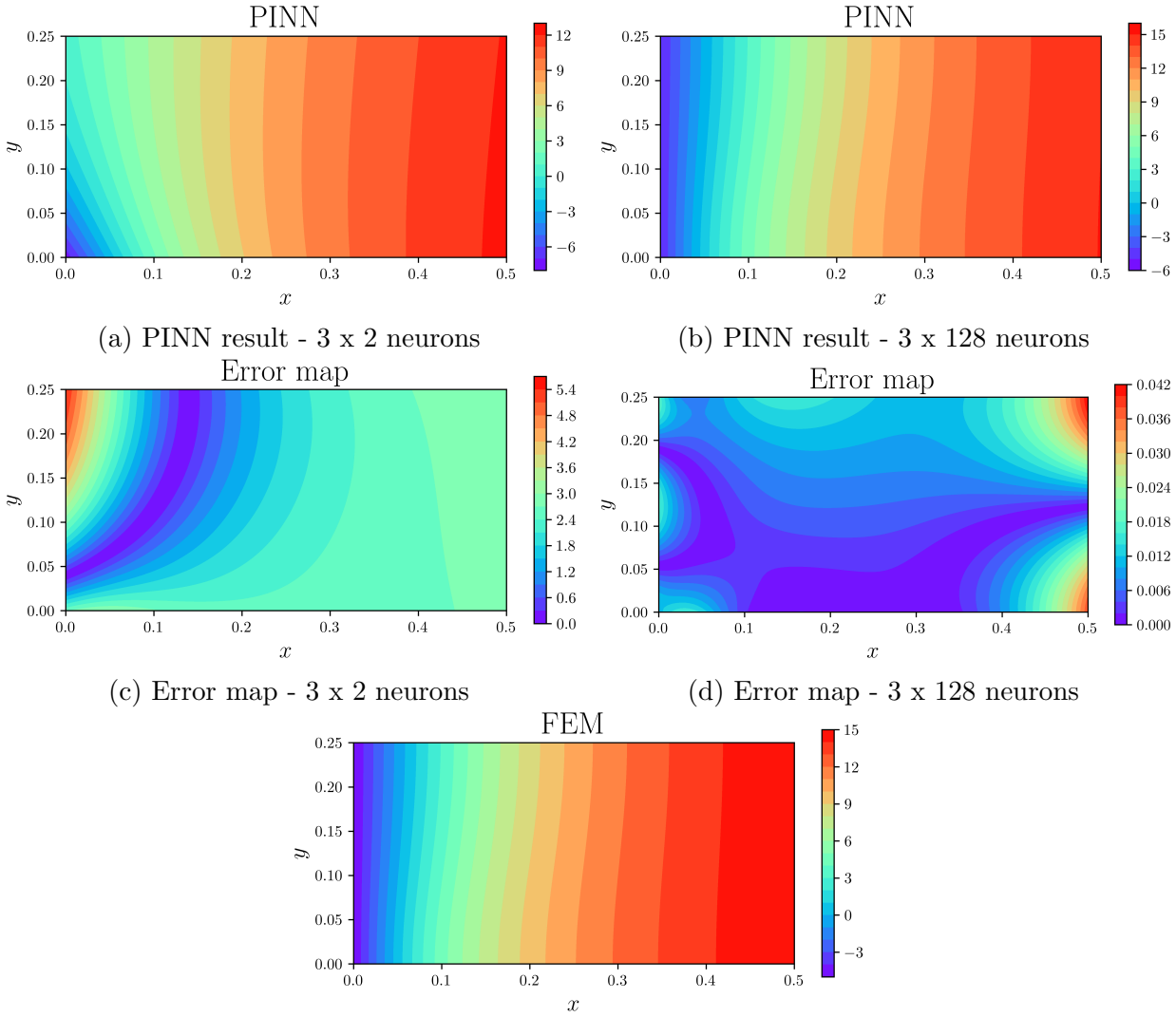
Figure 3.2: Evolution of training loss as a function of epoch - Example 1

The results in the figure 3.2 and the table 3.1 show, that even for the architecture with 2 layers and 4 neurons, the model is capable to minimize the loss function to the point, where the mean absolute error between the model prediction and the FEM is only $0.071396°C$, which is a decent result for model with only 37 parameters. The best results are achieved for the models with 128 neurons since they have the highest number of trainable parameters. Models with more parameters in general have faster convergence and better results, which implies that the usage of smaller models capable of similar results is not necessarily beneficial. Smaller models indeed take less time to process the calculation of one epoch, but if the convergence is significantly slower, this benefit is negligible, which is our case. The convergence for the 128-neuron models is more beneficial than the time spent for one epoch favoring models with fewer neurons. Although the results are evaluated only for one run of training and the algorithm is stochastic, we further use the architecture with 128 neurons.

Table 3.1: Summary of Results.

Architecture	N_w	Loss	MAE	MSE	STD	MAX	MIN
2 x 2 neurons	15	15.405536	0.863279	0.988423	0.493125	2.014402	3.35176e-04
2 x 4 neurons	37	0.666370	0.071396	0.007065	0.044361	0.445415	6.51665e-05
2 x 8 neurons	105	0.045101	0.017088	0.000505	0.014611	0.144072	1.09409e-05
2 x 16 neurons	337	0.029583	0.019831	0.000535	0.011922	0.076662	3.18487e-07
2 x 32 neurons	1185	0.013648	0.009718	0.000143	0.006997	0.043492	8.56302e-06
2 x 64 neurons	4417	0.068095	0.008944	0.000153	0.008518	0.057491	4.33780e-06
2 x 128 neurons	17025	0.018064	0.009611	0.000141	0.006948	0.045999	4.73365e-06
3 x 2 neurons	21	27.818975	2.099816	5.172948	0.873911	5.458745	9.56127e-04
3 x 4 neurons	57	0.199080	0.020547	0.000760	0.018382	0.164325	9.26590e-06
3 x 8 neurons	177	0.050924	0.035022	0.001685	0.021421	0.170085	4.93061e-06
3 x 16 neurons	609	0.043996	0.013871	0.000347	0.012421	0.153034	2.03154e-06
3 x 32 neurons	2241	0.082613	0.009133	0.000169	0.009227	0.045073	1.28970e-06
3 x 64 neurons	8577	0.120965	0.018484	0.000429	0.009365	0.055541	9.59330e-06
3 x 128 neurons	33537	0.007554	0.007968	0.000110	0.006784	0.041994	1.22507e-06

The Architecture describes how many layers and neurons are used in the particular model. For example, 2 x 4 neurons means 2 hidden with 4 neurons each. The symbol N_w indicates how many weights (trainable parameters) contain each model, the term Loss is the value of the loss function after 20000 epochs and the remaining columns contain statistics calculated on the difference between the model prediction and the FEM. The differences are calculated directly in the temperature degrees $°C$, MAE is the mean absolute error, MSE is the mean squared error, STD is the standard deviation, and MAX and MIN state the maximum and minimum values of error achieved. The results are evaluated only for one run of training for each architecture.



