University of Kaiserslautern
Faculty of Mathematics

Master Thesis

Generalization of PINNs for Various Boundary and Initial Conditions

Violetta Schäfer

supervised by

Dr. Martin Bracke
Prof. Dr. René Pinnau
Dr. rer. nat. Martin Siggel

11 Januar, 2022

Abstract

Physics-informed neural networks (PINNs) are neural networks (NNs) which are used for solving partial differential equations (PDEs). They minimize a loss function which incorporates the given PDE by using automatic differentiation. To the best of our knowledge, PINNs are currently only used for solving one instance of a PDE problem, i.e. fixed initial and boundary conditions. We investigate if PINNs can be generalized such that they are able to solve PDEs for various instances of initial or boundary conditions, respectively.

To this end, we consider two cases: fixed initial and various boundary conditions, and various initial conditions given parameterized by six Fourier coefficients. As application example we choose the one-dimensional heat equation with Dirichlet boundary conditions. We show that for fixed initial conditions, PINNs are able to achieve good prediction accuracy on unseen data. For various initial conditions the generalization question is still open. Due to the gradient evaluations in each iteration the training process is exceedingly time-consuming. This makes the generalization of PINNs for many instances of the problem infeasible in practice. From the theoretical point of view, we assume that further training and optimization of code and the training dataset could answer the question in some future work.

Zusammenfassung

Physikalisch-unterrichtete neuronale Netze (PINNs) sind neuronale Netze (NNs), die verwendet werden um partielle Differentialgleichungen (PDEs) zu lösen. Sie minimieren eine Verlustfunktion, die die gegebene PDE einarbeitet, indem automatische Differentiation benutzt wird. Nach unserem besten Wissen werden PINNs zurzeit nur für das Lösen von einer Instanz einer PDE zu lösen, sprich feste Anfangs- und Randbedingungen. Wir untersuchen ob PINNs generalisiert werden können, sodass sie in der Lage sind PDEs mit mehreren Instanzen von Anfangs- bzw. Randbedingungen zu lösen.

Zu diesem Zweck betrachten wir zwei Fälle: feste Anfangs- und variierende Randbedingungen, und variierende Anfangsbedingungen gegeben parametrisiert durch sechs Fourierkoeffizienten. Als Anwendungsbeispiel wählen wir die eindimensionale Wärmeleitungsgleichung mit Dirichlet Randbedingungen. Wir zeigen, dass für feste Anfangsbedingungen PINNs dazu in der Lage sind eine gute Genauigkeit für Vorhersagen auf ungesehen Daten zu erzielen. Für variierende Anfangsbedingungen ist die Frage der Generalisierung noch offen. Durch die Auswertung des Gradienten in jeder Iteration ist der Trainingsprozess äußerst zeitaufwändig. Dies macht die Generalisierung von PINNs für viele Instanzen des Problems in der Praxis undurchführbar. Aus theoretischer Sicht gehen wir davon aus, dass weiteres Training und die Optimierung des Codes und des Trainingsdatensatzes die Frage in einer zukünftigen Arbeit beantworten könnten.

Contents

1 Introduction

In recent years, the topic of machine learning has been well-established not only in scientific research topics, but also in our everyday lives. Be it through several language assistants that are integrated in our everyday gadgets [1] or objects like cars or robots that are able to decide autonomously using artificial intelligence [2, 3]. With tremendous speed scientists of all reasearch areas are developing and enhancing machine learning techniques. And all have one in common. Data is the fundament.

Having a sufficient amount of data available is crucial for machine learning applications. It is common knowledge that neural networks only learn from data and it is intuitively clear that a lack of data causes the issue that we probably cannot rely on the prediction of the resulting model. Fortunately, we live in an age where the amount of data and computational resources are not a concern at least for applications based on e.g. image recognition or language processing. However, most real-world phenomena are not described by images or words but physical laws that can be formulated as partial differential equations (PDEs). Unfortunately, it holds that "in the course of analyzing complex physical, biological or engineering systems, the cost of data acquisition is prohibitive, and we are inevitably faced with the challenge of drawing conclusions and making decisions under partial information" [4].

Physics-informed neural networks (PINNs) are neural networks which compensise this lack of data by respecting the given law of physics that stems from the PDE of the considered problem. They aim to provide a surrogate of the solution by minimizing the corresponding PDE residual. In general, PDEs are solved by numerical methods such as the finite element method. However, these classical methods are mesh-based and use approximative derivatives yielding discretization errors. PINNs provide the ability to solve PDEs by combining automatic differentiation, which computes gradients at machine precision [5] completely mesh-free, with the capability of neural networks to serve as universal function approximators [6]. For complex problems, already this mesh generation is time consuming, not to mention the high computational effort of the whole simulation itself.

We address this issue by investigating the generalization capabilities of PINNs. Can we train a PINN to solve several reference problems such that it is possible to use this model for different input parameters afterwards, and thus save computational time? Since a fully specified problem always requires boundary and initial conditions, the question is in other words: Can PINNs be trained in a way such that they provide reliable predictions for arbitrary boundary and initial conditions?

Motivated by the question if PINNs are able to learn physics at all, Raissi, Perdikaris, and Karniadakis (2017) already presented promising results which indicate that "if the given PDE is well-posed" PINNs are "capable of achieving good prediction accuracy" [4]. A theoretical justification for PINNs provide Shin, Darbon, and Karniadakis (2020), showing the consistency of PINNs. In the last three years the repertoire of scientific publications regarding PINNs has been growing rapidly. PINNs have been applied to various fluid mechanics problems [8–11] as well as heat transfer problems [12–15] especially showing the effectiveness of PINNs in solving inverse problems where traditional methods typically fail. Moreover, Lu et al. (2019) developed a user-friendly software tool called DeepXDE [17] which is capable of solving several forward and inverse problems involving differential equations. More recently, it has been shown [18–20] that PINNs can integrate seemlessly multi-fidelity experimental data with the governing equations.

This thesis is organized as follows. First, we study the method of neural networks and look at the concepts that are used during training such as regularization. Subsequently, we introduce physics-informed neural networks and expose parallels and differences towards standard neural networks. Within this section we also describe basic features of the software library DeepXDE. Afterwards, we give a brief introduction into the finite element method and how this method is used in the numerical PDE solver FEniCS which we utilize to construct our ground truth data. We then describe how we apply PINNs to a practical example considering the one-dimensional heat equation. After that, we present the results obtained by the previously explained methods and discuss them afterwards. Finally, we make conclusions based on our observings.

2 Methods

In this chapter we explain the basic concept of a *neural network* (NN) and how this concept is augmented such that the resulting model can be named a *physics-informed neural network* (PINN). Subsequently, we explain the functionality of the numerical PDE solver FEniCS which we are using for generating the ground truth data. After that, we present the application example to which these concepts are applied to.

2.1 Neural Networks

NNs are widely used in many visual based application fields such as autonomous driving, extracting information from text files or teaching a robot to tidy up your room [2, 3, 21]. More relevant for our purposes is that they provide a robust approach of approximating nonlinear mappings [6]. "The form of the mapping is governed by adjustable parameters that have to be *trained*." [22].

The NN requires a dataset that is divided into a training and test dataset under a certain ratio. The training is performed on the usually large training dataset in order to learn the task. The test dataset is used during training as well in order to examine if the model is able to produce proper results when using data that is not part of the training data. Additionally, it provides an indication if the current network architecture is sufficient to learn the task or has to be adjusted. A part of the test data is used to make predictions afterwards. The goal is to acquire a model that is able to perform well on unseen data.

NNs are either used for *classification* or *regression*. In a classification task the NN has to decide to which class a certain input feature belongs to. A common example is the classification of handwritten digits where the well-known MNIST dataset [23] is used. A regression task refers to learning a (nonlinear) function which we address in this thesis.

Network Architecture

The architecture of an artificial neural network is based on the concept of the biological analogue - the brain. An important role is played by the interconnected neurons that are responsible for the exchange of information.

Definition 2.1. (Neuron)
Let $d \in \mathbb{N}$. An artificial *neuron* $\nu\colon \mathbb{R}^d \to \mathbb{R}$ with *weight* $\boldsymbol{w} \in \mathbb{R}^d$, *bias* $b \in \mathbb{R}$ and *activation function* $\sigma\colon \mathbb{R} \to \mathbb{R}$ is defined as the map $\nu(\boldsymbol{x}) = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$ for $\boldsymbol{x} \in \mathbb{R}^d$.

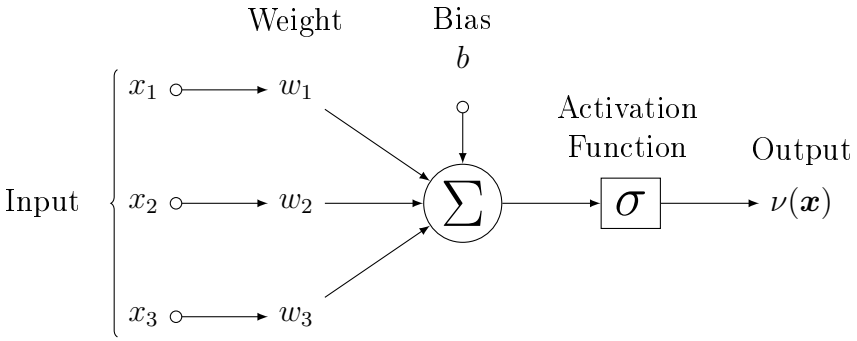


Figure 2.1: Model of an artificial neuron taking an input feature of dimension $d = 3$.

The neurons are arranged in *layers*. A NN can then be regarded as a parallel circuit of concatenated neurons as visualized in Figure 2.2.

Definition 2.2. [24] (FNN)
Let L, d, $n_1, \ldots, n_L \in \mathbb{N}$ and $n_0 := d$. A L-layer *feedforward neural network* $\hat{u}\colon \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ with affine linear maps $A_l\colon \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}$, $\boldsymbol{x} \mapsto A_l(\boldsymbol{x}) = \boldsymbol{W}_l\boldsymbol{x} + \boldsymbol{b}_l$ with $\boldsymbol{W}_l \in \mathbb{R}^{n_l \times n_{l-1}}$, $\boldsymbol{b}_l \in \mathbb{R}^{n_l}$ and *activation functions* $\sigma_l\colon \mathbb{R} \to \mathbb{R}$, $l = 1, ..., L$ is defined as

$$\begin{aligned}
\text{Input Layer}: \quad & \hat{u}^{(0)}(\boldsymbol{x}) = \boldsymbol{x} \in \mathbb{R}^d \\
\text{Hidden Layers}: \quad & \hat{u}^{(l)}(\boldsymbol{x}) = \sigma_l\left(\boldsymbol{W}_l\hat{u}^{(l-1)}(\boldsymbol{x}) + \boldsymbol{b}_l\right) \in \mathbb{R}^{n_l} \qquad \text{for } 1 \le l \le L-1 \\
\text{Output Layer}: \quad & \hat{u}(\boldsymbol{x}) \quad = \boldsymbol{W}_L\hat{u}^{(L-1)}(\boldsymbol{x}) + \boldsymbol{b}_L \in \mathbb{R}^{n_L}
\end{aligned}$$

where the activation functions are used component-wise. Here, d is the dimension of the *input layer*, L denotes the *number of layers* also called *depth* of $\hat{u}$, $n_1, \ldots, n_{L-1}$ denote the number of neurons for each of the $L-1$ *hidden layers*, also called *width* of the respective layer. If $n_1 = \cdots = n_{L-1}$ then, n_i is called *width* of $\hat{u}$ for $i \in \{1, \ldots, L-1\}$. n_L is the dimension of the *output layer*. The matrices $\boldsymbol{W}_l$ contain the network's *weights* and the vector $\boldsymbol{b}_l$ *biases*.

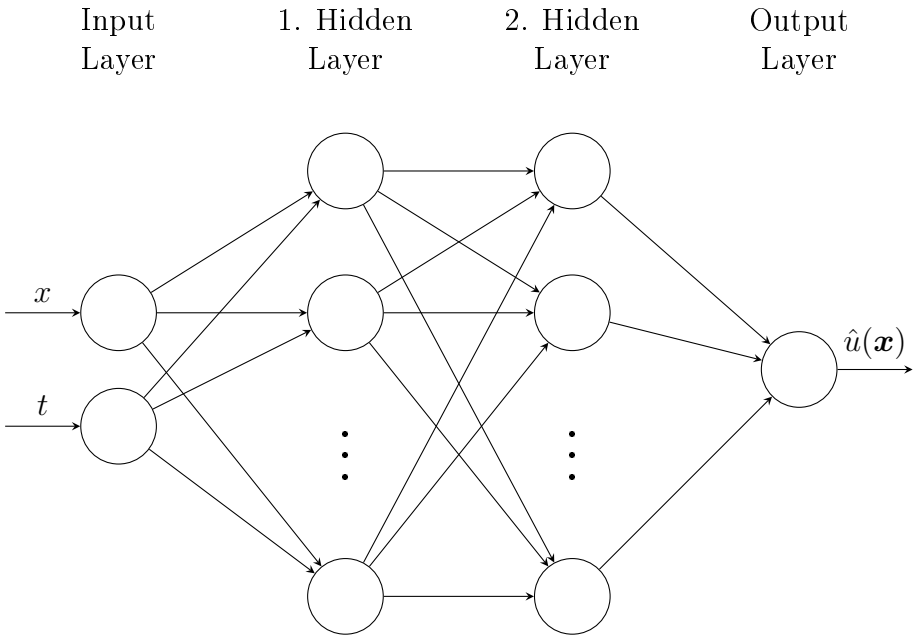


Figure 2.2: FNN with two hidden layers and $\boldsymbol{x} = (x, t)^T \in \mathbb{R}^2$ as input.

The graph of a FNN is acyclic and has only egdes which follow the direction from input to output. The activation function "decides whether a neuron is excited or inhibited through other neurons" [22]. Moreover, it is responsible for the nonlinear transformation of the input which prevents the model from being a linear regression model. Commonly used activation functions are pictured in Figure 2.3. Each activa-

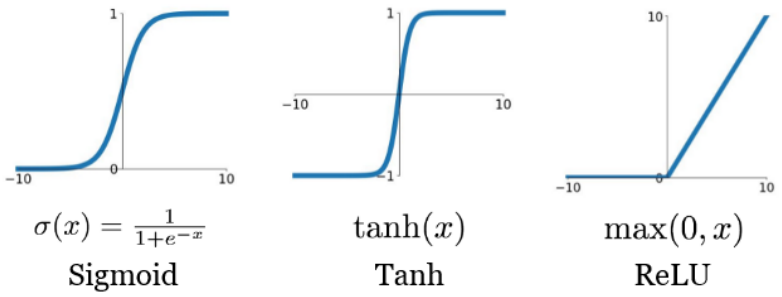


Figure 2.3: Examples of activation functions used for neural networks.

tion function has its advantages and disadvantages. For more information we refer to [25]. Notice, that there is no specific rule which activation function to choose for a given task. In this thesis, we are using the *hyperbolic tangent function*

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{\mathrm{e}^x - \mathrm{e}^{-x}}{\mathrm{e}^x + \mathrm{e}^{-x}}$$

as it is used in [16, 26] for similar regression tasks. Note, that for regression tasks

it is common to not apply an activation function to the output layer. Unlike for classification tasks, here, continuous values are required instead of discrete ones.

For simplicity we combine the learnable parameters $\boldsymbol{W}_l \in \mathbb{R}^{n_l \times n_{l-1}}$ and $\boldsymbol{b}_l \in \mathbb{R}^{n_l}$ into one parameter $\boldsymbol{\Theta}_l \in \mathbb{R}^{n_l \times n_{l-1}+1}$ for $l \in \{1, \dots, L\}$. Since the prediction $\hat{u}$ is dependent on $\Theta := (\boldsymbol{\Theta}_1, \dots, \boldsymbol{\Theta}_L) \in \mathbb{R}^\mu$ with $\mu = \sum_{l=1}^{L} n_l(n_{l-1}+1)$ we also write $\hat{u}_\Theta(\boldsymbol{x})$.

2.1.1 Training

The training of the network's parameters corresponds to minimizing a specified *loss function*. For a given input $\boldsymbol{x}$ the loss function describes a measurement of the error between the network's output $\hat{u}_\Theta(\boldsymbol{x})$ and the *ground truth* $u(\boldsymbol{x})$, i.e. the desired output. There are several loss functions that are used for machine learning applications according to which task they have to solve. Since we have a regression task the *mean squared error* (MSE) loss is a proper loss function.

Definition 2.3. (MSE Loss)
Let $\hat{u}_\Theta \colon \mathbb{R}^d \to \mathbb{R}^{n_L}$ with $d, n_L \in \mathbb{N}$ be a FNN as defined in Def. 2.2. Further, let $\Omega \subset \mathbb{R}^d$ be a bounded domain and $u \colon \Omega \to \mathbb{R}^{n_L}$ be the ground truth function the FNN aims to approximate. Then, for a given set of training points $\mathcal{T} \subset \Omega$ the *mean squared error loss* $\mathcal{L}_\mathcal{T} \colon \mathbb{R}^\mu \to \mathbb{R}$ is defined as

$$\mathcal{L}_\mathcal{T}(\Theta) = \frac{1}{|\mathcal{T}|} \sum_{\boldsymbol{x} \in \mathcal{T}} \|\hat{u}_\Theta(\boldsymbol{x}) - u(\boldsymbol{x})\|_2^2.$$

Due to the squaring, larger errors are penalized more heavily than smaller ones. In addition to that, quadratic functions are prefered over other mappings, e.g. $|\cdot|_1$-norm, since they are differentiable in each point. One can add an additional summand to the loss function containing a certain *regularization*.