(a) PINN result - 3 x 2 neurons

(b) PINN result - 3 x 128 neurons

(c) Error map - 3 x 2 neurons

(d) Error map - 3 x 128 neurons

(e) FEM result

Figure 3.3: Temperature for error field computed for thermal conductivity defined as a paraboloid function

Visualizations of the results for the worst model (3 x 2 neurons) and the best model (3 x 128 neurons).

3.1.2 Example 2 - Thermal conductivity defined as a wave function

To discuss more the limits, we consider a similar example as in the previous subsection with a more complicated thermal conductivity function:

$$\lambda(\boldsymbol{x}) = \frac{1}{15}\left(\left|5 + 3x^2 + 2y - yx - 5y\sin\left(\frac{x}{0.075}\right) + 10x\cos(10y)\right| + 10\right). \tag{3.7}$$

As you can see in the figure 3.4, it is a function that models wave shape. The modeled

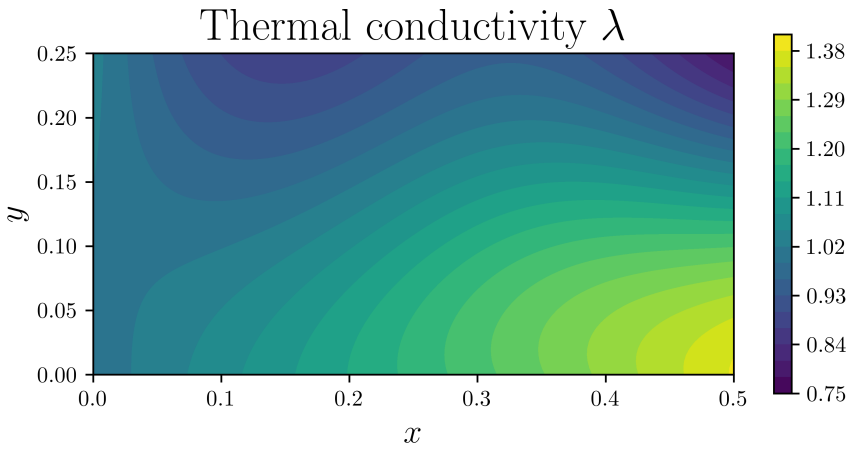
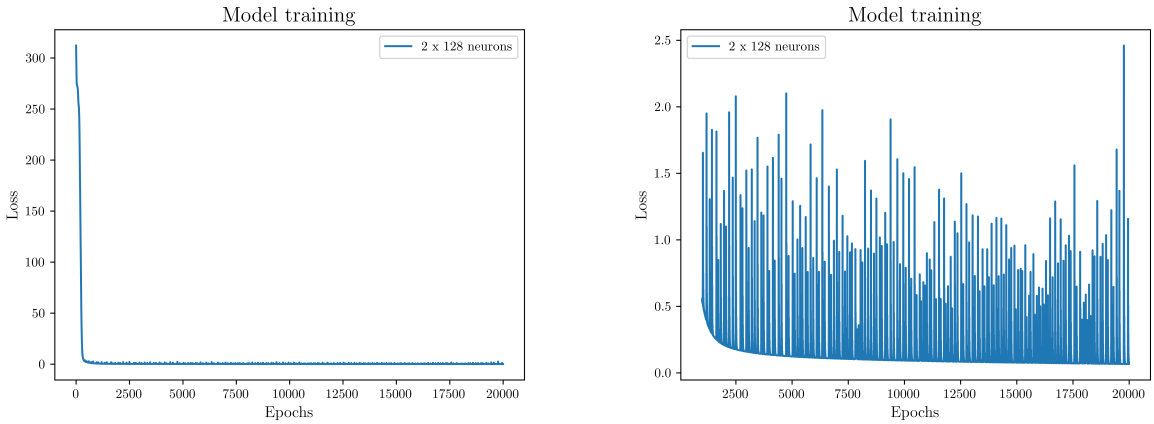


Figure 3.4: Thermal conductivity defined as a wave function.

problem has similar boundary conditions as before:

$$\begin{aligned} \overline{T}(0, y) &= 20, \quad y \in [0, 0.25], \\ \overline{T}(0.5, y) &= -15, \quad y \in [0, 0.25], \\ \overline{q}(x, 0) &= 0, \quad x \in [0, 0.5], \\ \overline{q}(x, 0.25) &= 0, \quad x \in [0, 0.5]. \end{aligned} \tag{3.8}$$

We again train the model for 20000 epochs, we use the Adam optimizer with the learning rate $\eta = 0.001$, batch sizes are equal to the sizes of particular datasets, and the loss function is defined by the equation 3.6. The used model contains 2 hidden 128-neuron layers with the hyperbolic tangent as an activation function. The table 3.2 contains computed



(a) Loss function (b) Loss function - zoom after 1000 epochs

Figure 3.5: The evolution of training loss as a function of epoch - Example 2
We can see the loss rapidly decreases from the beginning and then slows down with oscillations until the epoch 20000.

results. The model is again able to minimize the loss function successfully. However,

Table 3.2: Summary of Results.

Architecture	N_w	loss	MAE	MSE	STD	MAX	MIN
2 x 128 neurons	17025	0.068722	0.035624	0.002550	0.035797	0.189495	6.49811e-07

compared to a similar example with parabolic thermal conductivity 3.1.1, the value of error is one order of magnitude higher, even though the thermal conductivity in this example varies from 0.78 to 1.38 $\frac{W}{m°K}$ only. It is probably caused by the noncontinuous derivatives of the wave function since the formula contains the absolute value. Therefore, in the next sub-subsection, we modify the conductivity function to obtain larger extremes, which results in sharper gradients and more complicated temperature distribution.

Sharper wave function

To test the limits of the following algorithm, we take the same boundary condition as in the previous example. However, we modify the thermal conductivity function according to the following shape:

$$\lambda(\boldsymbol{x}) = \frac{1}{15}\left(\left|5 + 50x^2 + 100y - 50yx - 100y\sin\left(\frac{x}{0.075}\right) + 500x\cos(10y)\right| + 10\right). \quad (3.9)$$

The modifications cause fluctuations in thermal conductivity values varying from 0.67 to 18.5 $\frac{W}{m°K}$. This function changes the values of the thermal conductivity more radically than the previous one, resulting in a more complicated temperature distribution with sharper gradients as a response. We use the Adam optimizer and we train the models for 10000 epochs.

The following tables summarize the achieved values of the loss function at the end of the training process as a function of the learning rate. The tested architectures contain 2 and 3 hidden layers with 128 neurons.

Table 3.3: Results for various learning rates. The tables above show the loss value and the mean absolute error calculated for the prediction and the finite element method.

Learning rate	1e-06	5e-06	1e-05	5e-05	0.0001	0.0005	0.001	0.005	0.01
Loss - 2 x 128	311.278	307.165	306.207	306.015	306.029	305.882	305.928	305.778	305.859
MAE - 2 x 128	8.891	9.488	9.850	9.855	9.856	9.853	9.854	9.852	9.852
Loss value - 3 x 128	310.459	306.335	306.065	305.979	305.964	305.891	305.799	305.931	306.186
MAE - 3 x 128	8.962	9.818	9.857	9.855	9.855	9.853	9.851	9.854	9.858

From the table 3.3 we can see that we have not successfully trained the model for any learning rate. Every time the training progress freezes around the loss value of 300, which corresponds to the mean absolute error between model prediction and FEM approximately

equal to 10. Even if more complex models with more hidden layers and parameters are employed, the results remain similar. This is due to a well-known shortcoming of PINNs, namely that PINNs cannot approximate functions with sharp gradients and strong non-linearities. A common practice to resolve this issue is to apply the technique called distributed physics-informed neural network (DPINN), where we divide the entire domain into smaller subdomains, and for each, we apply a separate neural network approximating the solution on the subdomain only. And similarly to the finite volume method, on the boundaries between each subdomain, conditions of continuity are prescribed, see [18], and [19]. However, the implementation of DPINN is not part of this thesis.

3.2 PINN as a surrogate model

In this section, we further elaborate on the PINN concept from the previous section so that the trained model can predict multiple solutions based on varying boundary conditions or thermal conductivity. For the evaluation as well as for the training procedure, we again use FEM. To discretize the domain defined by equation 3.2 we use coarse finite element mesh with 50 nodes in the x direction and 25 nodes in the y direction only. The mesh contains 1176 bilinear rectangular elements. The number of nodes in the domain and on the boundaries are $N_{PDE} = 1250$, $N_D = 100$, and $N_N = 50$.

To serve as a surrogate model, we change the shape of the network input as follows: x and y coordinates, and the parameters determining the particular solution like the prescribed temperature on the boundaries, or some parameters influencing the thermal conductivity function.

3.2.1 Example 3 - Surrogate model with changing Dirichlet BC

As a first example, we formulate a model in a way so that it can predict the solution based on the prescribed temperature on the Dirichlet boundary Γ_D. The prescribed temperature varies from $-20°K$ to $20°K$, while the heat fluxes in the normal direction on the Neumann boundary Γ_N remain zero:

$$\begin{aligned} \overline{T}(0, y) &= T_1, \quad y \in [0, 0.25], T_1 \in [-20, 20], \\ \overline{T}(0.5, y) &= T_2, \quad y \in [0, 0.25], T_2 \in [-20, 20], \\ \overline{q}(x, 0) &= 0, \quad x \in [0, 0.5], \\ \overline{q}(x, 0.25) &= 0, \quad x \in [0, 0.5]. \end{aligned} \tag{3.10}$$

For such a case, the inputs to the model are the coordinates x and y, and the prescribed temperatures T_1 and T_2. The thermal conductivity on the domain is defined by the

equation 3.15. For the training dataset, we again use the mesh nodes specified above and a certain number of prescribed temperature combinations. First, we generate the prescribed temperature combinations, and for each combination, every node in the mesh is used to create one data sample as (x, y, T_1, T_2). The total numbers of points for particular datasets are then $N^T_{PDE} = N_{PDE}N_T$ for the evaluation of PDE loss in the entire domain l_{PDE}, $N^T_D = N_D N_T$ for the evaluation of boundary condition loss on the Dirichlet boundary l_{Γ_D}, and $N^T_N = N_N N_T$ for the boundary condition loss on the Neumann boundary l_{Γ_N}, where N_T is the number of the prescribed temperature combinations. The loss function is composed the same according to equation 2.63:

$$
\begin{aligned}
l &= c_{PDE} l_{PDE} + c_{BC} l_{\Gamma_D} + c_{BC} l_{\Gamma_N}, \\
l_{PDE} &= \frac{1}{N^T_{PDE}} \sum_{i=1}^{N^T_{PDE}} \left(\frac{\partial}{\partial x_i} \left(-\lambda_i \frac{T_i}{\partial x_i} \right) + \frac{\partial}{\partial y_i} \left(-\lambda_i \frac{T_i}{\partial y_i} \right) \right)^2, \quad \boldsymbol{x} \in \Omega, \\
l_{\Gamma_D} &= \frac{1}{N^T_D} \sum_{i=1}^{N^T_D} \left(T_i - \overline{T}_i \right)^2, \quad \boldsymbol{x} \in \Gamma_D, \\
l_{\Gamma_N} &= \frac{1}{N^T_N} \sum_{i=1}^{N^T_N} \left(\boldsymbol{n}_i^\intercal \boldsymbol{q}_i - \overline{q}_i \right)^2, \quad \boldsymbol{x} \in \Gamma_N, \\
c_{PDE} &= 1, \\
c_{BC} &= 1.
\end{aligned}
\tag{3.11}
$$