Regularization

The purpose of regularization is to avoid *overfitting* which would lead to a model that fits the training data too precisely. The other extreme is called *underfitting*. An underfitted model barely fits the training data and is most likely not able to provide reliable predictions for unseen data. The principles of overfitting and underfitting are visualized in Figure 2.4. Both cases result in a model that is not able to perform well on unseen data and thus has poor generalization capabilities. Since the concept of generalization is the key essence of this thesis, we explain the functionality of regularization with an example.

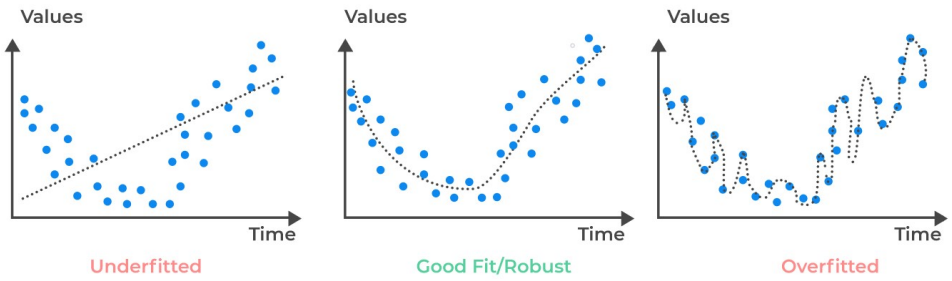


Figure 2.4: Example of a model that is underfitted, balanced or overfitted. The blue markers represent the training data and the dotted line the prediction of the model. Source: `https://analystprep.com/study-notes/cfa-level-2/quantitative-method/overfitting-methods-addressing/`.

Consider a regression problem where the constructed NN serves as function approximation. If the model is overfitted, the resulting function will contain a lot of curvature. Suppose this function has the mathematical description of a polynomial. Then, the coefficients of higher degrees would be large. In order to smoothen the function, these coefficients ought to shrink. This is achieved by regularization. It penalizes the weights that are considerably large compared to the others. There are different types of regularization such as e.g. L_1 *(Lasso) regularization* or L_2 *(Ridge) regularization*. We show the example of L_2 regularization. Consider the MSE loss as defined in Definition 2.3 for a L-layer FNN $\hat{u}_\Theta \colon \mathbb{R}^d \to \mathbb{R}^{n_L}$ with weights and biases summarized in the parameter Θ. If we add a L_2 regularization to the loss function we obtain

$$\mathcal{L}_\mathcal{T}(\Theta) = \frac{1}{|\mathcal{T}|} \sum_{\boldsymbol{x} \in \mathcal{T}} \|\hat{u}_\Theta(\boldsymbol{x}) - u(\boldsymbol{x})\|_2^2 + \lambda \sum_{j \in U \setminus B} \Theta_j^2$$

for $U = \{0, \dots, \mu - 1\}$ and $B = \bigcup_{l=1}^{L} B_l$ as the set of indices of all biases where $B_l = \left\{ \sum_{i=1}^{l-1} n_i(n_{i-1}+1) + k_l(n_{l-1}+1) - 1 : 1 \leq k_l \leq n_l \right\}$ denotes the set of indices of biases of layer l. The *regularization parameter* $\lambda \geq 0$ governs how much we want to penalize the flexibility of the model. Note, that only the weights and not the biases are considered for regularization.

The differentiability of the loss function follows from the differentiability of the activation function and the fact that the $\|\cdot\|_2$-norm is differentiable as well. The training process of the network under the MSE loss can then be formulated as the optimization problem

$$\min_\Theta \mathcal{L}(\Theta)$$

with loss function $\mathcal{L}$ as defined in Def. 2.3 and $\Theta \in \mathbb{R}^{\mu}$ representing the network's parameters. The minimization is done using a gradient descent approach. The *method of steepest descent* (Alg. 1) is "the simplest gradient method for optimization" [27]. However, for non-convex functions it is possible that the algorithm fails

Algorithm 1 Method of Steepest Descent

Require: objective function $\mathcal{L}(\Theta)$
Require: Initialization: Θ_0
  $k \leftarrow 0$ ▷ Initialize iteration step
  $\alpha_k > 0$ ▷ Compute step size via line search
  while Θ_k not converged **do**
    $k \leftarrow k + 1$
    $g_k \leftarrow \nabla_{\Theta}\mathcal{L}(\Theta_{k-1})$ ▷ Compute gradient at iteration step k
    $\Theta_k \leftarrow \Theta_{k-1} - \alpha_{k-1}g_k$ ▷ Update parameters
    $\alpha_k \leftarrow \alpha_{k-1}$ ▷ Compute new step size
  end while
  return Θ_k ▷ Resulting parameters

to converge towards the global minimum. This is the case for the choice of a small learning rate for instance. The algorithm converges more slowly and possibly towards a local minimum. Otherwise, although a large learning rate could accelerate the process until convergence there is the risk of not reaching the global optimum at all. Therefore, this method is rarely used in practice.

In this thesis, we consider a *stochastic gradient descent* (SGD) approach. Unlike gradient descent, a SGD method only takes small subsets of the training dataset, so-called *mini-batches*, in each optimization step. This yields only approximations of the actual gradients leading to a noisy gradient in total. Due to this noise, the method is capable of escaping possible local minima. One can draw an analogy to the momentum from physics where it is described as a mass having a certain velocity while moving in a specific direction. If the algorithm reaches a local minimum, it is likely to leaving it again due to the momentum and the algorithm continues optimizing. Instead, a gradient descent approach would end optimization when reaching a local minimum due to a zero gradient. Thus, SGD is effective for objective functions with a large amount of curvatures.

The SGD method we are dealing with is called *Adam* (adaptive moment estimation), first proposed by Kingma and Ba, 2014. According to the authors, it "only requires first-order gradients with little memory requirement" [28].Also, "the magnitudes of parameter updates are invariant to rescaling of the gradient". Adam tackles the aforementioned problems by using estimations of first (mean) and second moments (uncentered variance) of the gradient to adapt the learning rate for

Algorithm 2 Adam

Require: stepsize α, decay rates $\beta_1, \beta_2 \in [0, 1)$, objective function $\mathcal{L}(\Theta)$
Require: Initialization: Θ_0
$k \leftarrow 0$ ▷ Initialize iteration step
$m_0 \leftarrow 0$ ▷ Initialize 1st moment vector
$v_0 \leftarrow 0$ ▷ Initialize 2nd moment vector
while Θ_k not converged **do**
$k \leftarrow k + 1$
$g_k \leftarrow \nabla_\Theta \mathcal{L}(\Theta_{k-1})$ ▷ Compute gradient at iteration step k
$m_k \leftarrow \beta_1 \cdot m_{k-1} + (1 - \beta_1) \cdot g_k$ ▷ Update biased 1st moment estimate
$v_k \leftarrow \beta_2 \cdot v_{k-1} + (1 - \beta_2) \cdot g_k^2$ ▷ Update biased 2nd raw moment estimate
$\hat{m}_k \leftarrow \frac{m_k}{1-\beta_1^k}$ ▷ Compute bias-corrected 1st moment estimate
$\hat{v}_k \leftarrow \frac{v_k}{1-\beta_2^k}$ ▷ Compute bias-corrected 2nd raw moment estimate
$\Theta_k \leftarrow \Theta_{k-1} - \alpha \cdot \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \epsilon}$ ▷ Update parameters
end while
return Θ_k ▷ Resulting parameters

the networks's parameters in each update. By calculating the exponential moving average, it "adds history to the parameter update equation based on the gradient encountered in the previous updates" [29].

In Algorithm 2, g_k^2 denotes the elementwise square $g_k \odot g_k$. The authors recommend $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ as default settings. Further details can be read in [28].

In deep learning the gradient $\nabla\mathcal{L}(\Theta)$ is obtained by *backpropagation* which, as the name suggests, computes the gradient by using the chain rule while propagating backwards through the network. For more details we refer to [30, pp. 140-148]. Machine learning frameworks such as TensorFlow (Google Brain Team, 2015) and PyTorch (Paszke et al., 2016) have implemented the method of *automatic differentiation* (AD). One might assume that these are distinct methods as they have different names. But both are strongly related to each other. As a matter of fact, backpropagation is a special case of AD.

2.2 Physics-Informed Neural Networks

The NN is a solely data-driven method mainly used for applications where enough data is available. Especially image data can easily be gathered from open source databases. However, most real-world phenomena are not described by images but physical laws characterizing conservation laws, diffusion processes, advection-diffusion-reaction or other systems. It is possible that only partial data is available when observing such phenomena which results in a poor predictive performance when using NNs. This motivated the idea to involve these physical laws, often formulated as partial differential equations (PDEs), as additional source of information into practical computations and paved the way for PINNs to emerge. PINNs are NNs which are not only trained with ground truth data, but learn the scientific laws described by the considered PDE. Before we explain the concept of a PINN in more detail, we introduce some basic definitions which are based on [31].

Definition 2.4. (Higher-Order Partial Derivative)
Let $u\colon \Omega \to \mathbb{R}$ with $\Omega \subset \mathbb{R}^d$ being a domain. Further, let $\boldsymbol{k} = (k_1, \dots, k_d)^T \in \mathbb{N}_0^d$ be a multi-index of order $|\boldsymbol{k}| := k_1 + \cdots + k_d$. Then

$$D^{\boldsymbol{k}} u = \partial_{x_1}^{k_1} \cdots \partial_{x_d}^{k_d} u$$

denotes a *partial derivative of order* $|\boldsymbol{k}|$.

Definition 2.5. (PDE)
Let $f\colon \Omega \times \mathbb{R}^m \to \mathbb{R}$ and $\boldsymbol{k}_1, \dots, \boldsymbol{k}_m$ be multi-indices and $u\colon \Omega \to \mathbb{R}$. Then

$$f\left(\boldsymbol{x}, D^{\boldsymbol{k}_1} u, \dots, D^{\boldsymbol{k}_m} u\right) = 0, \quad \boldsymbol{x} \in \Omega,$$

is called *partial differential equation* (PDE) of order $p = \max_{1 \le i \le m} |\boldsymbol{k}_i|$ for the function u.

We restrict ourselves to the class of *well-posed* problems first introduced by Hadamard. In the context of PDEs we obtain the following definition.

Definition 2.6. (Well-Posed Problem)
Consider the problem described by the PDE

$$f\left(\boldsymbol{x}, D^{\boldsymbol{k}_1} u, \dots, D^{\boldsymbol{k}_m} u\right) = h$$

on $\Omega \times (0, T)$, $\Omega \subset \mathbb{R}^d$, with boundary condition (BC)

$$\mathcal{B}(u) = g \quad \text{on } \partial\Omega$$

and initial condition (IC)

$$\mathcal{I}(u) = u_0 \quad \text{in } \Omega \text{ at } t = 0.$$

Then, the problem is *well-posed in the sense of Hadamard* if it holds

1. There exists at least one solution for every (appropriate) data.
2. For every h, g and u_0 there exists at most one solution.
3. The solution $u = u(h, g, u_0)$ depends continuously on the data h, g and u_0, i.e. small changes in the data yield only small changes in the solution.

A PDE defined with an IC and BCs constitutes an *initial-boundary-value problem*. In order to deal with well-posed problems in practice, we need to prescribe proper BCs that u assumes along the boundary $\partial\Omega$. Possible choices are

1. *Dirichlet*: $u = u_D$ on $\partial\Omega$
2. *Neumann*: $\boldsymbol{n} \cdot \nabla u = u_N$ on $\partial\Omega$
3. *Robin*: $\alpha \boldsymbol{n} \cdot \nabla u + \beta u = u_R$ on $\partial\Omega$

for given functions $u_D, u_N, u_R \colon \partial\Omega \to \mathbb{R}$ and constants α, $\beta \in \mathbb{R}$. Here, $\boldsymbol{n}$ denotes the *normal vector* and $\nabla = \left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_d}\right)^T$ the *gradient operator*. Further, by $\Delta = \left(\frac{\partial^2}{\partial x_1^2}, \ldots, \frac{\partial^2}{\partial x_d^2}\right)^T$ we denote the *Laplace operator*.

PINN Architecture

We describe the concept of a PINN using ideas and notations from [4, 16]. Since we consider PDEs in numerical simulations, we restrict our domain to be bounded. Therefore, let $\tilde{\Omega} = \Omega \times [0, T]$ where $\Omega \subset \mathbb{R}^d$, $d \in \mathbb{N}$, is bounded. Additionally, we define our boundary to be $\partial\tilde{\Omega} = \partial\Omega \times (0, T) \cup \Omega \times \{0\}$ since numerically the boundary and initial data are treated equally. Thus, in the following we treat the IC as a special type of BC. For $\boldsymbol{x} = (x_1, \ldots, x_d, t)^T \in \tilde{\Omega}$ we consider a PDE with solution $u \colon \tilde{\Omega} \to \mathbb{R}$ denoted by

$$f\left(\boldsymbol{x}, D^{\boldsymbol{k}_1} u, \ldots, D^{\boldsymbol{k}_m} u\right) = 0, \quad \boldsymbol{x} \in \tilde{\Omega},$$

with boundary conditions

$$\mathcal{B}(u, \boldsymbol{x}) = 0, \quad \boldsymbol{x} \in \partial\tilde{\Omega}.$$

In order to solve this PDE with a PINN, the following procedure [16] has to be fulfilled.

1. Construct a NN $\hat{u}_\Theta$ with network parameters Θ.
2. Specify the two training sets $\mathcal{T}_f$ and $\mathcal{T}_b$ for the PDE and BC/IC, respectively.
3. Specify the loss function $\mathcal{L}_\mathcal{T}(\Theta)$ by summing the weighted MSE loss of both the PDE and BC/IC residual.
4. Train the NN by minimizing the loss function $\mathcal{L}_\mathcal{T}(\Theta)$.

A visualization of this procedure solving the one-dimensional heat quation $u_t = \lambda u_{xx}$ with mixed BCs is pictured in Figure 2.5. The basis forms a NN $\hat{u}_\Theta$ with network parameters Θ as defined in Definition 2.2. Notice that PINNs are not constrained to FNNs but could have other network architectures as well. The model $\hat{u}_\Theta$ acts as a

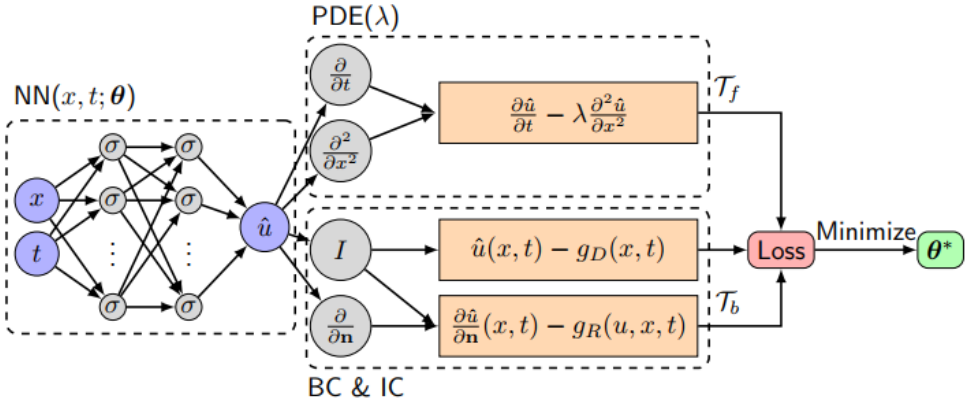


Figure 2.5: Schematic of a PINN solving the one-dimensional heat equation $u_t = \lambda u_{xx}$ with mixed BCs $u(x,t) = g_D(x,t)$ on $\Gamma_D \subset \partial\tilde{\Omega}$ and $\frac{\partial u}{\partial \boldsymbol{n}}(x,t) = g_R(u,x,t)$ on $\Gamma_R \subset \partial\tilde{\Omega}$. The IC is treated as a special type of Dirichlet BC. $\mathcal{T}_f$ and $\mathcal{T}_b$ denote the sets of training points for the PDE and BC/IC residuals, respectively. Source: [16].

surrogate of the solution u. In order to solve the PDE, the computation of (partial) derivatives is essential. By the usage of AD we are able to compute the derivatives (of any order) of our network $\hat{u}_\Theta$ with respect to all relevant input variables, independent from the structure of the underlying programming code. Thus, we can include the PDE residual into our computations with no need for a mesh generation as it is used for the *finite element method*. This inclusion is done by considering two subsets of the training data $\mathcal{T} \subset \tilde{\Omega}$. The set $\mathcal{T}_f \subset \tilde{\Omega}$ contains points in the domain and $\mathcal{T}_b \subset \partial\tilde{\Omega}$ points on the boundary and initial data. The training points are randomly distributed in the domain. As indicated in Figure 2.5, only for points in $\mathcal{T}_b$ ground

truth data is used. For points in $\mathcal{T}_f$ the given PDE residual is minimized. This is the key difference from a standard NN which uses ground truth data for all considered training data. In doing this, the model $\hat{u}_\Theta$ learns the physical law imposed by the PDE.

2.2.1 Training

Like NNs, PINNs aim to minimize a certain loss function. But different from standard NNs, the loss function of a PINN is a linear combination of two loss functions evaluated at points in $\mathcal{T}_f$ and $\mathcal{T}_b$, separately. It is conveninent to choose the MSE as loss function for each training set.