The only difference is in the count of points used to calculate the error. Automatic differentiation determines the derivatives exclusively with respect to x and y, omitting the derivatives with respect to the input parameters T_1 and T_2 since they do not contribute to the loss function

For the generation of prescribed temperature combinations, we test three statistical methods for various numbers of samples. The first method is random generation from the uniform distribution, the second method is generation using Sobol´s sequences and the third one is the Latin hypercube sampling. We use the architecture with two hidden layers with 128 neurons. As the optimizer, Adam with the learning rate $\eta = 0.001$ is used. Batch sizes for particular datasets are equal to 5000 for l_{PDE}, 100 for l_{Γ_D}, and 200 for l_{Γ_N}, and the individual data in each dataset are randomly scrambled to more accurately reflect the true gradient, as was explained in the subsection 2.3.3. The models are trained for 1000 epochs with a validation dataset (described below). If the validation loss value after a particular epoch is smaller than 0.005, the model is considered trained and the training process is stopped. The following tables display the results for all three sampling methods.

Table 3.4: Random uniform.

N_T	Epochs	Train loss	Val loss	MAE	MSE	STD	MAX	MIN
1	1000	3.774943	34.063988	4.654212	34.063988	3.521690	21.325756	1.335e-05
2	1000	0.509748	54.040901	5.915277	54.040901	4.364676	22.640238	1.049e-05
5	1000	0.890595	99.277161	7.283341	99.277161	6.799272	32.910744	1.478e-05
10	1000	1.079337	79.621719	6.305774	79.621719	6.313394	36.031082	3.815e-06
20	1000	1.239672	23.900618	3.684005	23.900618	3.213833	21.085701	1.335e-05
30	1000	0.830663	7.113817	1.891686	7.113817	1.880250	14.574469	4.947e-06
50	1000	0.516627	0.647840	0.520723	0.647840	0.613749	5.415449	2.861e-06
100	1000	0.332348	0.105506	0.198922	0.105506	0.256779	2.770552	9.537e-07
200	1000	0.187373	0.012604	0.074679	0.012604	0.083830	1.331676	0.000e+00
500	475	0.189282	0.005157	0.048719	0.005157	0.052762	0.704285	0.000e+00

Table 3.5: Sobol´s sequences.

N_T	Epochs	Train loss	Val loss	MAE	MSE	STD	MAX	MIN
1	1000	0.879237	85.339752	7.316768	85.339752	5.639561	25.567598	9.894e-06
2	1000	0.810116	38.564598	4.818148	38.564598	3.917914	26.837009	4.292e-05
5	1000	3.599277	73.592392	6.260838	73.592392	5.864666	46.895874	1.907e-06
10	1000	1.692223	20.841707	3.657211	20.841707	2.732492	14.581657	9.537e-07
20	1000	1.552725	3.275407	1.230261	3.275407	1.327352	10.242945	0.000e+00
30	1000	0.954964	0.510728	0.465147	0.510728	0.542555	4.593925	1.669e-06
50	1000	0.520235	0.134904	0.253009	0.134904	0.266252	2.279523	4.768e-07
100	1000	0.306546	0.019302	0.094918	0.019302	0.101452	0.972975	0.000e+00
200	534	0.270542	0.005906	0.047591	0.005906	0.060341	0.639710	0.000e+00
500	219	0.235817	0.005223	0.043194	0.005223	0.057943	0.555103	0.000e+00

Table 3.6: Latin hypercube sampling.

N_T	Epochs	Train loss	Val loss	MAE	MSE	STD	MAX	MIN
1	1000	1.771804	175.738617	10.685577	175.738617	7.845831	36.227482	8.678e-05
2	1000	0.810276	35.806599	4.782364	35.806599	3.596609	26.645348	5.531e-05
5	1000	1.876327	152.403137	9.617827	152.403137	7.739543	43.682590	2.432e-05
10	1000	0.791286	10.411078	2.328101	10.411078	2.234061	13.039709	5.722e-06
20	1000	1.213744	2.923914	1.209685	2.923914	1.208543	10.267759	2.384e-06
30	1000	0.741756	0.152917	0.265530	0.152917	0.287073	2.000020	2.384e-07
50	1000	0.348275	0.032570	0.104262	0.032570	0.147308	1.277060	0.000e+00
100	1000	0.234512	0.009830	0.066331	0.009830	0.073687	0.707731	4.321e-07
200	955	0.143667	0.003456	0.042607	0.003456	0.040500	0.511026	0.000e+00
500	452	0.117360	0.002524	0.033812	0.002524	0.037156	0.651495	3.576e-07

The N_T in tables 3.4, 3.5, and 3.6 is the number of prescribed temperature combinations used for model training, Train loss is the value of the training loss after the last epoch, and Val loss is the value of the validation loss after the last epoch calculated as the mean square error between model prediction and validation dataset of 250 temperature combinations computed by FEM. The remaining columns have the same meaning as those in the previous tables and are calculated also on the validation dataset after the training. The 250 validation combinations were generated with the Latin hypercube sampling. The combinations are provided in the GitHub repository [17].

As expected, the random uniform distribution achieves the worst results of all compared methods. The Latin hypercube sampling and Sobol´s sequence exhibit similar values. However, for the results utilizing the Latin hypercube sampling method, we obtained the best results, especially for the higher number of temperature combinations. Sobol´s sequencing achieves better results for several combinations varying from 1 to 10. However, none of these values can be considered satisfiable for a surrogate model. The smallest number of combinations for which we obtain the MAE value smaller than $1°C$ is equal to 30, for both Latin hypercube sampling and Sobol´s sequences. Although the results are evaluated only for one run we decide to use the Latin hypercube sampling in the next examples.

On the figure 3.6 we can see the typical progress of particular loss functions during the training process. From the beginning, the highest value belongs to the l_{Γ_D}, and the optimizer practically minimizes only this loss itself. As the loss on the Dirichlet boundary decreases, the l_{PDE} and l_{Γ_D} start to play a role, and all three losses are minimized equally until the end of the training process.

To visualize the performance of the models, we take the validation combinations and calculate the mean absolute error between the model prediction and FEM for all 250

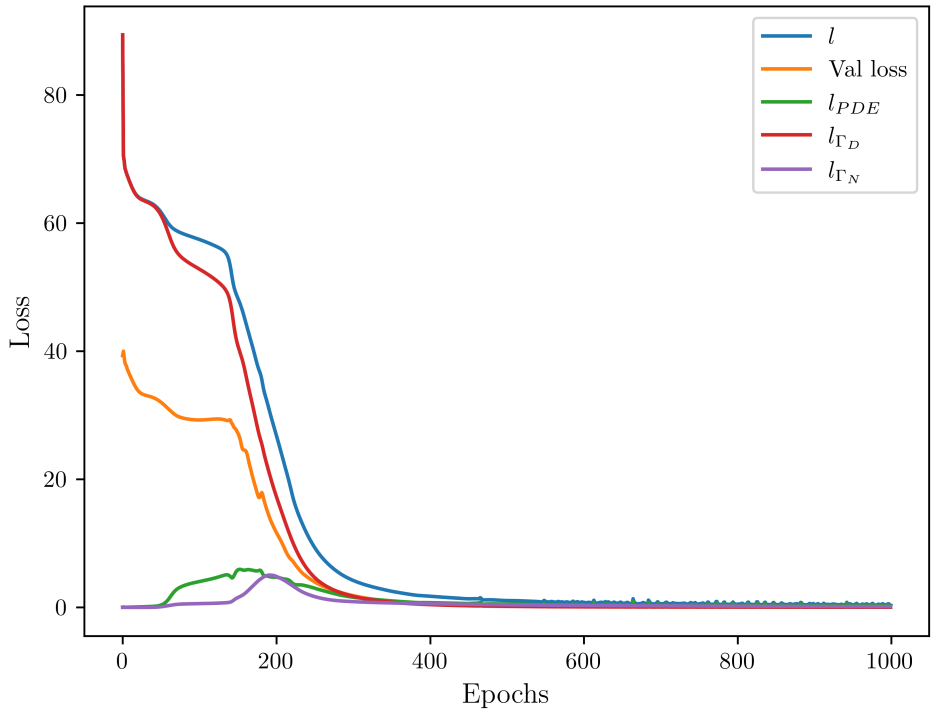
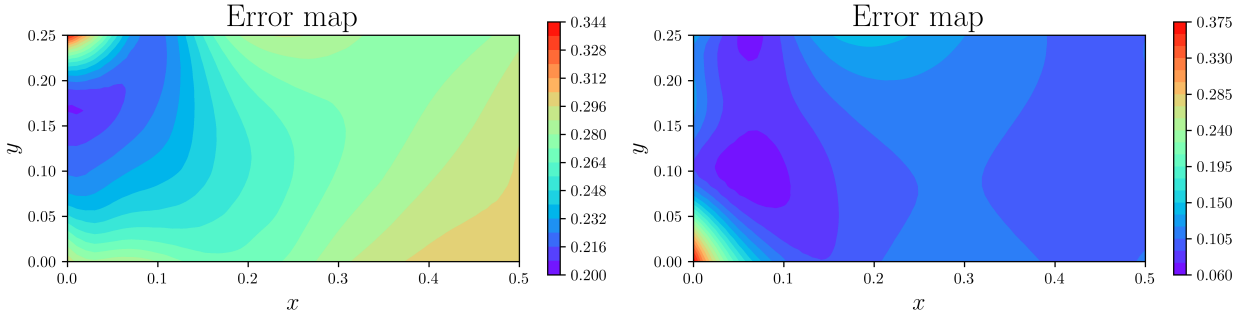


Figure 3.6: The loss value inspection. This figure displays the evolution of the loss function and its parts during training of the Latin hypercube sampling for $N_T = 50$ case.

combinations resulting in the average error maps in figure 3.7.



(a) LHS ($N_T = 30$) (b) LHS ($N_T = 500$)

Figure 3.7: Average error maps.

Influence of additional data

This subsection discusses the influence of the additional data on the training. We expand the previous example by adding the training dataset of 100 FEM simulations. To incorporate such a dataset in the loss function, additional loss l_{data} along with its scaling

coefficient c_{data} (which is again considered as 1 is introduced:

$$
\begin{aligned}
l &= c_{PDE} l_{PDE} + c_{BC} l_{\Gamma_D} + c_{BC} l_{\Gamma_N} + c_{data} l_{data}, \\
l_{PDE} &= \frac{1}{N_{PDE}^T} \sum_{i=1}^{N_{PDE}^T} \left(\frac{\partial}{\partial x_i} \left(-\lambda_i \frac{T_i}{\partial x_i} \right) + \frac{\partial}{\partial y_i} \left(-\lambda_i \frac{T_i}{\partial y_i} \right) \right)^2, \quad \boldsymbol{x} \in \Omega, \\
l_{\Gamma_D} &= \frac{1}{N_D^T} \sum_{i=1}^{N_D^T} \left(T_i - \overline{T}_i \right)^2, \quad \boldsymbol{x} \in \Gamma_D, \\
l_{\Gamma_N} &= \frac{1}{N_N^T} \sum_{i=1}^{N_N^T} \left(\boldsymbol{n}_i^\mathsf{T} \boldsymbol{q}_i - \overline{q}_i \right)^2, \quad \boldsymbol{x} \in \Gamma_N, \\
l_{data} &= \frac{1}{N_{data}} \sum_{i=1}^{N_{data}} \left(T_i - T_{d,i} \right)^2, \\
c_{PDE} &= 1, \\
c_{BC} &= 1, \\
c_{Data} &= 1,
\end{aligned}
\tag{3.12}
$$

where N_{data} is the number of points in the additional dataset and $T_{d,i}$ is the particular temperature in the dataset. The batch size for the additional dataset is set to 1250. The remaining hyperparameters are the same as in the previous example. The generation of the prescribed temperature combinations is done via the LHS method.