Definition 2.7. (PINN Loss)
Let $\hat{u}_\Theta \colon \mathbb{R}^d \to \mathbb{R}$ with $d \in \mathbb{N}$ be a FNN as defined in Def. 2.2. Further, let $\tilde{\Omega} \subset \mathbb{R}^d$, $d \in \mathbb{N}$, be a bounded domain and $u \colon \tilde{\Omega} \to \mathbb{R}$ be the solution of the PDE $f\left(\boldsymbol{x}, D^{\boldsymbol{k}_1}u, \dots, D^{\boldsymbol{k}_m}u\right) = 0$ with boundary and initial data $\mathcal{B}(u, \boldsymbol{x}) = 0$ for $\boldsymbol{x} \in \tilde{\Omega}$. Then, for given sets of training points $\mathcal{T}_f \subset \tilde{\Omega}$ and $\mathcal{T}_b \subset \partial\tilde{\Omega}$ the *PINN loss* $\mathcal{L}_\mathcal{T} \colon \mathbb{R}^\mu \to \mathbb{R}$ is defined as

$$\mathcal{L}_\mathcal{T}(\Theta) = w_f \mathcal{L}_{\mathcal{T}_f}(\Theta) + w_b \mathcal{L}_{\mathcal{T}_b}(\Theta)$$

where

$$\mathcal{L}_{\mathcal{T}_f}(\Theta) = \frac{1}{|\mathcal{T}_f|} \sum_{\boldsymbol{x} \in \mathcal{T}_f} \left\| f\left(\boldsymbol{x}, D^{\boldsymbol{k}_1}\hat{u}, \dots, D^{\boldsymbol{k}_m}\hat{u}\right) \right\|_2^2,$$
$$\mathcal{L}_{\mathcal{T}_b}(\Theta) = \frac{1}{|\mathcal{T}_b|} \sum_{\boldsymbol{x} \in \mathcal{T}_b} \left\| \mathcal{B}(\hat{u}, \boldsymbol{x}) \right\|_2^2$$

with *weights* w_f and w_b.

In section 2.1 we explained the concept of regularization which prevents the model from the issue of overfitting. In fact, when looking at the PINN loss, the term $w_f \mathcal{L}_{\mathcal{T}_f}(\Theta)$ acts as regularization with regularization parameter w_f in order to penalize those parameters Θ which would lead to solutions not satisfying the PDE. "Therefore, a key property of PINNs is that they can be effectively trained using small datasets" [4]. Also, the authors of [7] note that "it has been empirically reported that a properly chosen boundary weight w_b could accelerate the overall training and result in a better performance". It is recommended [32] to choose w_b large and positive in order to improve the accuracy of the model.

Like NNs, the training corresponds to the minimization problem $\min_\Theta \mathcal{L}(\Theta)$. Since

the PINN loss is a linear combination of two MSE losses, it is differentiable by the same reasoning as for NNs. The training of the network's parameters Θ is proceeded using a gradient descent approach like Adam [28] or L-BFGS [33]. "The required number of iterations highly depends on the problem (e.g. the smoothness of the solution)" [16]. In each step of the training process, (partial) derivatives have to be calculated. Thus, if the number of points in $\mathcal{T}_f$ is very large, it is computationally expensive to calculate the PINN loss in every iteration. Lu et al. proposed a strategy to improve the efficiency of the training process called *residual-based adaptive refinement* which we will discuss later.

Since we are solving a non-convex optimization problem, there is no theoretical guarantee that this process converges to the global minimum. However, Raissi, Perdikaris, and Karniadakis emphasize in [4] that "if the given PDE is well-posed" the method PINN "is capable of achieving good prediction accuracy given a sufficient number of collocation points" lying in $\mathcal{T}_f$. As a matter of fact, Shin, Darbon, and Karniadakis present in [7] that for linear second-order elliptic and parabolic type PDEs "the sequence of minimizers strongly converges to the PDE solution in C_0", where C_0 denotes the set of continuous functions. They state furthermore that "if each minimizer satisfies the initial/boundary conditions the convergence mode becomes H_1" [7] where H_1 denotes the *Sobolev space* which we will introduce later. Even more than that state Cai et al. in [18] where they ascertained with great success that PINNs are able to solve ill-posed problems as well when providing only partial measurement data on the boundary.

2.2.2 Application Library: DeepXDE

Lu et al. developed a software library called DeepXDE [17] which enables the user to solve forward and inverse PDEs via PINNs. Additionally, it is able to solve other kinds of problems such as forward and inverse integro-differential equations. In [16] it is said that the library was designed "to make the user code stay compact and manageable, resembling closely the mathematical formulation". Thus, it simplifies the process of analyzing scientific problems when using machine learning methods. DeepXDE supports the Python libraries Tensorflow 1.x, 2.x and PyTorch. In the following, we expose some of the advantages of DeepXDE proposed in [16].

A useful feature of the library is that it supports complex geometry domains using the technique of constructive solid geometry (CSG). Although DeepXDE already supports basic geometries (rectangle, circle, etc.), it is possible that the user needs to define a more complex geometry. CSG supports two- and three-dimensional domains. Some examples of possible domains are displayed in Figure 2.6.

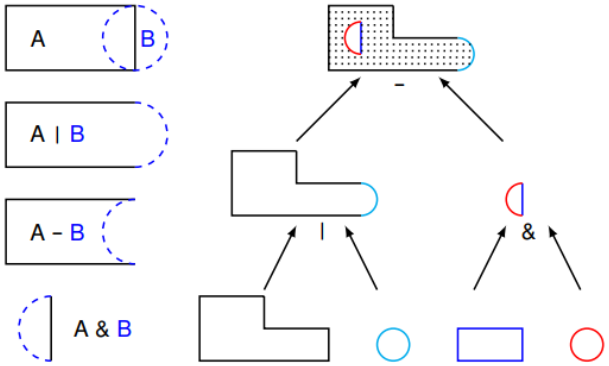


Figure 2.6: Examples of constructive solid geometry (CSG) in 2D. *Left*: A and B represent the rectangle and circle, respectively. From A and B the union $A|B$, difference $A-B$ and intersection $A\&B$ are constructed. *Right*: A complex geometry (top) is constructed from a polygon, a rectangle and a circle (bottom). Source: [16].

One major advantage is that the user does not have to provide training data. The user defines the PDE and BC/IC as functions represented in analytical form. One can either specify the locations of the points or only set the number of points and then DeepXDE will sample the required number randomly on a grid covering the specified domain.

Another advantage is that DeepXDE increases the training efficiency by employing the *residual-based adaptive refinement* (RAR) method. In some cases where functions contain areas with steep gradients, it is useful to choose more training points in these areas. However, in many applications the shape of the solution is unknown in advance. Thus, the design of the distribution of considered training points poses a challenge. To this end, Lu et al. proposes with RAR a method which improves the distribution of training points during training process by adding more points in the locations where the PDE residual is large. This is repeated until the mean residual

$$\varepsilon_r = \frac{1}{V} \int_{\tilde{\Omega}} \left\| f\left(\boldsymbol{x}, D^{\boldsymbol{k}_1} u, \ldots, D^{\boldsymbol{k}_m} u\right) \right\| d\boldsymbol{x}$$

is smaller than a threshold ε where V denotes the volume of $\tilde{\Omega}$.

Further, the user has the ability to add so-called callback functions. This enables the user to monitor the training process and to make modifications in real time, e.g. change the learning rate, if necessary. Possible callback functions can for example save the model after certain epochs, calculate the first derivative of the outputs with respect to the inputs, or monitor a movie of the spectrum of the function's Fourier transform. It is convenient to define customized callback functions which are called at specified stages of the training process.

An overview of the usage of DeepXDE for solving PDEs is depicted in Figure 2.7. The corresponding procedure is according to [16] summarized as follows.

1. Specify the computational domain using the `geometry` module.
2. Specify the PDE using the grammar of `TensorFlow`.
3. Specify the boundary and initial conditions.
4. Combine geometry, PDE and BCs/ICs together into `data.PDE` or `data.TimePDE` for time-independent or time-dependent problems, respectively. Specify the training data by either set the specific point locations or only set the number of points.
5. Construct a NN using the `maps` module.
6. Define a `model` by combining the PDE problem in 4 and the NN in 5.
7. Call `model.compile` to set the optimization hyperparameters such as optimizer and learning rate. The weights w_f, w_b can be set here by `loss_weights`.
8. Call `model.train` to train the network from random initialization or a pre-trained model using the argument `model_restore_path`. It is extremely flexible to monitor and modify the training behavior using `callbacks`.
9. Call `model.predict` to predict the PDE solution at different locations.

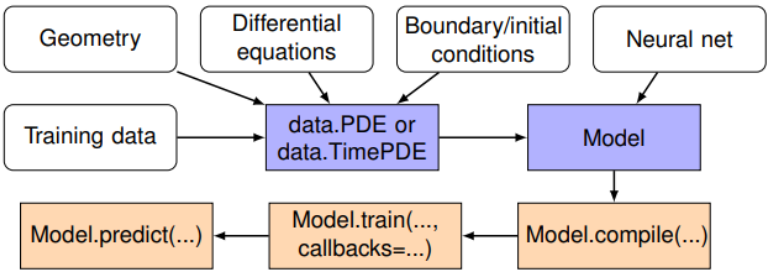


Figure 2.7: Flow chart of the usage of DeepXDE. The white boxes define the PDE and the training hyperparameters. The blue boxes combine the PDE problem and training hyperparameters in the white boxes. The orange boxes describe the steps to solve the PDE (from right to left). Source: [16].

At the current state of the art, DeepXDE does not support the definition of various BCs/ICs for a specific PDE. Thus, we cannot use this library for the generalization task without having to change the code manually. Also, although it is user-friendly it is difficult to take control over the single steps that proceed in the background.

2.3 Generating Ground Truth

Since we address the generalization of PINNs in the manner of *proof of concept*, we need reference solutions to which we compare our numerical results. Thus, we have to solve various initial-boundary-value problems numerically. The conventional way to do this is via the *finite element method* (FEM). A popular software library for solving PDEs by using the FEM is FEniCS [34]. Before we describe how to use FEniCS in practical applications, we explain the theory of the FEM using material from [35, 36].

2.3.1 Finite Element Method

The FEM is the most important method for solving PDEs, at least for elliptic and parabolic PDEs. The basic idea is to transform the considered PDE which is usually given in the so-called *strong form* into the *weak form*. Essential for this approach is the proper choice of the underlying function space. Therefore, we briefly explain which function spaces we are dealing with in this section.

Consider an open set $\Omega \subset \mathbb{R}^d$ with piecewise smooth boundaries. By $L_2(\Omega)$ we denote the Hilbert space of square-integrable functions with inner product and norm

$$\langle f, g \rangle_{L_2(\Omega)} := \int_\Omega f(x)\overline{g(x)}dx, \quad \|f\|_{L_2(\Omega)} := \left(\int_\Omega |f(x)|^2 dx \right)^{\frac{1}{2}}.$$

Concerning PDEs, we obviously need to deal with derivatives. In $\mathbb{R}$ we would intuitively use

$$u'(x) = \lim_{h \to 0} \frac{u(x+h) - u(x)}{h}.$$

However, pointwise evaluation does not make sense in $L_2(\Omega)$. Thus, we require another definition of derivatives for this function space. To this end, we define

$$C_\infty^0(\Omega) := \{\phi \in C_\infty(\Omega) : \Omega \supset \text{supp}\phi \text{ is compact}\}$$

where $C_\infty(\Omega)$ denotes the set of infinitely often differentiable functions on Ω.

Definition 2.8. [36] (Weak Derivative)
A function $f \in L_2(\Omega)$ posseses a *weak derivative* $D_w^{\boldsymbol{k}} f$ if there exists a function $g \in L_2(\Omega)$ with

$$\int_\Omega g(x)\phi(x)dx = (-1)^{|\boldsymbol{k}|} \int_\Omega f(x) D^{\boldsymbol{k}} \phi(x) dx \quad \forall \phi \in C_\infty^0(\Omega).$$

Then it holds $D_w^{\boldsymbol{k}} f = g$.

We can express Definition 2.8 also in terms of the inner product by

$$\langle g, \phi \rangle_{L^2(\Omega)} = (-1)^{|\boldsymbol{k}|} \left\langle f, D^{\boldsymbol{k}} \phi \right\rangle \quad \forall \phi \in C_\infty^0(\Omega).$$

With this formalism we can define the important function space which is used for the FEM.

Definition 2.9. [36] (Sobolev Space)
For $m > 0$ let $H_m(\Omega)$ be the set of all functions $u \in L_2(\Omega)$ which posses a weak derivative $D_w^{\boldsymbol{k}} u \in L_2(\Omega)$ for all $|\boldsymbol{k}| \leq m$. In $H_m(\Omega)$ we define a inner product and norm

$$\langle u, v \rangle_{H_m(\Omega)} = \sum_{|\boldsymbol{k}| \leq m} \left\langle D_w^{\boldsymbol{k}} u, D_w^{\boldsymbol{k}} v \right\rangle_{L_2(\Omega)}, \quad \|u\|_{H_m(\Omega)} = \left(\sum_{|\boldsymbol{k}| \leq m} \|D_w^{\boldsymbol{k}} u\|_{L_2(\Omega)}^2 \right)^{\frac{1}{2}}.$$

$H_m(\Omega)$ is called *Sobolev space*.

The Sobolev space $H_m(\Omega)$ is complete and hence a Hilbert space [36]. Since we apply the presented methods to the heat equation later, we will describe how the FEM is applied to this example.

Let $\Omega \subset \mathbb{R}^d$ be a bounded domain and $[0, T]$ be the time interval of interest. We are looking for a solution $u \colon \Omega \times [0, T] \to \mathbb{R}$ which fulfills the heat equation

$$u_t = \Delta u + f \tag{2.1}$$

where $f \colon \Omega \to \mathbb{R}$ is the supplied heat. We consider a combination of Dirichlet and Neumann BCs

$$u = u_D \quad \text{on } \Gamma_D, \qquad \frac{\partial u}{\partial \boldsymbol{n}} = u_N \quad \text{on } \Gamma_N,$$

with Γ_D and Γ_N forming the boundary of Ω, i.e. $\Gamma_D \cup \Gamma_N = \partial\Omega$ and $\Gamma_D \cap \Gamma_N = \varnothing$. Since we have a time-dependent problem, we need to prescribe initial data

$$u(\cdot, 0) = u_0 \quad \text{on } \Omega.$$

For now, we have given our problem in strong form. The FEM requires the problem to be in weak form. The idea is to fix a time t and consider (2.1) as stationary problem. Then, the following steps are proceeded.

1. Define the function space for the so-called *test functions*

$$V = \{v \in H_1(\Omega) : v = 0 \text{ on } \Gamma_D\}.$$

2. Multiply the strong form (2.1) by test functions $v \in V$ and integrate over domain Ω

$$\int_\Omega v\dot{u}\, dx - \int_\Omega v\Delta u\, dx = \int_\Omega vf\, dx \qquad \forall v \in V.$$

3. Transform second-order derivatives into first-order derivatives by applying Green's formula

$$\int_\Omega \langle \nabla u, \nabla v\rangle\, dx = -\int_\Omega v\Delta u\, dx + \int_{\partial\Omega} v\, \langle \nabla u, \boldsymbol{n}\rangle\, ds$$

and obtain due to $\frac{\partial u}{\partial \boldsymbol{n}} = u_N$ on Γ_N the weak form:

$$\boxed{\begin{aligned}&\text{Find } u \in H_1(\Omega) \text{ with } u = u_D \text{ on } \Gamma_D \text{ such that}\\ &\int_\Omega v\dot{u}\, dx + \int_\Omega \langle \nabla u, \nabla v\rangle\, dx = \int_\Omega vf\, dx + \int_{\Gamma_N} vu_N\, ds \qquad \forall v \in V.\end{aligned}} \tag{2.2}$$

Note that for homogeneous BCs the boundary integral vanishes and we obtain $u \in V$. It is conveninent to introduce an abstract notation for (2.2). We define on $H_1(\Omega) \times H_1(\Omega)$ the *bilinear form*

$$a(u, v) := \int_\Omega \langle \nabla u, \nabla v\rangle\, dx$$

and the linear functionals

$$\langle \dot{u}, v\rangle := \int_\Omega v\dot{u}\, dx, \qquad L(v) := \int_\Omega vf\, dx + \int_{\Gamma_N} vu_N\, ds,$$

for given $f(\cdot, t) \in L_2(\Omega)$ and $u_N(\cdot, t) \in L_2(\Gamma_N)$. Still, we consider an infinite-dimensional function space. Since the FEM is a numerical method, the key step is to switch to a discrete (finite-dimensional) space $V_h \subset V$, where h denotes the grid size of the considered mesh, and we end up in the final step.

4. Apply the *Galerkin projection* which corresponds to the task:

$$\boxed{\begin{aligned}&\text{For each time step } t \text{ find the approximation } u_h \in V_h\\ &\text{with } u_h = u_D \text{ on } \Gamma_D \text{ such that}\\ &\qquad \langle \dot{u}_h, v\rangle + a(u_h, v) = L(v) \qquad \forall v \in V_h.\end{aligned}} \tag{2.3}$$

(2.3) results in a system of ordinary differential equations where the initial value $u_h(\cdot, 0)$ can be computed with interpolation of the IC $u(\cdot, 0) = u_0$ or a so-called L_2 projection of u_0 by $\langle u_h(\cdot, 0), v\rangle = \langle u_0, v\rangle$, for all $v \in V_h$.