Table 3.7: Latin hypercube sampling with additional data

N_T	Epochs	Train loss	Val loss	MAE	MSE	STD	MAX	MIN
1	1000	0.942911	66.412666	6.406090	66.412666	5.037328	23.897648	1.335e-05
2	1000	3.140736	55.363750	6.006547	55.363750	4.391486	22.222092	2.384e-05
5	1000	2.920565	123.502281	7.826291	123.502281	7.889961	46.502453	9.537e-06
10	1000	1.023740	30.931248	4.007959	30.931248	3.855841	24.351902	2.718e-05
20	1000	1.599458	1.851657	0.896976	1.851657	1.023275	8.410130	9.537e-06
30	1000	1.651856	0.461668	0.454308	0.461668	0.505244	3.651707	3.815e-06
50	1000	0.502627	0.115929	0.233377	0.115929	0.247920	1.792378	0.000e+00
100	1000	0.491345	0.051310	0.147572	0.051310	0.171850	3.571630	3.576e-07
200	1000	0.272304	0.007897	0.060866	0.007897	0.064751	0.681201	0.000e+00
500	407	0.246598	0.004431	0.051792	0.004431	0.041816	0.620075	0.000e+00

The results in the table 3.7 imply that the additional data of 100 FEM simulations with c_{data} do not improve the results at all. The evolution of loss with the number of combinations $N_T = 50$ is depicted to investigate such behavior from the figure, we can

see that the l_{data} is minimized in the beginning and further contributes negligibly to the overall value of the loss. Therefore, the optimization is driven mainly by the other types of losses.

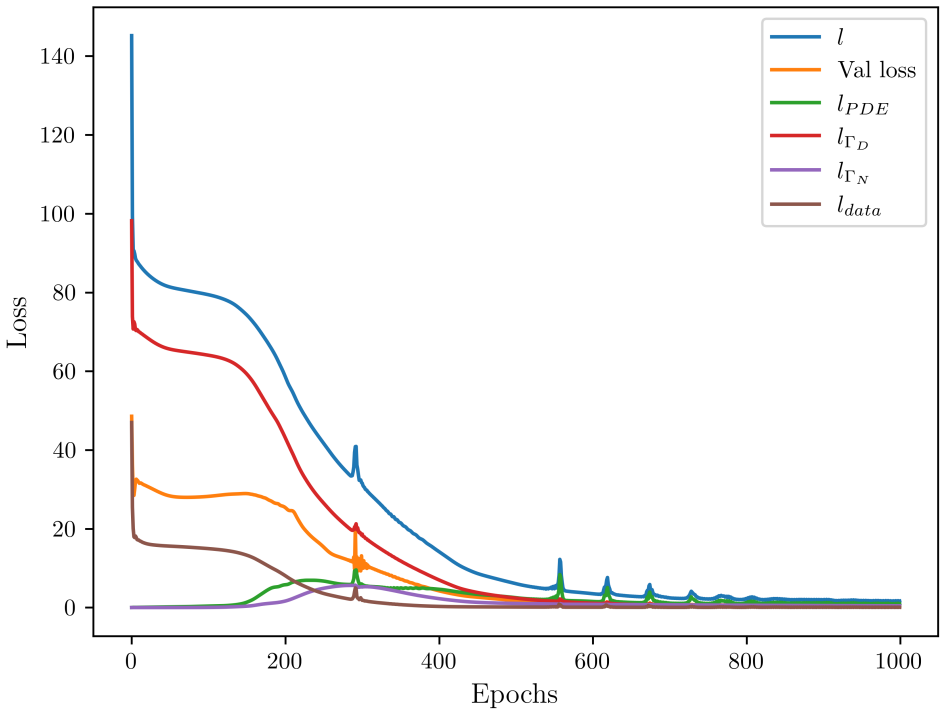


Figure 3.8: The evolution of loss computed for the example with additional data.

Comparison with the polynomial chaos-based surrogate models

This study comparatively evaluates the classical approach to surrogate model construction and the proposed methodology based on the PINN concept. There are various types of surrogate models, each characterized by distinct advantages and tailored to specific application domains. This comparison focuses on polynomial chaos expansions (PCE), prominent in probabilistic uncertainty quantification, as the most suitable candidate for benchmarking.

With a certain degree of simplification, the PCE-based approximation $\boldsymbol{T}_{PCE}(\boldsymbol{\xi}(\omega))$ of the model response $\boldsymbol{T}(\boldsymbol{\xi}(\omega))$ be written as

$$\boldsymbol{T}_{PCE}(\boldsymbol{\xi}(\omega)) = \sum_{\alpha \in A} \boldsymbol{T}_{\alpha} P_{\alpha}(\boldsymbol{\xi}(\omega)), \tag{3.13}$$

where $\boldsymbol{T}_{\alpha}$ is a vector of PC coefficients, $P_{\alpha}(\boldsymbol{\xi}(\omega))$ represents the multivariate Legendre polynomials, and the index set A is a finite set of non-negative integer sequences with

only finitely many non-zero terms, i.e., multi-indices. For this specific example, $\boldsymbol{\xi}(\omega)$ is the set of four independent standard uniform variables representing the variability in the model inputs, i.e. spatial coordinates x, y, and the prescribed boundary temperatures T_1, T_2. The number of PC terms $|A|$, representing the model complexity, is given as

$$|A| = \frac{(n_p + n_s)!}{n_p! n_s!}, \tag{3.14}$$

where n_p is the polynomial degree and n_s is the number of input variables, here $n_s = 4$. The estimation of PC coefficients $\boldsymbol{T}_\alpha$ is performed via least-squares minimization, with sampling points selected using Latin Hypercube Sampling. For further details, readers are directed to the journal paper that discusses these issues, see [20].

table 3.8 presents a comparison of surrogate models, with metrics and error values computed using a common validation dataset comprising 250 samples. One particular PINN model has been selected for this comparison, see the last row of table 3.6, $N_T = 500$. To enable straightforward visual matching, the obtained values are duplicated in the first row of table 3.8. For the surrogate model based on polynomial chaos expansion, selecting the polynomial degree n_p, in addition to the size of the training dataset N_T, determines the total number of unknowns $|A|$. Here the the polynomial degree varies from 1 to 6. As evidenced by the presented results, the values obtained from both methodologies are similar, indicating that the PINN concept merits further study.

Table 3.8: Comparison of surrogate models.

N_T	Model description			MAE	MSE	STD	MAX
500	PINN			0.03381	0.00252	0.03716	0.65150
500	PCE	$n_p = 1$	$\|A\| = 5$	3.10339	20.15579	3.25069	19.02546
500	PCE	$n_p = 2$	$\|A\| = 15$	0.98853	2.09106	1.05751	6.91909
500	PCE	$n_p = 3$	$\|A\| = 35$	0.26910	0.17301	0.31781	2.19298
500	PCE	$n_p = 4$	$\|A\| = 70$	0.08135	0.02353	0.13032	1.23824
500	PCE	$n_p = 5$	$\|A\| = 126$	0.04205	0.00640	0.06819	0.61712
500	PCE	$n_p = 6$	$\|A\| = 210$	0.02138	0.00176	0.03611	0.34305

3.2.2 Example 4 - Surrogate model constructed for changing Dirichlet BC and thermal conductivity

We further test the PINN concept to serve as a surrogate model, by adding two other parameters affecting the thermal conductivity field. This means that the model should predict the solution as a function of spatial coordinates by considering prescribed boundary

temperatures and the thermal conductivity function as inputs. We consider the boundary conditions defined by equation 3.10, and the thermal conductivity field is defined by the following equation:

$$\lambda(\boldsymbol{x}) = p_1 5x^2 + p_2 5y^2 + xy + 0.05, \quad p_1, p_2 \in [0.05, 2]. \tag{3.15}$$

The parameters p_1 and p_2 modify the original equation 3.15 and can take values from 0.05 to 2. Practically speaking the limits of the the thermal conductivity on the domain values can now vary from 0.05 to 3.3 with amplitudes in each point displayed in the figure 3.9 The input to the neural network is a vector of 6 with the following compo-

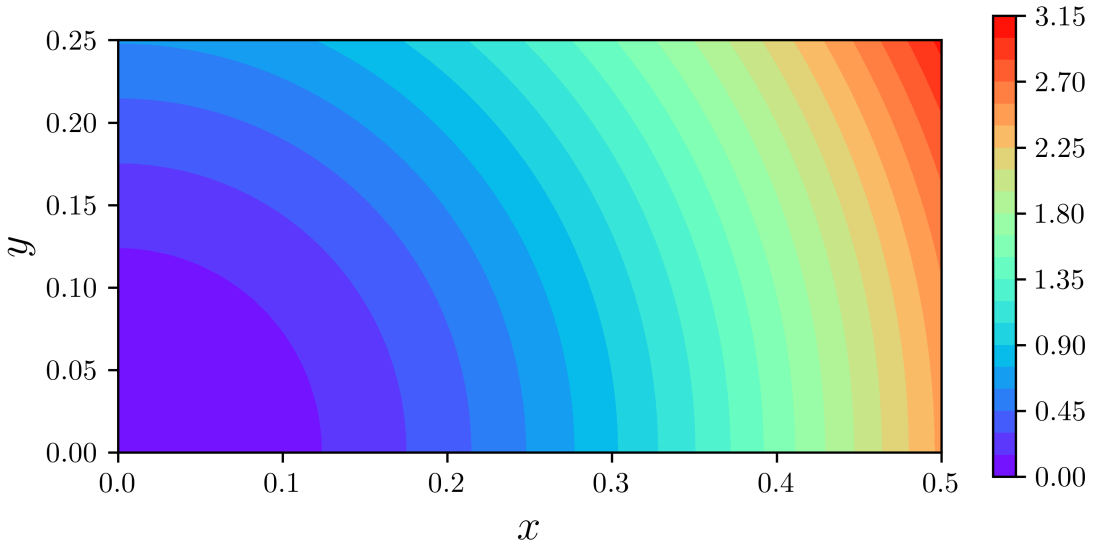


Figure 3.9: Amplitude of the thermal conductivity

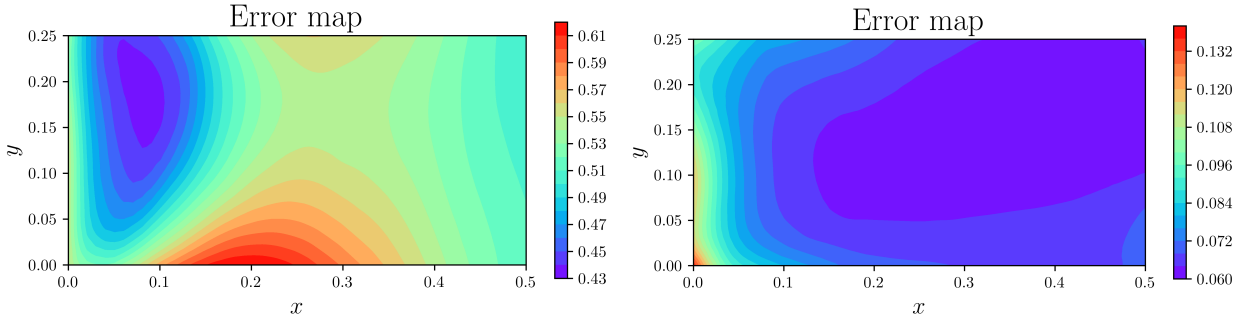
nents $(x, y, T_1, T_2, p_1, p_2)$. The loss function is calculated the same way as in subsection 3.2.1, see equation 3.10. The datasets for training are created from the given number of combinations - T_1, T_2, p_1, p_2 generated by Latin hypercube sampling.

Similarly, we train the model for 1000 epochs with a validation dataset of 250 combinations of temperatures and parameters p_1, and p_2. If the validation loss after a particular epoch, calculated as the mean square between the model prediction and FEM, is smaller than 0.005, the training process i stopped. The optimizer is Adam with the learning rate $\eta = 0.001$. The batch sizes used for the calculation of each loss term are set as follows: l_{PDE} - 5000, l_{Γ_D} - 100, and l_{Γ_N} - 200.

Table 3.9: Error metric computed for different numbers of N_T.