It remains the question of how to choose the subspace V_h for the Galerkin projection. In the FEM, the domain Ω is first discretized, i.e. the domain is divided into *elements* that have a simple shape as depicted in Figure 2.8. In each element, the functions in V_h are locally defined piecewise polynomials in d variables with typically the same degree over all elements.

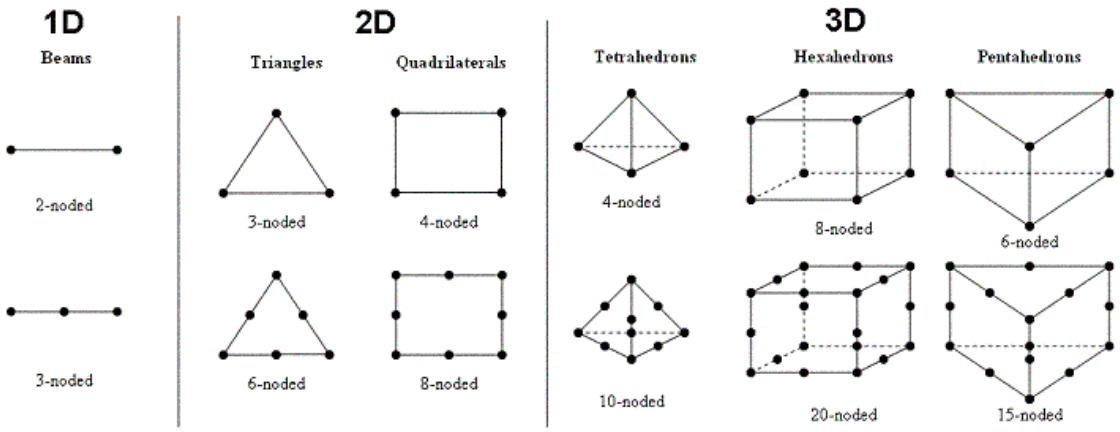


Figure 2.8: *Top:* Element types of degree=1. *Bottom*: Element types of degree=2. The markers • identify nodes in which a function value is interpolated. Source: `https://www.femto-engineering.de/stories/festigkeitsberechnungen/`.

2.3.2 Numerical PDE Solver: FEniCS

As already menioned before, FEniCS is a numerical PDE solver which uses the FEM method. It can be used when programming in Python or C++. Basically, the user has to perform the steps as stated in section 2.3.1. The goal is to provide FEniCS a variational problem. What makes FEniCS attractive is that the steps of writing the program code for defining the variational problem with all relevant assumptions "result in fairly short code, while a similar program in most other software frameworks for PDEs require much more code and technically difficult programming" [35].

Assume we have given the heat equation as stated in section 2.3.1. For simplicity we only assume Dirichlet BCs. FEniCS distinguishes between the *trial space* U and

the *test space* V where

$$U = \{u \in H_1(\Omega) : u = u_D \text{ on } \partial\Omega\},$$
$$V = \{v \in H_1(\Omega) : v = 0 \text{ on } \partial\Omega\}.$$

In order to solve a PDE with FEniCS the user has to perform the following steps [35]:

- Choose the finite element spaces U, V by specifying the domain (the mesh) and the type of function space (polynomial degree and type).
- Express the PDE as a (discrete) variational problem.

We want a representation of the form 2.3. Thus, we consider discrete finite-dimensional subspaces $U_h \subset U$, $V_h \subset V$ and want to find the discrete solution u_h. In the following, the superscript $_h$ is dropped since FEniCS aims to provide "(almost) a one-to-one relationship between the mathematical formulation of a problem and the corresponding FEniCS program". Thus, we refer by U and V to the discrete finite element spaces. Also, u denotes the solution of the discrete problem and u_e corresponds to the exact solution if we need to distinguish between both. Thus, we want to solve the discrete variational problem:

For each time step t find $u \in U$ such that
$a(u, v) = L(v) \qquad \forall v \in V.$

where the bilinear form $a(u, v)$ collects all terms with the unknown solution u and $L(v)$ is the linear functional compromised of all known functions. But how does FEniCS handle the derivative in time? As mentioned in the introduction, there are no implemented methods here computing derivatives accurately. FEniCS solves time-dependent problems by first discretizing the time derivative by a finite difference approximation. For a fixed t, each stationary problem is then transformed into a variational formulation. We will stick to the notations given in [35].

Let u^n denote the solution u at time t_n where n is an integer iterating over all considered time levels. "For simplicity and stability reasons" [35], FEniCS chooses a simple backward difference:

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

with Δt as the discretization parameter. With this, we obtain the time discrete version of the heat equation sampled at some time level t_{n+1}, also called *backward*

Euler or *implicit Euler* discretization

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

FEniCS now solves, given the initial value u_0, a sequence of spatial stationary problems for u^{n+1} iteratively

$$u^0 = u_0, \tag{2.4}$$

$$u^{n+1} - \Delta t \Delta u^{n+1} = u^n + \Delta t f^{n+1}, \quad \text{for } n = 0, 1, 2, \ldots \tag{2.5}$$

To turn (2.5) into weak form, we have to multiply the equation by a test function $v \in V$ and integrate over the domain Ω where second-order derivatives are removed by Green's formula. We obtain the variational formulation

$$a(u^{n+1}, v) = L^{n+1}(v)$$

where

$$a(u^{n+1}, v) = \int_\Omega \left(u^{n+1} v + \Delta t \left\langle \nabla u^{n+1}, \nabla v \right\rangle\right) dx,$$
$$L^{n+1}(v) = \int_\Omega \left(u^n + \Delta t f^{n+1}\right) v \, dx.$$

We also need to approximate the IC (2.4). As mentioned in section 2.3.1, this is done by either interpolating the initial value u_0 or performing a L_2 projection of u_0 into the finite element space. Turning (2.4) into a variational formulation as well yields

$$a(u^0, v) = L^0(v)$$

where

$$a(u^0, v) = \int_\Omega u^0 v \, dx,$$
$$L^0(v) = \int_\Omega u_0 v \, dx.$$

Thus, we obtain the following sequence of variational problems in order to solve the heat equation with the FEM in FEniCS: For $n = 0, 1, 2, \ldots$

Find $u^0 \in U$ such that $a(u^0, v) = L^0(v) \quad \forall v \in V$.
Find $u^{n+1} \in U$ such that $a(u^{n+1}, v) = L^{n+1}(v) \quad \forall v \in V$

2.4 Application to Heat Equation

Since PINNs provide an adequate method for solving PDEs efficiently [4] we want to investigate whether they can be applied to certain simulation models e.g. a simplified thermal model of a satellite. A satellite lives in the earth orbit and absorbs heat from the sun or other sources via radiation. We are interested in modelling the heat conduction in the metal plates that have to be kept under a certain temperature. To this end, we consider the well-known heat equation. In this thesis, we are only looking at the one-dimensional case in order to expose first ideas how to deal with a generalization task when using PINNs.

Let $\Omega \subset \mathbb{R}$ be a bounded domain and $I = [0, T]$ be an interval. Then $u \colon \Omega \times I \to \mathbb{R}$ is a solution of the *heat equation* if

$$u_t = \lambda u_{xx} + f$$

with *thermal diffusivity* λ and *source term* $f : \Omega \to \mathbb{R}$. The solution u is unique if it further satisfies given boundary and initial conditions

$$\begin{aligned} u &= u_D \quad \text{on } \partial\Omega \\ u &= u_0 \quad \text{in } \Omega. \end{aligned}$$

For the sake of simplicity we only consider the heat equation with $f = 0$ and Dirichlet BCs u_D. The heat equation has for specified boundary and initial conditions a unique analytical solution which we are using for training.

Analytical Solution

The following is based on [37]. We exemplarily show the computation of the solution of the heat equation with $\Omega = [0, \pi]$, $\lambda = 1$, $f = 0$ and zero Dirichlet BCs $u_D = 0$ given by

$$\begin{aligned} u_t &= u_{xx} \quad \text{in } \Omega \times I \\ u &= 0 \quad \text{on } \partial\Omega \\ u(\cdot, 0) &= u_0 \quad \text{in } \Omega. \end{aligned}$$

We solve this PDE by separation of variables and use the ansatz

$$u(x, t) = X(x)T(t).$$

We immediately obtain $X(x)T'(t) = X''(x)T(t)$ which yields

$$\underbrace{\frac{T'(t)}{T(t)}}_{\text{constant in space}} = \frac{X''(x)}{X(x)}$$

assuming $X(x) \neq 0 \neq T(t)$. Let $c \coloneqq \frac{T'(t)}{T(t)}$ be a constant. Due to the boundary conditions $X(0) = X(\pi) = 0$ it follows

1) $c = 0$

$$\begin{aligned} X''(x) &= cX(x) \\ \implies X(x) &= a_1 x + a_2 \\ \implies a_1 &= a_2 = 0 \end{aligned}$$

2) $c > 0$

$$\begin{aligned} X''(x) &= cX(x) \\ \implies X(x) &= a_1 \mathrm{e}^{\sqrt{c}x} + a_2 \mathrm{e}^{-\sqrt{c}x} \end{aligned}$$

We obtain $a_1 + a_2 = 0$ and $a_1 \mathrm{e}^{\sqrt{c}\pi} + a_2 \mathrm{e}^{-\sqrt{c}\pi} = 0$.

$$\begin{aligned} &\implies a_1(\mathrm{e}^{\sqrt{c}\pi} - \mathrm{e}^{-\sqrt{c}\pi}) = 0 \\ &\implies a_1 = a_2 = 0 \end{aligned}$$

The above cases both lead to the trivial solution. Hence, $c < 0$. Let us denote $c = -k^2$ with $k \in \mathbb{N}$. Then

$$\begin{aligned} X''(x) &= -k^2 X(x) \\ \implies X(x) &= a_k \sin(kx). \end{aligned}$$

Note that $X(\pi) = 0$ for all $k \in \mathbb{N}$. The corresponding solution for $T(t)$ is given by

$$T(t) = \mathrm{e}^{-k^2 t}.$$

Thus, all functions

$$u_k(x,t) = a_k \sin(kx)\, \mathrm{e}^{-k^2 t}$$

are solutions of the one-dimensional heat equation on the interval $[0, \pi]$ with zero boundary conditions. By superposition we obtain the general solution

$$u(x,t) = \sum_{k=1}^{\infty} a_k \sin(kx)\, \mathrm{e}^{-k^2 t}\,.$$

The coefficients a_k are given by the initial condition u_0. The functions $\left\{\sqrt{\frac{2}{\pi}}\sin(kx) : k \in \mathbb{N}\right\}$ form an orthonormal basis of the Hilbert space of square integrable functions on $[0, \pi]$ with zero boundary conditions. Hence, the initial condition u_0 can be written as

$$\begin{aligned} u_0(x) &= \sum_{k=1}^{\infty} \frac{2}{\pi} \langle u_0, \sin(kx)\rangle \sin(kx) \\ &= \sum_{k=1}^{\infty} \frac{2}{\pi} \left(\int_0^{\pi} u_0(z) \sin(kz) dz \right) \sin(kx). \end{aligned}$$

It follows

$$a_k = \frac{2}{\pi} \left(\int_0^{\pi} u_0(z) \sin(kz) dz \right).$$

Thus, the analytical solution is given by

$$u(x,t) = \sum_{k=1}^{\infty} \frac{2}{\pi} \left(\int_0^{\pi} u_0(z) \sin(kz) dz \right) \sin(kx)\, \mathrm{e}^{-k^2 t}\,.$$

Example 2.1. Consider the one-dimensional heat equation with $\lambda = 1$ and zero Dirichlet boundary conditions $u_D = 0$. Let $u_0(x) = \sin(x)$ be the initial condition for $x \in [0, \pi]$. Then, the analytical solution is given by

$$u(x,t) = \sin(x)\, \mathrm{e}^{-t}\,.$$

Since we want our model to be generalizable, we look at the heat equation with various boundary and initial conditions. As already mentioned before (section 2.2.2), using PINNs with current state of the art methods, this requires the model to be trained anew for every modified boundary or initial condition. This constraint raises a fundamental question:

Can we construct a model which enables us to make reliable predictions for various problems without the necessity of training the model over and over again?

To answer this question, we take a look at several cases. First, we compare a PINN to a NN in terms of the size of the training dataset and extrapolation capabilities. With this, we want to expose if and under what conditions PINNs are superior over NNs. Subsequently, we vary the BCs for a fixed IC. Under this setting we also investigate the impact of an additional nonlinear term in the residual. The most interesting case we look at deals with a compilation of several ICs. Here, the BCs are given implicit by the ICs. In order to be able to describe the input functions with as few parameters as possible we ought to choose a proper representation. Hence, we parameterize the functions by Fourier coefficients.

2.4.1 Fourier Analysis

We want to verify the usage of Fourier coefficients as function representation by using ideas and results from [38, 39]. Therefore, we require some preliminaries.

Let $\mathbb{T} := [0, 1]$ be a *Torus* of length 1 and $L_2(\mathbb{T})$ be the Hilbert space of square-integrable functions $f\colon \mathbb{T} \to \mathbb{C}$ with inner product and norm as stated in section 2.3.1. Elements $f \in L_2(\mathbb{T})$ are periodic functions living on $\mathbb{R}$ that are continued periodically beyond the interval $\mathbb{T}$.

Theorem 2.2. (ONB of $L_2(\mathbb{T})$)
The functions

$$\left\{ \mathrm{e}^{2\pi \mathrm{i} kx} = \cos 2\pi kx + \mathrm{i} \sin \pi kx : k \in \mathbb{Z} \right\} \tag{2.6}$$

form an orthonormal basis (ONB) of $L_2(\mathbb{T})$.

Proof. See for example [38]. □

With (2.6), we can represent the elements of the Hilbert space $L_2(\mathbb{T})$ with respect to that basis. Let $S_n\colon L_2(\mathbb{T}) \to L_2(\mathbb{T})$ be the linear operator defined by

$$S_n f := \sum_{k=-n}^{n} \left\langle f, \mathrm{e}^{2\pi \mathrm{i} k \cdot} \right\rangle \mathrm{e}^{2\pi \mathrm{i} k \cdot}$$

which we denote by the *nth Fourier partial sum* of $f \in L_2(\mathbb{T})$. Note, that this is also defined for $f \in L_1(\mathbb{T}) \supset L_2(\mathbb{T})$ where $L_1(\mathbb{T})$ denotes the Hilbert space of measurable functions. Now, we can state the following theorem.

Theorem 2.3. (Fourier Series of a Function)
Every function $f \in L_2(\mathbb{T})$ has a unique representation of the form

$$f(x) = \sum_{k \in \mathbb{Z}} c_k(f)\, \mathrm{e}^{2\pi \mathrm{i} kx}, \quad c_k(f) := \left\langle f, \mathrm{e}^{2\pi \mathrm{i} k \cdot} \right\rangle = \int_0^1 f(x)\, \mathrm{e}^{-2\pi \mathrm{i} kx}\, dx \tag{2.7}$$

where the series $(S_n(f))_{n=0}^{\infty}$ converges in $L_2(\mathbb{T})$ to f. We call (2.7) *Fourier series* of f with *Fourier coefficients* $c_k(f)$. Further, the Parseval equation is fulfilled

$$\|f\|_{L_2(\mathbb{T})}^2 = \sum_{k\in\mathbb{Z}} |c_k(f)|^2 < \infty.$$

Proof. Let

$$\mathcal{P}_n := \left\{ \sum_{k=-n}^{n} c_k \, \mathrm{e}^{2\pi \mathrm{i} k \cdot} : c_k \in \mathbb{C} \right\}$$

be the space of all trigonometric polynomials of degree $\leq n$. We show that the Fourier partial sum operator $S_n \colon L_2(\mathbb{T}) \to L_2(\mathbb{T})$ is an orthogonal projector onto $\mathcal{P}_n$, i.e.

$$\|f - S_n f\| = \min\{\|f - p\| : p \in \mathcal{P}_n\}$$

for an arbitrary $f \in L_2(\mathbb{T})$. Let $f \in L_2(\mathbb{T})$ and $p_n \in \mathcal{P}_n$. Then, it holds

$$\begin{aligned}
\langle S_n f - f, p_n \rangle &= \langle S_n f, p_n \rangle - \langle f, p_n \rangle \\
&= \left\langle \sum_{k=-n}^{n} c_k(f) \, \mathrm{e}^{2\pi \mathrm{i} k \cdot}, \sum_{k=-n}^{n} c_k \, \mathrm{e}^{2\pi \mathrm{i} k \cdot} \right\rangle - \langle f, p_n \rangle, \quad c_k \in \mathbb{C} \\
&= \sum_{k=-n}^{n} c_k(f) \overline{c_k} \underbrace{\left\langle \mathrm{e}^{2\pi \mathrm{i} k \cdot}, \mathrm{e}^{2\pi \mathrm{i} k \cdot} \right\rangle}_{=1} - \sum_{k=-n}^{n} \overline{c_k} \left\langle f, \mathrm{e}^{2\pi \mathrm{i} k \cdot} \right\rangle \\
&= \sum_{k=-n}^{n} \overline{c_k} \left(\underbrace{c_k(f) - \left\langle f, \mathrm{e}^{2\pi \mathrm{i} k \cdot} \right\rangle}_{=0} \right) \\
&= 0 \qquad \forall p_n \in \mathcal{P}_n
\end{aligned}$$

Thus, $f - S_n f$ is orthogonal to $S_n f \in \mathcal{P}_n$. It immediately follows

$$\begin{aligned}
\|f - S_n f\|_{L_2(\mathbb{T})}^2 &= \|f\|_{L_2(\mathbb{T})}^2 - \|S_n f\|_{L_2(\mathbb{T})}^2 \\
&= \|f\|_{L_2(\mathbb{T})}^2 - \int_0^1 \left| \sum_{k=-n}^{n} c_k(f) \, \mathrm{e}^{2\pi \mathrm{i} k x} \right|^2 dx \\
&= \|f\|_{L_2(\mathbb{T})}^2 - \sum_{k=-n}^{n} |c_k(f)|^2 \int_0^1 \underbrace{\left| \mathrm{e}^{2\pi \mathrm{i} k x} \right|^2}_{=1} dx \\
&= \|f\|_{L_2(\mathbb{T})}^2 - \sum_{k=-n}^{n} |c_k(f)|^2 . \qquad (2.8)
\end{aligned}$$

Since the functions $\left\{ \mathrm{e}^{2\pi \mathrm{i} k x} : k \in \mathbb{Z} \right\}$ form an ONB of $L_2(\mathbb{T})$ the Parseval equation

holds

$$\|f\|_{L_2(\mathbb{T})}^2 = \sum_{k\in\mathbb{Z}} |c_k(f)|^2 .$$

As an immediate consequence the assertion follows from (2.8) for $n \to \infty$. □

Theorem 2.3 ensures that a function $f \in L_2(\mathbb{T})$ can be described completely by its Fourier coefficients. Note, that in (2.7) the equality is meant in the sense of L_2. There is no pointwise or uniform convergence here. The Fourier coefficients $c_k(f)$ correspond to the amplitudes of oscillations contained in f.