N_T	Epochs	Train loss	Val loss	MAE	MSE	STD	MAX	MIN
1	1000	0.253600	45.412998	5.565204	45.412998	3.800197	23.624317	1.559e-04
2	1000	8.252272	42.727272	5.310050	42.727272	3.811907	18.368614	1.240e-05
5	1000	10.084234	174.884430	10.252326	174.884430	8.353098	44.148556	4.816e-05
10	1000	1.689702	83.350510	6.862163	83.350510	6.021730	39.621708	1.907e-06
20	1000	1.674614	17.475906	3.306230	17.475906	2.558270	14.063549	1.431e-05
30	1000	1.341758	7.573429	2.128058	7.573429	1.744935	12.191624	2.193e-05
50	1000	1.282806	2.448461	1.139659	2.448461	1.072212	11.289648	2.861e-06
100	1000	0.445469	0.609402	0.522769	0.609402	0.579753	4.526513	1.431e-06
200	1000	0.438859	0.126222	0.231569	0.126222	0.269439	2.155493	4.768e-07
500	1000	0.237113	0.009826	0.068405	0.009826	0.071743	1.028811	0.000e+00

From the result in the table 3.9 we can see that even for a number of combinations $N_T = 500$ the models do not achieve value of validation loss smaller than 0.005. In comparison to the table 3.6, the model performance is lower. This is expected since the LHS-filled space now has four dimensions rather than the two in the earlier example, and mostly the loss function is probably more difficult to approximate. Nevertheless, with the number of combinations of 100, 200, and 500, we obtain the values of MAE smaller than $1°C$. The following figure shows the average error maps for the $N_T = 100$ and $N_T = 500$.



(a) LHS ($N_T = 100$) (b) LHS ($N_T = 500$)

Figure 3.10: Average error maps

Chapter 4

Conclusion

In this thesis, we implemented the concept of a physics-informed neural network to approximate a solution to the heat conduction equation, both in the case of a single solution of a partial differential equation and in the case of multiple solutions, depending on the input parameters - the surrogate model. The solutions are found by minimizing the loss function with incorporated physical laws utilizing automatic differentiation.

For the scenario involving a single solution, we experimented with various architectures to establish a balance between model robustness and effectiveness. Our goal was to determine whether employing simple models could be advantageous due to their faster prediction and loss assessment capabilities. Almost all of the architectures were capable of finding satisfactory solutions. However, the fastest convergence was observed for the complex models with the most parameters. Therefore further experiments were carried out for these architectures. We demonstrated that a simple physical-informed neural network implemented in this thesis is capable of finding the solution for the gradually changing thermal conductivity field. However, if the thermal conductivity field is more complicated or the response field exhibits sharp gradients, the algorithm does not satisfactorily converge.

For the case of surrogate modeling, we successfully trained the models that can predict the solution based on boundary temperatures and thermal conductivity inside the domain as inputs.

Even though physics-informed neural networks have certain shortcomings and disadvantages, we conclude that further research can bring interesting results and reveal significant potential. Neural networks have proven their versatility across different fields. They can either combine architectures and pre-trained models to develop large, sophisticated systems for complex tasks, or use smaller models to efficiently perform simple tasks.

4.1 Future works

The work could be improved by conducting further investigations into how additional data impacts the accuracy of the surrogate model, as this aspect has only been briefly discussed. Additionally, we aim to enhance the numerical studies by incorporating internal heat sources, a wider variety of boundary conditions, and potentially more complex geometries.

We would also like to test the physical-informed neural network capabilities on non-stationary problems. It would be interesting to see the model behavior if the time derivative is incorporated into the loss function. Another potential experiment is to implement the distributed physic-informed neural network and observe its behavior.

Bibliography

[1] O. C. Zienkiewicz and R. L. Taylor, *The finite element method for solid and structural mechanics*. Elsevier, 2005.

[2] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[3] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature Reviews Physics*, vol. 3, no. 6, pp. 422–440, 2021.

[4] M. Raissi, P. Perdikaris, 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*, vol. 378, pp. 686–707, 2019.

[5] K. Bao, W. Yao, X. Zhang, W. Peng, and Y. Li, "A physics and data co-driven surrogate modeling approach for temperature field prediction on irregular geometric domain," *Structural and Multidisciplinary Optimization*, vol. 65, no. 10, p. 302, 2022.

[6] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis, "Deepxde: A deep learning library for solving differential equations," *SIAM review*, vol. 63, no. 1, pp. 208–228, 2021.

[7] Y. Wei, Q. Serra, G. Lubineau, and E. Florentin, "Coupling physics-informed neural networks and constitutive relation error concept to solve a parameter identification problem," *Computers & Structures*, vol. 283, p. 107 054, 2023.

[8] Y. Wang, J. Zhou, Q. Ren, Y. Li, and D. Su, "3-d steady heat conduction solver via deep learning," *IEEE Journal on Multiscale and Multiphysics Computational Techniques*, vol. 6, pp. 100–108, 2021.

[9] T. Würth, C. Krauß, C. Zimmerling, and L. Kärger, "Physics-informed neural networks for data-free surrogate modelling and engineering optimization–an example from composite manufacturing," *Materials & Design*, vol. 231, p. 112 034, 2023.

[10] M. Abadi, A. Agarwal, P. Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[11] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[12] R. Al-Rfou, G. Alain, A. Almahairi, *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv e-prints*, arXiv–1605, 2016.

[13] F. Chollet, *Deep learning v jazyku Python: knihovny Keras, Tensorflow*. Grada Publishing, 2019.

[14] *Basic classification: Classify images of clothing — TensorFlow Core — tensorflow.org*, https://www.tensorflow.org/tutorials/keras/classification, [Accessed 11-11-2024].

[15] *Basic regression: Predict fuel efficiency — TensorFlow Core — tensorflow.org*, https://www.tensorflow.org/tutorials/keras/regression, [Accessed 11-11-2024].

[16] *Introduction to gradients and automatic differentiation — TensorFlow Core — tensorflow.org*, https://www.tensorflow.org/guide/autodiff, [Accessed 26-11-2024].

[17] *GitHub - sperlon/Deep-Learning-Based-Modeling-and-Simulation-of-Heat-Conduction — github.com*, https://github.com/sperlon/Deep-Learning-Based-Modeling-and-Simulation-of-Heat-Conduction, [Accessed 06-01-2025].

[18] V. Dwivedi, N. Parashar, and B. Srinivasan, "Distributed physics informed neural network for data-efficient solution to partial differential equations," *arXiv preprint arXiv:1907.08967*, 2019.

[19] S. Rout, V. Dwivedi, and B. Srinivasan, "Numerical approximation in cfd problems using physics informed machine learning," *arXiv preprint arXiv:2111.02987*, 2021.

[20] A. Kučerová, J. Sýkora, P. Havlásek, D. Jarušková, and M. Jirásek, "Efficient probabilistic multi-fidelity calibration of a damage-plastic model for confined concrete," *Computer Methods in Applied Mechanics and Engineering*, vol. 412, p. 116 099, 2023.