To be able to analyze a function in practical applications, it is crucial to have a discrete setting. Besides, in some cases only certain frequencies k are of interest. Therefore, consider the regarded function sampled on an equispaced grid, i.e. $(f_j)_{j=0}^{N-1}$ with $f_j := f\left(\frac{j}{N}\right)$, $N \in \mathbb{N}$.

Definition 2.10. [38] (Discrete Fourier Transform)
The linear map $\mathcal{F}\colon \mathbb{C}^N \to \mathbb{C}^N$, $\mathcal{F}f = \hat{f}$ which is defined by

$$\hat{f}_k := \sum_{j=0}^{N-1} f_j \, \mathrm{e}^{-2\pi i j k/N}, \quad k = 0, \ldots, N-1$$

is called *discrete Fourier transform* (DFT) of $f \in \mathbb{C}^N$.

The discretization induces an *aliasing error*. A proper approximation for the relation between the Fourier coefficients $c_k(f)$ and their discrete analogue $\hat{f}_k$ is given by

$$\begin{aligned} c_k(f) &\approx \frac{1}{N}\hat{f}_k, \qquad k = 0, \ldots, \frac{N}{2} - 1 \\ c_{-k}(f) &\approx \frac{1}{N}\hat{f}_{N-k}, \quad k = 1, \ldots, \frac{N}{2} \end{aligned}$$

where w.l.o.g let N be even. In the following, we identify the phrase Fourier coefficients with the values $\hat{f}_k$ computed by the DFT as stated in Definition 2.10). The DFT is invertible [38]. For given Fourier coefficients $\hat{f}_k$ the original sampled function can be obtained by the *inverse discrete Fourier transform*.

Definition 2.11. (Inverse Discrete Fourier Transform)
The linear map $\mathcal{F}^{-1}\colon \mathbb{C}^N \to \mathbb{C}^N$, $\mathcal{F}^{-1}\hat{f} = f$ defined by

$$f_j := \frac{1}{N} \sum_{k=0}^{N-1} \hat{f}_k \, \mathrm{e}^{2\pi i j k/N}, \quad j = 0, \ldots, N-1$$

is called *inverse discrete Fourier transform* (IDFT).

2.4.2 Comparison of PINN to NN

Before we tackle the generalization task of this thesis, we first look at a more basic question. What is the added value of a PINN compared to a NN which performs a simple function approximation by using the ground truth data? According to Raissi et al. PINNs seem to be the answer for problems where only small data is available [4]. Thus, we consider two cases

1. **Size of Training Dataset**
   How many training points are required in order to gain a reliable model?

2. **Extrapolation**
   Do PINNs have better extrapolation capabilities than NNs?

In order to answer this, we examine what impact the residual has on the training process of a PINN. This is done by comparing a PINN to a NN with the same network architecture and net initialization. Furthermore, we include points of the inner domain into the set of boundary points where the ground truth is used in the loss function. Generally, this would not be done when using PINNs in practice (see Def. 2.7). However, with this approach we achieve that the only difference between the two models is that the PINN has the additional residual term in the loss function which enables us to compare both methods.

Let $\Omega = [0, \pi]$, $I = [0, 1]$ and $\lambda = 1$. Consider a function $u\colon \Omega \times I \to \mathbb{R}$ satisfying the one-dimensional heat equation with the following initial and boundary conditions

$$\begin{aligned} \text{IC: } & u_0(x) = \sin(x) \\ \text{BC: } & u_D = 0. \end{aligned} \tag{2.9}$$

As in section 2.2 we define $\tilde{\Omega} = \Omega \times I$ with $\partial\tilde{\Omega} = \partial\Omega \times (0, T) \cup \Omega \times \{0\}$ containing boundary and initial data. We consider the set of training points $\mathcal{T} \subset \tilde{\Omega}$. In section 2.2.1 it is claimed that $\mathcal{T}$ is divided into sets $\mathcal{T}_f \subset \tilde{\Omega}$ and $\mathcal{T}_b \subset \partial\tilde{\Omega}$ where ground truth data is only used for $\mathcal{T}_b$. However, as mentioned above, for comparison reasons we set $\mathcal{T}_b = \mathcal{T}$. The ground truth data is obtained by solving the PDE with FEniCS (see subsection 2.3.2) on a 100×100 mesh, i.e. a grid consisting of 100 points in x- and 100 points in t-direction where $x \in \Omega$, $t \in I$.

Tackling the first question we train the models with different sizes of the training dataset. We consider sizes in $M = \{10, 15, 20, 30, 40, 60, 80, 100, 120, 160, 200\}$. First, we train the NN with a selected size of randomly distributed training points. Then, with the same net initialization and the same training data we train the corresponding PINN. The test dataset consists of 200 randomly distributed points. For

all considered cases we use the same test dataset. With the IC and BC from (2.9) we obtain the settings for each considered size of the training dataset as depicted in Table 2.1. With the obtained models we perform extrapolation and look at the behavior for the extreme case $t \in [1, 5]$.

Table 2.1: Setting for comparison of PINN to NN. Both were trained with the same net initialization on the same data with the Adam optimizer and a learning rate of 0.0001.

Ω	I	λ	Depth	Width	Epochs
$[0, \pi]$	$[0, 1]$	1.	5	50	20000

2.4.3 Generalization of PINNs

The quality of a NN is usually measured by its generalization ability. In fact, one can consider a model as useless if it makes inaccurate predictions on unseen data. Thus, we investigate the generalization capabilities of PINNs for various problem settings of the heat equation. We consider two main cases

1. **Fixed Initial and Various Boundary Conditions**
   a) Sine as IC
   b) Line as IC with discontinuities at the boundary
   c) Nonlinear reaction term in residual with sine as IC
2. **Various Boundary and Initial Conditions**
   d) Functions parametrized by Fourier coefficients

In the following we elaborate on the cases mentioned above. In general, the heat equation is a special case of the *diffusion-reaction system*

$$u_t = \nabla \cdot (d(u)\nabla u) + r(u)$$

where $d(u) > 0$ is the diffusivity and $r(u) > 0$ a reaction term. In the cases considered before, it holds $d = \lambda$ and $r = 0$. However, regarding the motivation to model a reaction of the satellite with the space environment, for the nonlinear case we take an additional radiative loss into account. In a vaccuum, a satellite can only absorb or emit heat via radiation. We model this radiative loss in a simple fashion by the *Stefan-Boltzmann law* $r(u) = -\sigma(u^4 - v^4)$ where $v(x, t)$ denotes the temperature of the surroundings and σ being a constant. For simplicity we restrict

to $v = 0$. In doing this, we investigate the PINN's behavior when the residual contains an additional nonlinear term.

To summarize, we are looking for a solution $u\colon \Omega \times I \to \mathbb{R}$ with $\Omega, I \subset \mathbb{R}$ that satisfies the PDE

$$u_t = \lambda u_{xx} + r(u)$$

for given IC u_0 and BC u_D. For all cases we consider $I = [0, 1]$.

a) Let $\Omega = [0, \pi]$, $r = 0$ and

$$\text{IC: } u_0(x) = \sin(x)$$

$$\text{BC: } u_D(x) = \begin{cases} a, & \text{for } x = 0 \\ b, & \text{for } x = \pi \end{cases}.$$

We choose $a, b \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$. This results in a dataset consisting of 36 individual datasets.

b) Let $\Omega = [0, 1]$, $r = 0$ and

$$\text{IC: } u_0(x) = -x + 1$$

$$\text{BC: } u_D(x) = \begin{cases} a, & \text{for } x = 0 \\ b, & \text{for } x = 1 \end{cases}.$$

As in a) we choose $a, b \in [0, 1]$ with stepsize 0.2 and obtain a dataset consisting of 36 individual datasets. The discontinuities at the boundaries force the straight line to change over time in a way such that it fulfills the BC.

c) Let $\Omega = [0, \pi]$, $a, b \in [0, 1]$ and the IC and BC as defined in a). Unlike in a), b), here it holds $r \neq 0$. The considered PDE becomes

$$u_t = \lambda u_{xx} - \sigma u^4.$$

We again obtain a dataset consisting of 36 individual datasets.

d) Let $\Omega = [0, 1]$ and $r = 0$. The ICs are given by functions that are parametrized by 6 Fourier coefficients (see Def. 2.10). The BCs are given implicit. Here, we provide the PINN 2065 functions. Thus, the dataset consists of 2065 individual datasets. The functions are variations of rectangular functions, lines with different slopes and polynomials. In doing so, we achieve that the network

learns various types of functions such that it is able to predict the solution for an arbitrary IC represented by its Fourier coefficients.

As usual for PINNs, we define a set of training points $\mathcal{T} \subset \tilde{\Omega}$ with $\mathcal{T}_f \subset \tilde{\Omega}$ and $\mathcal{T}_b \subset \partial\tilde{\Omega}$. This is done for each individual dataset. Consider case a): If we choose a size of 20 training points, each of the 36 individual datasets uses 20 training points where 10 points live in the inner region an the other 10 on the boundary. The 10 boundary points are also used for the residual loss in order to increase the regularization. Thus, the complete training dataset consists of 720 points. The same procedure is done for the test dataset. Here, for each case we use a size of 200 test points. The different problem settings with the respective network architectures are

Table 2.2: Settings for the generalization of PINNs. The column *Size* denotes the size of the training dataset for each individual dataset. In the nonlinear case we choose $\sigma = 3.7$. All models were trained with the Adam optimizer and a learning rate of 0.0001.

	Sizes	Ω	I	λ	r	Depth	Width	Epochs
Sine	20, 40, 60	$[0, \pi]$	$[0, 1]$	1.	0	5	60	60000
Line	40, 60, 80	$[0, 1]$	$[0, 1]$	0.4	0	5	100	20000
Nonlinear	30, 40, 60	$[0, \pi]$	$[0, 1]$	0.1	$-\sigma u^4$	5	100	60000
Fourier	40, 60, 80	$[0, 1]$	$[0, 1]$	0.2	0	5	60	60000

summarized in Table 2.2. We use Python's `scipy` module to compute the complex-valued Fourier coefficients $\hat{f}_k$. We sample the function used as IC at 100 points and take 6 coefficients as its parametrization. We are interested in those coefficients with higher frequencies, i.e. c_0, c_1, c_2, c_{-1}, c_{-2}, c_{-3}. Thus, we have to use $\hat{f}_0$, $\hat{f}_1$, $\hat{f}_2$, $\hat{f}_{97}$, $\hat{f}_{98}$, $\hat{f}_{99}$ according to the relation mentioned above. For simplicity reasons we only use the real and imaginary part of the numbers instead of their complex representation. As before, we use FEniCS to generate the ground truth data for $\mathcal{T}_b$. Therefore, we use the 6 Fourier coefficients and compute the resulting function using the IDFT.

3 Results

We finally present the results of the tasks presented in section 2.4. First, we show the results when fixing the boundary and initial condition. Here, we show the comparison between a NN and a PINN. In the next section, we look at the effects of selected initial conditions and nonlinearities in the residual when varying the boundary conditions. Lastly, we present the results gained when training the model with various initial conditions provided as six Fourier coefficients. The results are generated on a 100×100 grid.

3.1 Fixed Boundary and Initial Condition

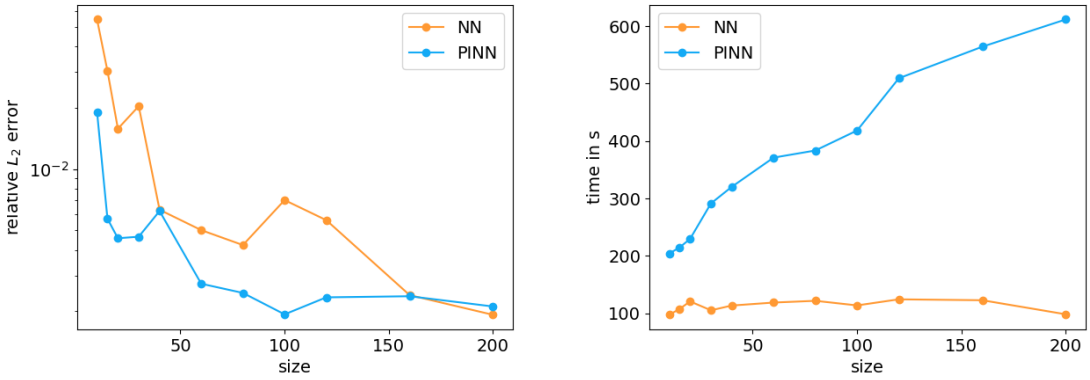
3.1.1 Size of Training Dataset

We compare a PINN to a NN with the same prerequisites. We are interested in the models' performance for increasing size of the training dataset. In the following the phrase size corresponds to the number of training points. For each size we present the model at that training step where it achieved the best test loss since the test loss serves as indicator for a good generalization ability. An overview of the obtained results for each considered size is displayed in Table 3.1. For all models the test loss was evaluated at the same 200 test points. The relative L_2 error takes all 10000 grid points into account. The relative L_2 error of the PINN is smaller than the error of the NN for all sizes up to 160 inclusively. The PINN test loss is smaller than the NN test loss up to size 100. Looking at specific sizes, different behaviors of the NN and the PINN, respectively, can be observed. From size 30 to 40 the test loss and the relative L_2 error of the NN decrease whereas for the PINN these values are increasing or being equal. From size 80 to 100 the test loss and the error of the NN are increasing, for the PINN they are decreasing. From size 100 to 120, the relative L_2 error of the NN is decreasing whereas for the PINN it is increasing.

A visualization of the evolution of the relative L_2 error and the runtime over the sizes can be seen in Figure 3.1. Consider the errors displayed in Figure 3.1a. At first

Table 3.1: Comparison of PINNs and NNs trained for various numbers of random sampled training points. Train loss, test loss and relative L_2 error are compared. The column size denotes the number of training points. The test losses of all models are evaulated at the same 200 points.

	NN			PINN		
size	train loss	test loss	rel. L_2 error	train loss	test loss	rel. L_2 error
10	1.99e-06	4.24e-04	5.45e-02	6.71e-06	9.1e-05	1.91e-02
15	4.85e-06	2.67e-04	3.04e-02	4.5e-07	8.e-06	5.7e-03
20	1.8e-07	2.23e-05	1.58e-02	4.e-07	5.1e-06	4.57e-03
30	7.78e-06	5.1e-05	2.04e-02	1.3e-06	4.3e-06	4.65e-03
40	1.25e-06	4.38e-06	6.31e-03	2.9e-06	4.3e-06	6.22e-03
60	1.1e-06	2.92e-06	5.01e-03	1.2e-06	1.4e-06	2.73e-03
80	8.8e-07	1.77e-06	4.23e-03	1.2e-06	1.5e-06	2.46e-03
100	5.7e-07	4.15e-06	7.04e-03	4.5e-07	1.e-06	1.93e-03
120	1.86e-06	6.94e-06	5.61e-03	7.3e-07	1.5e-06	2.34e-03
160	1.e-06	1.27e-06	2.4e-03	1.2e-06	1.3e-06	2.37e-03
200	5.7e-07	8.3e-07	1.92e-03	9.e-07	1.e-06	2.11e-03



(a) Relative L_2 error **(b)** Runtime

Figure 3.1: Plot of relative L_2 error and runtime over sizes. (a): The relative L_2 error is evaluated at the best model. (b): The runtime corresponds to a training of 20000 epochs.

glance, it can be seen that the PINN error is smaller than the NN error for sizes up to 160 inclusively. The largest difference between the error of the PINN and the NN is at size 15. In Figure 3.1b the runtime of the training process for each considered size is displayed. For the PINN the runtime increases over the sizes whereas the runtime of the NNs stays within a range of about 97 and 124 seconds. At size 10 the PINN takes twice as long as the NN. For 200 training points the PINN's runtime is

about 6 times larger than the runtime of the NN.

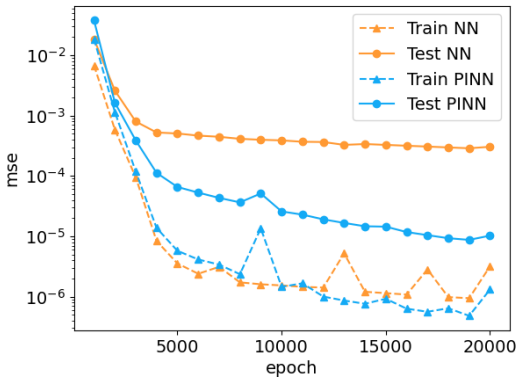
Loss Curves

We exemplarily look at the loss curves of the PINNs and NNs trained for sizes in $\{15, 30, 40, 80, 100, 160\}$ pictured in Figure 3.2. The losses for size 15 are shown in Figure 3.2a. One can see the large difference between the train and test losses for both the PINN and NN. This indicates a lack of train data which corresponds to a insufficient regularization and thus results in a poor generalization. For the NN this gap is larger and there is no significantly decrease of test loss from epoch 4000 on, whereas for the PINN it decreases up to epoch 19000. For size 30 (Fig. 3.2b), the gap between the PINN's train and test loss is small compared to the NN where it is still large. Here, the NN's test loss stops decreasing significantly at epoch 13000. From size 40 on (Fig. 3.2c), the difference of train and test loss is for both PINN and NN small compared to the smaller sizes. All curves are decreasing over the number of epochs. One can see in Figures 3.2a-d that the gap between train and test loss becomes smaller for increasing size of the training dataset. This gap becomes larger again for size 100 as can be seen in Figure 3.2e. Figure 3.2f displays the similar loss values of both the PINN and NN.

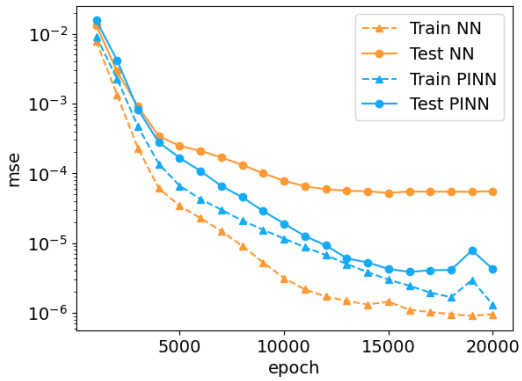
Solution Plots

Now, we present the solution plots that correspond to the losses presented before. In Figure 3.3 the solution plots of NNs and PINNs trained with sizes in $\{15, 30, 40, 80, 100, 160\}$ are compared. For each size the prediction of the model and the used training data are displayed. Additionally, we look at the relative L_∞ error evaluated at each grid point in order to see where the errors are made. Furthermore, we depict the maximal errors of all considered models in Table 3.2.

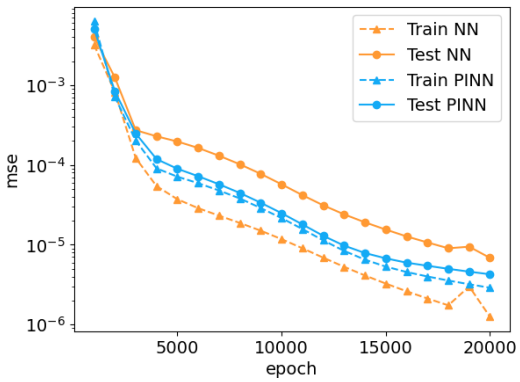
In case of size 15 (Fig. 3.3a-b) one can see that in the case of the NN the errors are especially made on the boundary and in those regions where no training points were available. The PINN has in the aforementioned areas only minor errors. In case of size 30, pictured in Figures 3.3c-d, the larger errors made by the NN are located in areas of the boundary, especially around the initial condition where no training points are available. The PINN has only a maximal error of 1.95% although there are no training points in that region. Also, compared to the PINN the NN makes larger errors in the inner domain. In Figures 3.3e-f, representing 40 utilized training points, the error plots of PINN and NN have a similar ratio between large and smaller errors. Comparing size 80 (Fig. 3.3g-h) to size 100 (Fig. 3.3i-j), one can see in the case of the NN that for size 100 larger errors are made than for size



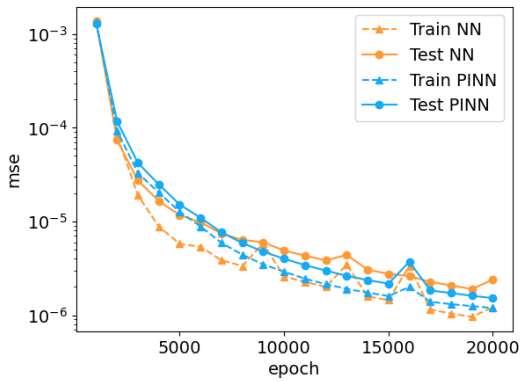
(a) Size 15



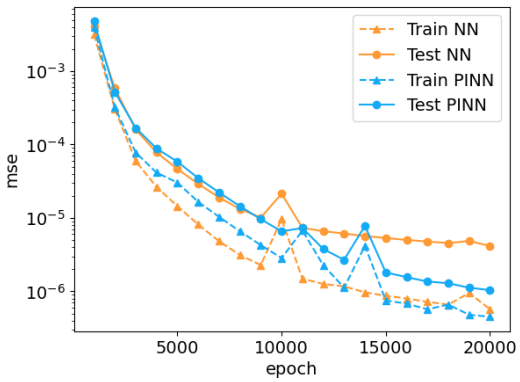
(b) Size 30



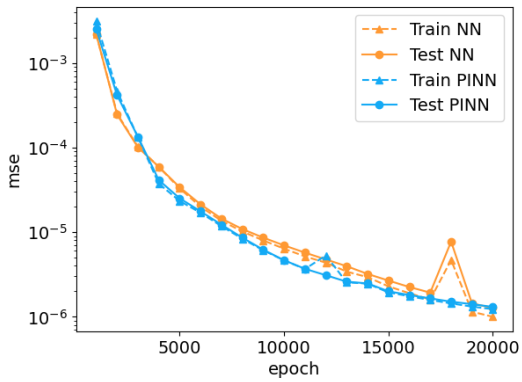
(c) Size 40



(d) Size 80

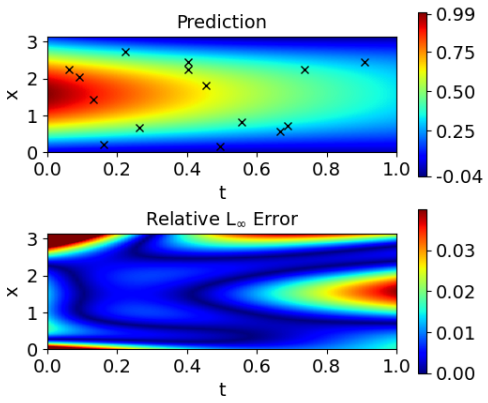


(e) Size 100

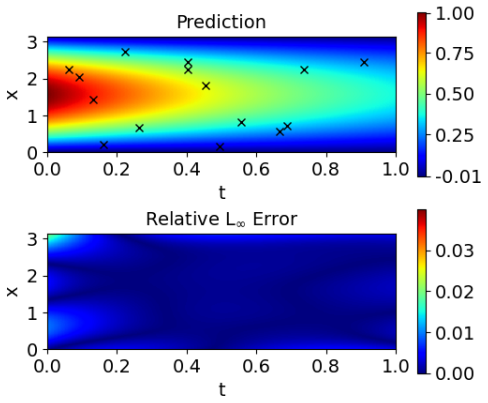


(f) Size 160

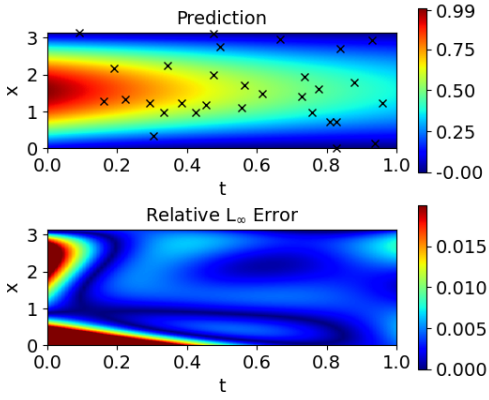
Figure 3.2: Loss curves of PINN and NN over number of epochs for sizes in $\{15, 30, 40, 80, 100, 160\}$. (a): Large gap between train and test loss for both PINN and NN. For the NN this gap is larger. (b): Gap between train and test loss still large for NN but smaller for PINN. (c)-(d): For growing size the loss curves are approaching each other for both PINN and NN. (e): Difference between train and test loss increases again (for NN larger difference). (f): Loss values of PINN and NN similar for almost all epochs.



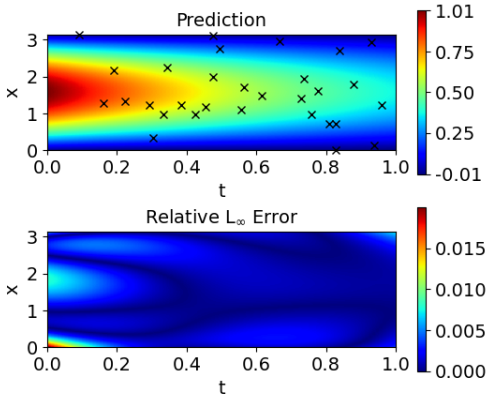
(a) NN: Size 15



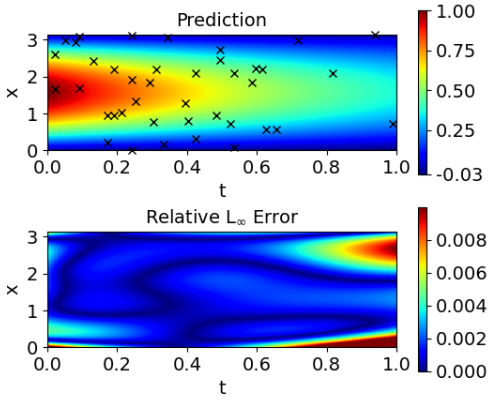
(b) PINN: Size 15



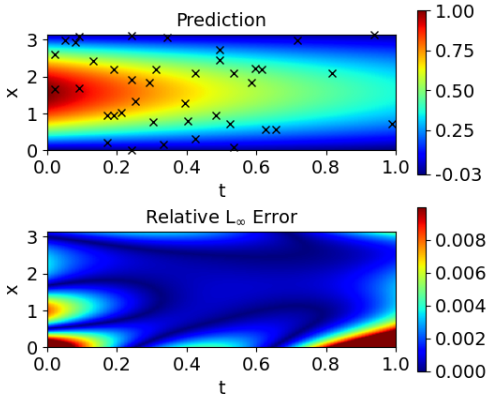
(c) NN: Size 30



(d) PINN: Size 30

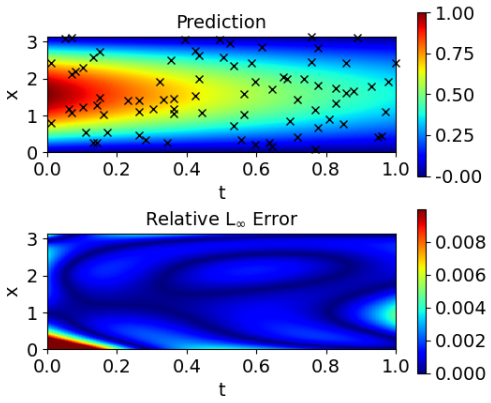


(e) NN: Size 40

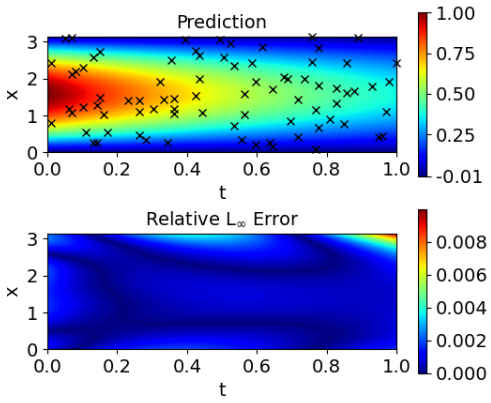


(f) PINN: Size 40

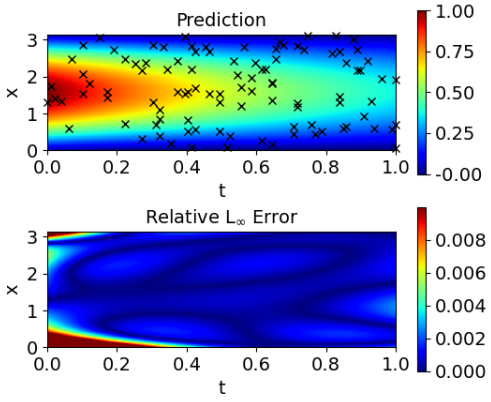
Figure 3.3: Predicted solutions of PINNs and NNs trained with sizes in $\{15, 30, 40\}$. The black markers represent the used training points. *Top*: Prediction. *Bottom*: Relative L_∞ error. (a)-(d): NN makes larger errors than PINN. Significant errors are made at the boundaries. (e)-(f): Both are making similar errors. Errors are mainly made at the boundaries.



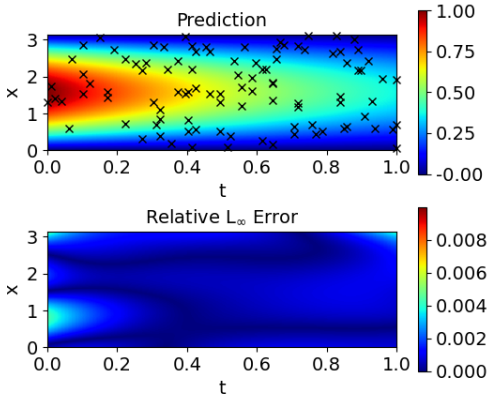
(g) NN: Size 80



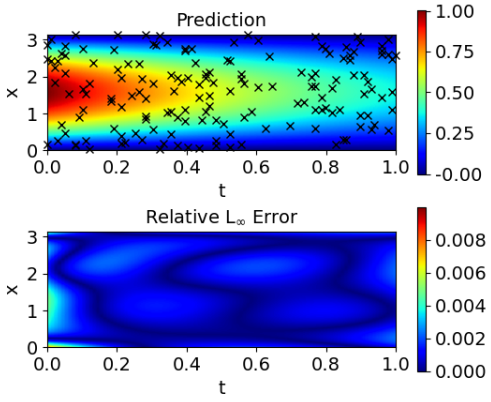
(h) PINN: Size 80



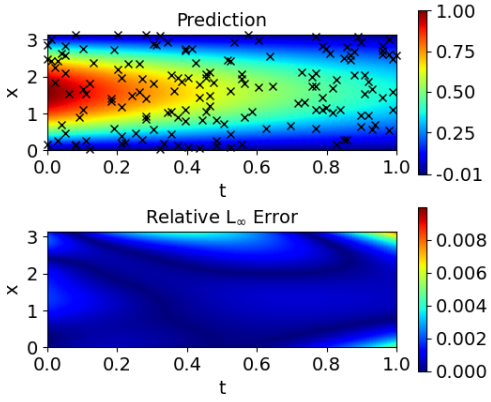
(i) NN: Size 100



(j) PINN: Size 100



(k) NN: Size 160



(l) PINN: Size 160

Figure 3.3: Predicted solutions of PINNs and NNs trained with sizes in $\{80, 100, 160\}$. The black markers represent the used training points. *Top*: Predicted solution. *Bottom*: Relative L_∞ error. (g)-(j): NN makes larger errors than PINN. Errors are mainly made at the boundaries. (k)-(l): Both are making similar errors. Minor errors are made at the boundaries.

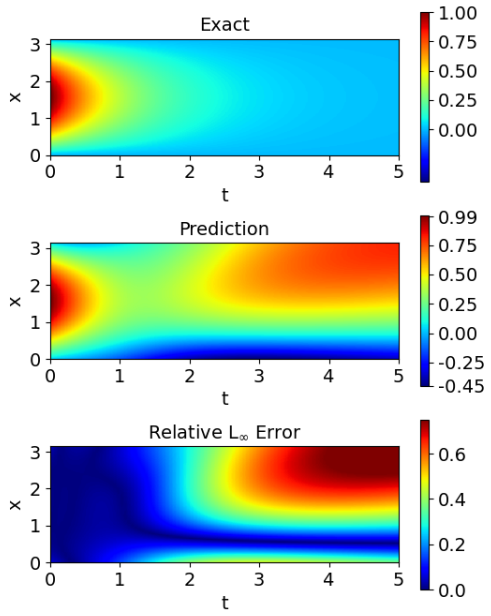
80. This can be seen by the increasing of the error at the boundary in positive t-direction. In contrast to the NN, the PINNs trained with 80 and 100 training points, respectively, have a similar error distribution with a maximal error of 0.98% for size 80 and 0.45% for size 100. In case of size 160 (Fig. 3.3k-l), the PINN and NN behave similar in the sense of the amount of errors. Both are making minor errors at the boundaries although the locations of the errors differ. Also, the inner domain of the PINN has a smoother error distribution than the NN having isolated spots.

Table 3.2: Predictive errors of PINNs and NNs for various number of training samples. The left column corresponds to the size of the training dataset, the maximal relative L_∞ error with locations are displayed.

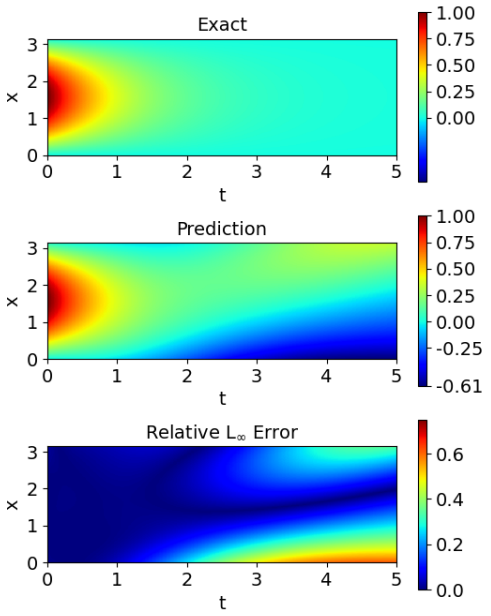
	NN		PINN	
size	$(x,t)^T$	max. error	$(x,t)^T$	max. error
10	$(0,1)^T$	14.5%	$(0,1)^T$	4.5%
15	$(\pi,0)^T$	10.86%	$(\pi,0)^T$	2.05%
20	$(0,0)^T$	7.77%	$(\pi,1)^T$	1.43%
30	$(0,0)^T$	10.61%	$(0,0)^T$	1.95%
40	$(0,1)^T$	2.72%	$(0,1)^T$	2.7%
60	$(0,0)^T$	3.32%	$(\pi,1)^T$	0.89%
80	$(0,0)^T$	2.92%	$(\pi,1)^T$	0.98%
100	$(0,0)^T$	4.87%	$(\pi,0)^T$	0.45%
120	$(\pi,0)^T$	2.38%	$(0.63,0)^T$	0.57%
160	$(0,0)^T$	0.67%	$(\pi,1)^T$	0.83%
200	$(\pi,0.27)^T$	0.54%	$(\pi,1)^T$	0.78%

3.1.2 Extrapolation

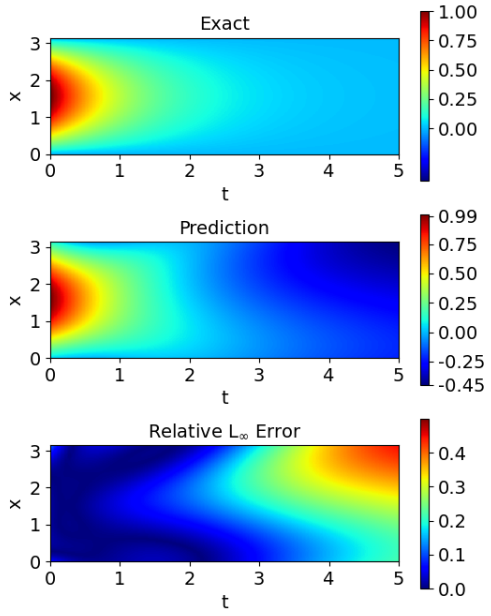
In terms of extrapolation we depict the models trained with sizes in $\{10, 15, 40, 80, 100\}$ in Figure 3.4. Generally, it can be said that the error increases in positive t-direction. For completeness we additionally display the maximal errors of the extrapolation performed with all regarded models in Table 3.3. In case of size 10 (Fig. 3.4a-b), the PINN achieves better extrapolation results than the NN. The error of the NN increases up to 81.48%. In case of size 15 (Fig. 3.4c-d), both are making a similar amount of errors. In Figures 3.4e-f it can be seen that the NN extrapolates better than the PINN. Here, the NN has a maximal error of 47.12% compared to the PINN's error of 57.09%. Looking at size 80 (Fig. 3.4g-h), the PINN achieves better



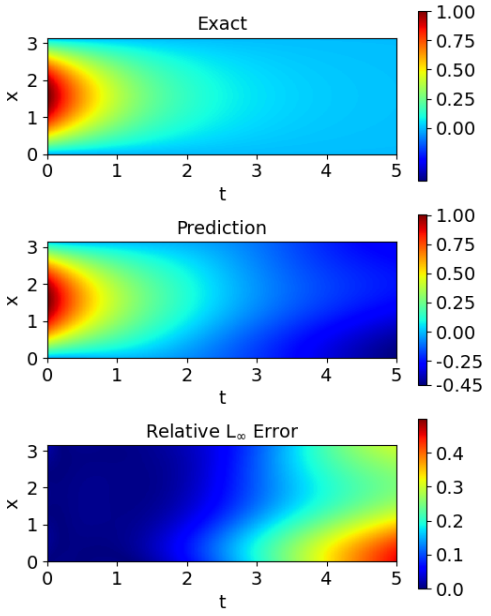
(a) NN: Size 10



(b) PINN: Size 10

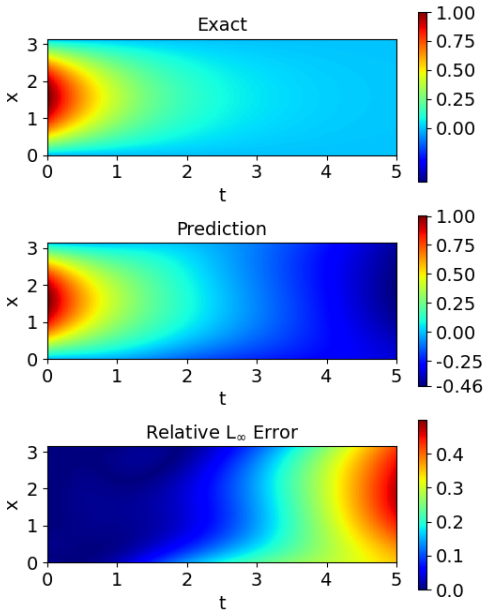


(c) NN: Size 15

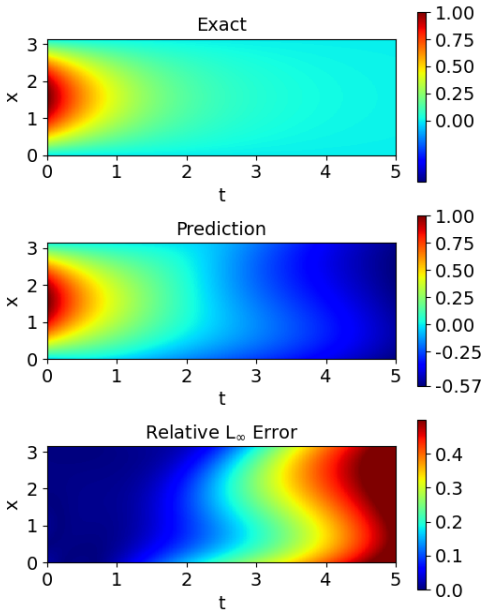


(d) PINN: Size 15

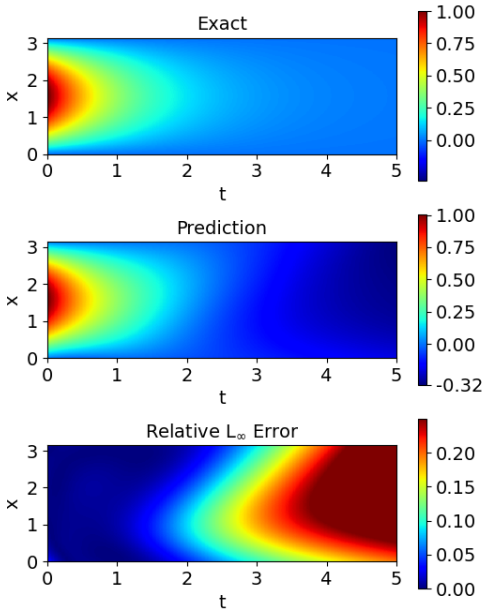
Figure 3.4: Extrapolated solution of PINNs and NNs trained with sizes in $\{10, 15\}$. The models were trained for $t \in [0, 1]$. *Top*: Exact solution. *Middle*: Predicted solution. *Bottom*: Relative L_∞ error. (a)-(b): NN makes larger errors than PINN. (c)-(d): Both are making similar errors.



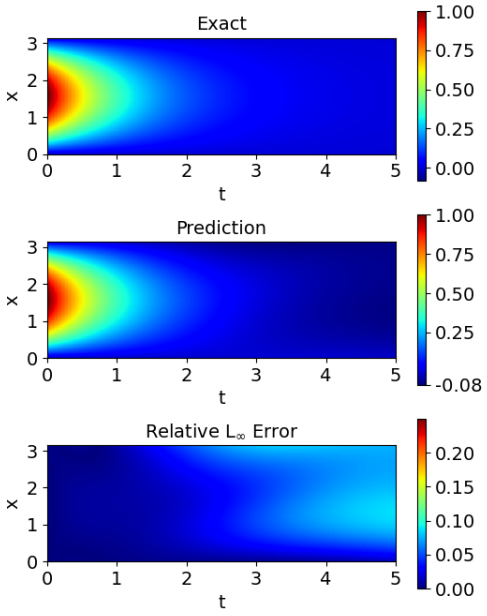
(e) NN: Size 40



(f) PINN: Size 40

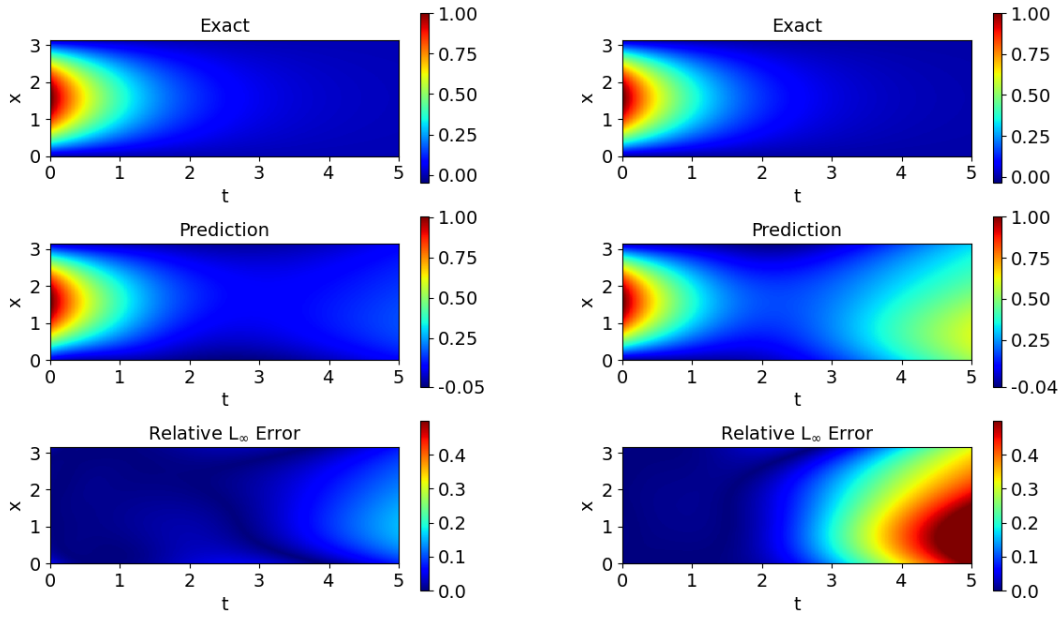


(g) NN: Size 80



(h) PINN: Size 80

Figure 3.4: Extrapolated solution of PINNs and NNs trained with sizes in $\{40, 80\}$. The models were trained for $t \in [0, 1]$. *Top*: Exact solution. *Middle*: Predicted solution. *Bottom*: Relative L_∞ error. (e)-(f): PINN makes larger errors than NN. (c)-(d): NN makes larger errors than PINN.



(i) NN: Size 100 **(j)** PINN: Size 100

Figure 3.4: Extrapolated solution of PINN and NN trained with 100 training points. The models were trained for $t \in [0, 1]$. *Top*: Exact solution. *Middle*: Predicted solution. *Bottom*: Relative L_∞ error. (i)-(j): PINN makes larger errors than NN.

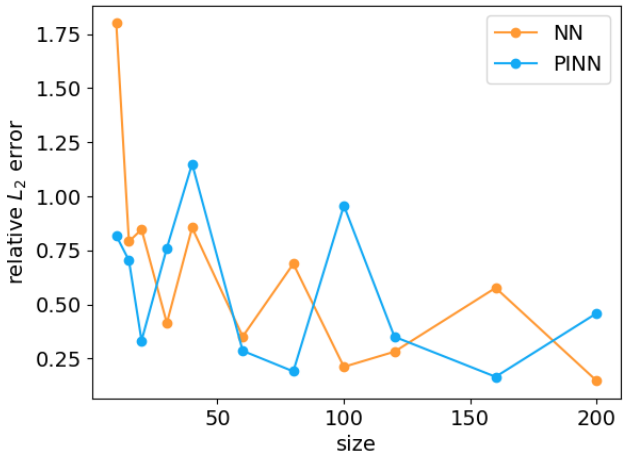


Figure 3.5: Relative L_2 error for extrapolation performed with the models trained with different sizes of the training dataset.

extrapolation results with only a small maximal error of 8.64%. The error of the NN increases up to 31.7%. In case of size 100 (Fig. 3.4i-j), the NN is superior over the PINN. The maximal error of the PINN is of 59.23% whereas the NN has only

an error of 15.37%.

In Figure 3.5 we plotted the relative L_2 error over the number of training samples. One can see great fluctuations of the error for growing size of the training dataset. Thus, there is no specific rule that can be deduced under what assumptions a NN or PINN performs better in terms of extrapolation.

Table 3.3: Predictive errors of extrapolation performed with PINN and NN models trained for various number of training samples. The left column corresponds to the size of the training dataset, the relative L_∞ error with locations are depicted.

	NN		PINN	
size	$(x,t)^T$	max. error	$(x,t)^T$	max. error
10	$(\pi,5)^T$	81.48%	$(0,4.75)^T$	60.66%
15	$(\pi,5)^T$	44.76%	$(0,5)^T$	45.23%
20	$(0,5)^T$	55.03%	$(0,5)^T$	21.8%
30	$(\pi,5)^T$	33.25%	$(0.98,5)^T$	45.47%
40	$(1.87,5)^T$	47.12%	$(2.38,5)^T$	57.09%
60	$(2.82,5)^T$	19.05%	$(1.75,5)^T$	18.61%
80	$(2.82,5)^T$	31.7%	$(1.3,5)^T$	8.64%
100	$(1.02,5)^T$	15.37%	$(0.63,5)^T$	59.23%
120	$(0,5)^T$	19.34%	$(0,5)^T$	37.18%
160	$(2.16,5)^T$	37.98%	$(0,5)^T$	14.91%
200	$(0,4.8)^T$	9.29%	$(1.33,5)^T$	26.46%

3.1.3 Validation with DeepXDE

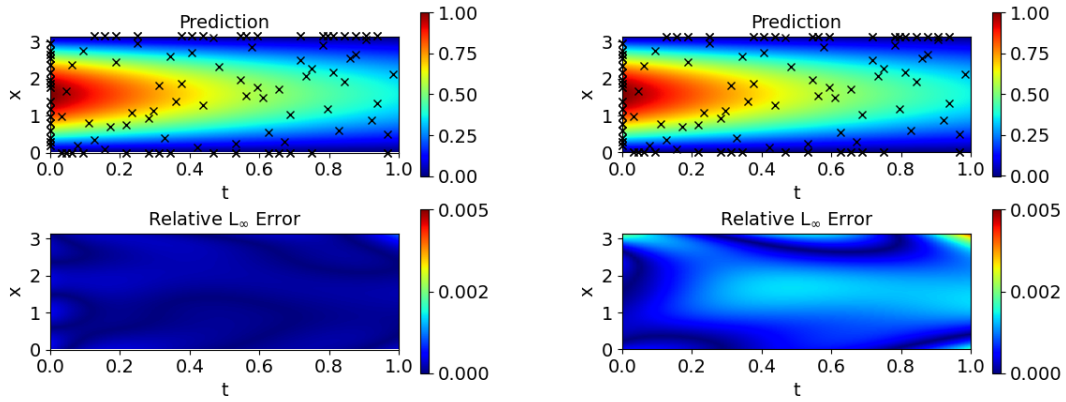
Within this section we validate our implementation using PyTorch with the library DeepXDE mentioned in section 2.2.2. This is done in order to exploit advantages or disadvantages of using DeepXDE compared to a self-made PINN implementation. Although DeepXDE provides the ability to use different backends, it fails to generate results with the backend PyTorch since relevant functions have not been implemented yet. Thus, for the following results we use the default backend TensorFlow 1.x. The network architecture is the same as the one used for the comparison of NN and PINN (see Tab. 2.1).

In Figure 3.6 the comparison of the predicted solutions is pictured. One can see that smaller errors are achieved by DeepXDE with a maximal error of 0.13% at $(x,t)^T = (\pi,1)^T$. Our self-made implementation has a maximal error of 0.42% also at $(x,t)^T = (\pi,1)^T$. Looking at Table 3.4 it can be seen that DeepXDE is about

two times faster than the self-made implementation. Also, the relative L_2 error of DeepXDE is one order of magnitude smaller.

Table 3.4: Comparison of the training process of a PINN using DeepXDE and PyTorch. Both were trained for 20000 epochs on the same train and test data with the Adam optimizer and a learning rate of 0.0001.

	train loss	test loss	time	rel. L_2 error
DeepXDE	1.04e-06	1.18e-06	112.8s	3.56e-04
Torch	2.42e-06	3.79e-06	273.3s	2.33e-03



(a) DeepXDE **(b)** PyTorch

Figure 3.6: Comparison of DeepXDE with own PINN implementation using PyTorch. *Top*: Predicted solution. The black markers represent the used training data (100 data points). *Bottom*: Relative L_∞ error.

3.2 Fixed Initial and Various Boundary Conditions

We now address the generalization task which is the core of this thesis. At the moment we are looking at models trained with fixed initial conditions and boundary conditions with values in $M := \{0, 0.2, 0.4, 0.6, 0.8, 1\}$. Within this section, we also include investigations on the behavior of the training process when the PDE residual contains an additional nonlinear term. The following results are compmuted on a Tesla V100 SXM2 unless othwise stated.

3.2.1 Initial Condition: Sine

The loss curves of models trained with sine as initial condition and training set sizes in $\{20, 40, 60\}$ are presented in Figure 3.7. For size 20 one can see a large gap between train and test loss. This indicates a poor regularization and yields a model that does not generalize well. Comparing size 40 to size 60, there is no significantly difference of the test loss curves. In Table 3.5 we have displayed the runtime of

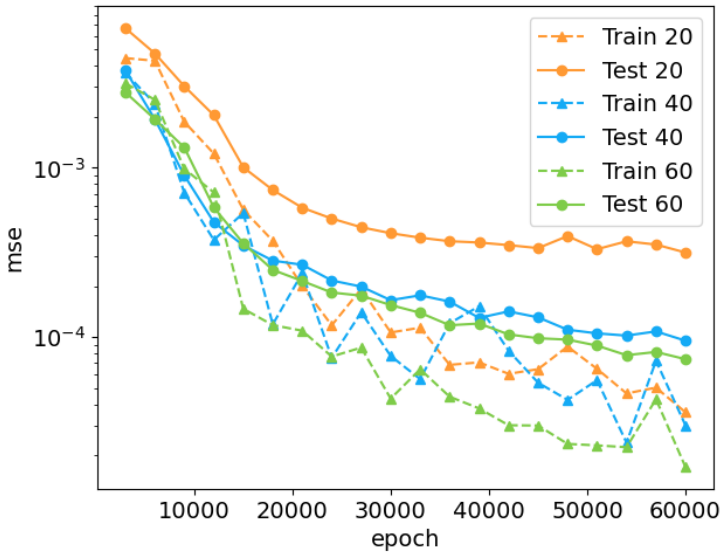


Figure 3.7: Plot of train and test losses of models trained with sine as IC and boundary values in $\{0, 0.2, 0.4, 0.6, 0.8, 1\}$. The models were trained for sizes in $\{20, 40, 60\}$ and 60000 epochs with Adam as optimizer and a learning rate of 0.0001.

each model and the train and test loss values of the respective best model, i.e. the model at that training step which achieves the best test loss. For the given problem setting the best test loss is achieved by size 60. The runtime increases linearly with

Table 3.5: Runtime of the models with sine as IC trained for 60000 epochs with sizes in $\{20, 40, 60\}$. Additionally, the train and test loss values of the best models are depicted.

size	runtime	train loss	test loss
20	15.3h	1.2e-04	3e-04
40	17.8h	3e-05	9.5e-05
60	19.5h	4e-05	7.35e-05

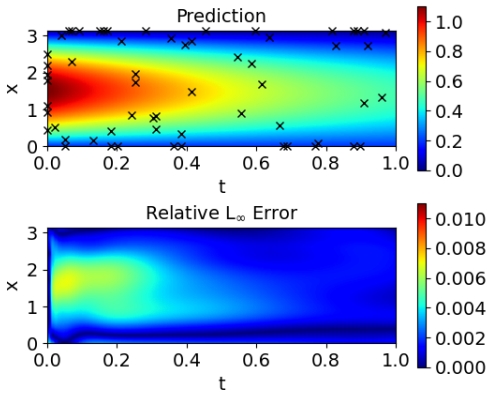
Table 3.6: Predictive errors of the model trained for solving the one-dimensional heat equation with sine as IC, boundary values in $\{0, 0.2, 0.4, 0.6, 0.8, 1\}$ and size 60. The two left columns represent the left (l) and right (r) boundary values, respectively. Additionally, the maximal relative L_∞ error with locations and the overall relative L_2 error are depicted.

l	r	$(x,t)^T$	max. error	rel. L_2 error
0.2	0.0	$(1.65, 0.05)^T$	0.7%	5.28e-03
0.3	0.0	$(1.65, 0.05)^T$	0.74%	5.51e-03
0.4	0.0	$(1.62, 0.05)^T$	0.74%	5.38e-03
0.4	0.4	$(1.52, 0.02)^T$	0.68%	4.47e-03
0.4	0.7	$(1.55, 0.02)^T$	0.62%	3.85e-03
0.4	0.9	$(1.59, 0.06)^T$	0.57%	3.24e-03
0.5	0.0	$(1.62, 0.05)^T$	0.69%	4.95e-03
0.6	0.7	$(1.52, 0.01)^T$	0.57%	3.64e-03
0.8	0.0	$(0.29, 0.03)^T$	0.46%	2.38e-03
0.8	0.3	$(1.55, 0.02)^T$	0.52%	3.41e-03
0.8	0.7	$(1.52, 0.01)^T$	0.53%	3.02e-03
1.0	0.7	$(1.59, 0.01)^T$	0.54%	2e-03
1.0	0.9	$(1.62, 0.01)^T$	0.52%	1.47e-03

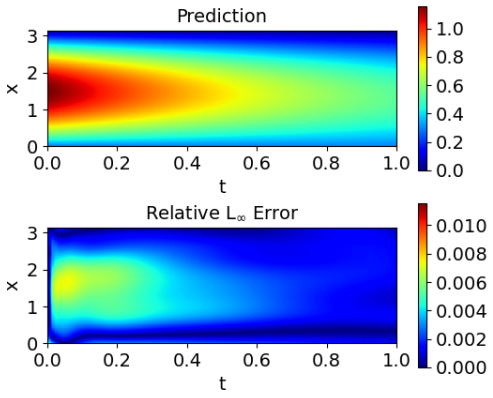
growing size of the training dataset. For size 60 the training process has a runtime of about 19.5 hours.

We are looking at the results obtained by the model trained with size 60. We predict the solutions of the one-dimensional heat equation with BCs in $[0, 1]$ and stepsize 0.1 resulting in 121 cases. Some examples of predicted solutions with their relative L_∞ error are pictured in Figure 3.8. In Table 3.6, we additionally show the maximal relative L_∞ error and their locations, as well as the relative L_2 error. For better readibility, in the following we refer by maximal error to the maximal relative L_∞ error and by overall error to the relative L_2 error. By l, r we denote the left and right boundary value, respectively.

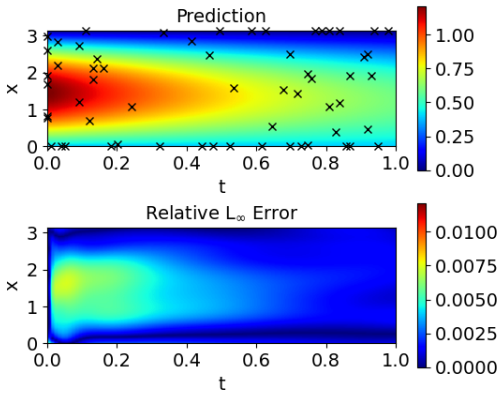
For all cases we have only a small maximal error of less than 1%. The largest (maximal and L_2) error can be observed for case $l = 0.3$, $r = 0.0$ shown in Figure 3.8b. Here, the maximal error is of 0.74% and the relative L_2 error is 5.51×10^{-3}. A similar error can be observed for cases $l = 0.2$, $r = 0.0$ and $l = 0.4$, $r = 0.0$ which are part of the training data. This can be seen in Figures 3.8a, c and Table 3.6. It occurs that for fixed $l \in \{0.2, 0.3, 0.4, 0.5\}$ the overall error is largest for $r = 0.0$ and then decreases for growing r as indicated by Figures 3.8c-f. This behavior changes for $l \in \{0.6, 0.7, 0.8, 0.9\}$. Here, the overall error increases up to a certain point and then starts decreasing again which is shown in Figures 3.8h-j. The smallest relative L_2 error of 1.47×10^{-3} is achieved by case $l = 1.0$, $r = 0.9$ (see Fig. 3.8l) which is not part of the training data.



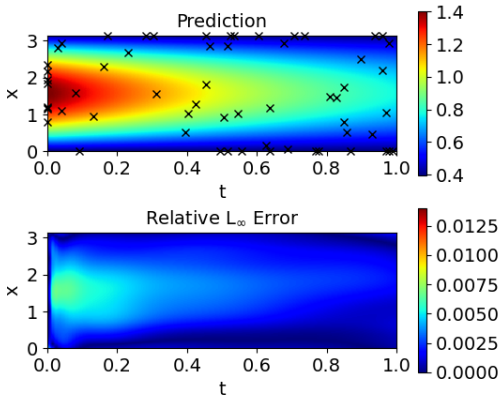
(a) l=0.2, r=0.0



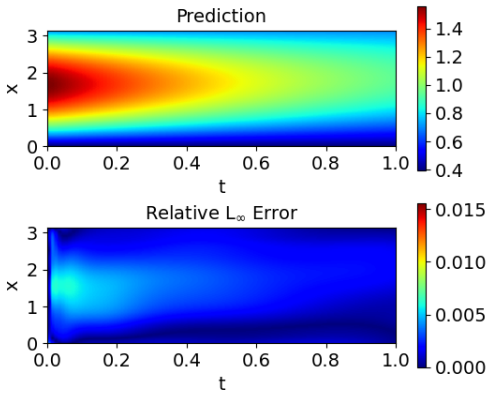
(b) l=0.3, r=0.0



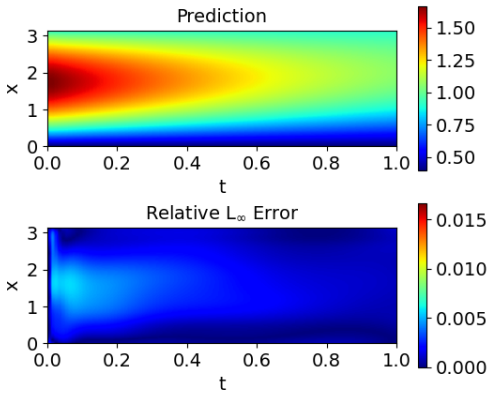
(c) l=0.4, r=0.0



(d) l=0.4, r=0.4

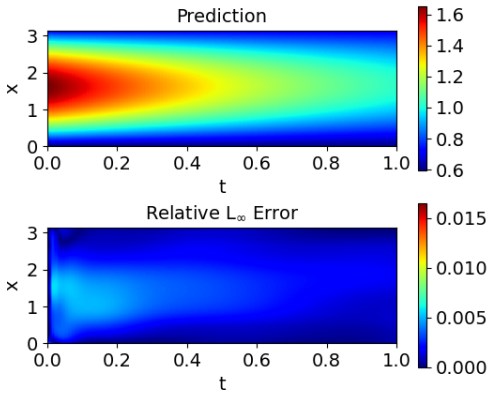


(e) l=0.4, r=0.7

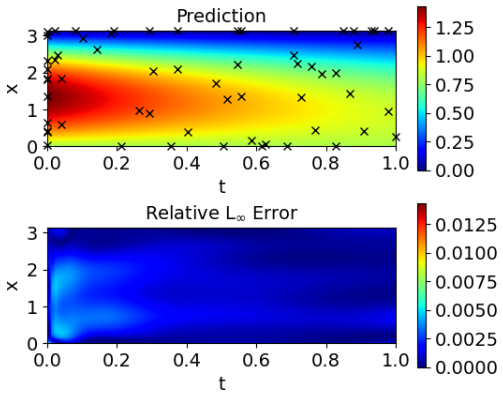


(f) l=0.4, r=0.9

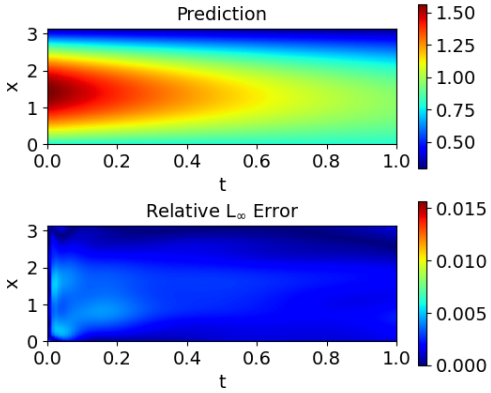
Figure 3.8: Predicted solutions of a PINN trained with sine as IC and 60 training points for various left (l) and right (r) boundary values. The black markers represent the used training points. Cases where no markers are visible are not part of the training data. *Top*: Prediction. *Bottom*: Relative L_∞ error. (a)-(c): Minor errors at around $x = \frac{\pi}{2}$ near initial condition. Error vanishes in positive t-direction. (c)-(f): Error vanishes for fixed l and growing r.



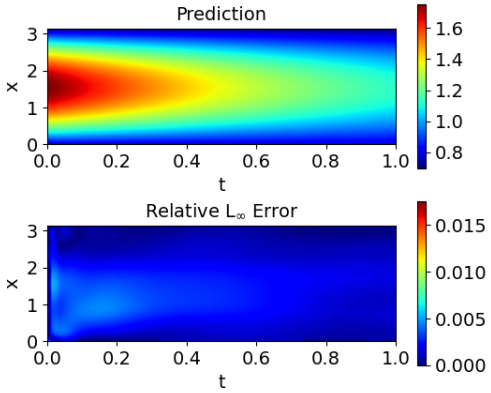
(g) l=0.6, r=0.7



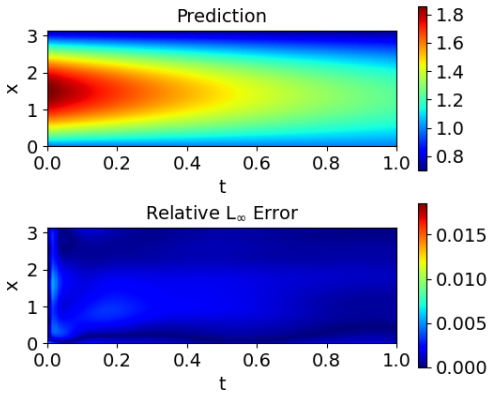
(h) l=0.8, r=0.0



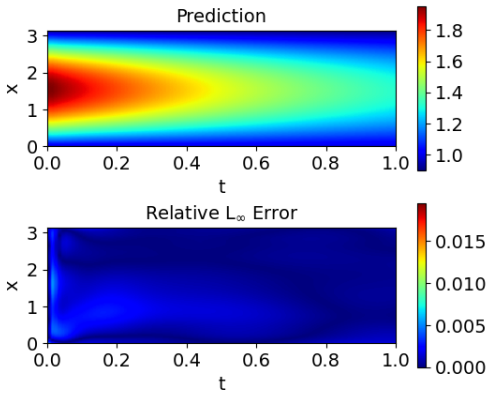
(i) l=0.8, r=0.3



(j) l=0.8, r=0.7



(k) l=1.0, r=0.7



(l) l=1.0, r=0.9

Figure 3.8: Predicted solutions of a PINN trained with sine as IC and 60 training points for various left (l) and right (r) boundary values. The black markers represent the used training points. Cases where no markers are visible are not part of the training data. *Top*: Prediction. *Bottom*: Relative L_∞ error. For all cases one can only see minor errors smaller than 0.6%. (g): Minor errors (up to 0.57%) near initial condition. (h)-(j): The overall error decreases for fixed l and growing r. (l): Case of smallest relative L_2 error (1.47×10^{-3}).

3.2.2 Initial Condition: Line with Discontinuous Boundaries

In Figure 3.9 the train and tess losses of the models trained with sizes in $\{40, 60, 80\}$ are pictured. As initial condition we choose the line $u_0(x) = -x + 1$ where the boundary has varying values in M. For cases 60 and 80 the loss curves show an overfitting of the models which is indicated by the large gap between the train and test losses, respectively. For size 60 this gap is larger. Here, the best model is already achieved at epoch 8700. The corresponding loss values and the runtimes are depicted in Table 3.7. For this setting we do not consider a training beyond 20000 epochs since this amount is already sufficient for the test loss curves to converge as can be seen in Figure 3.9. For a size of 80 the runtime of the training process is 7.2 hours.

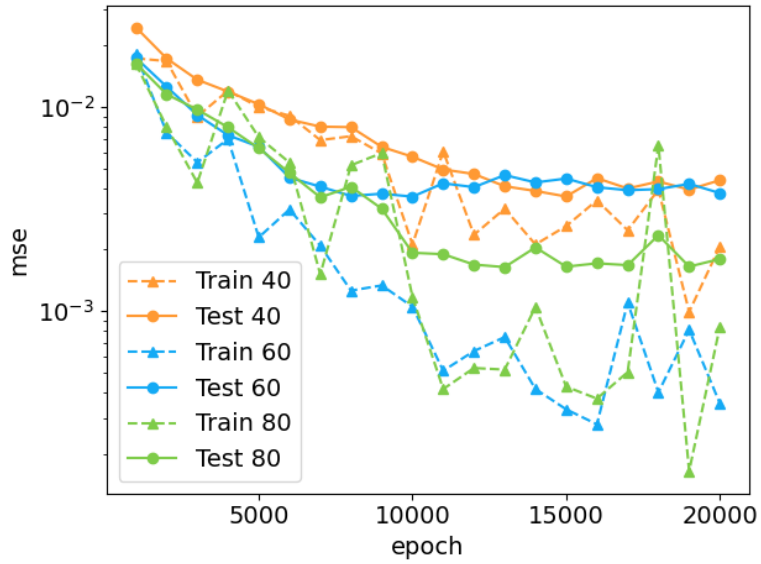
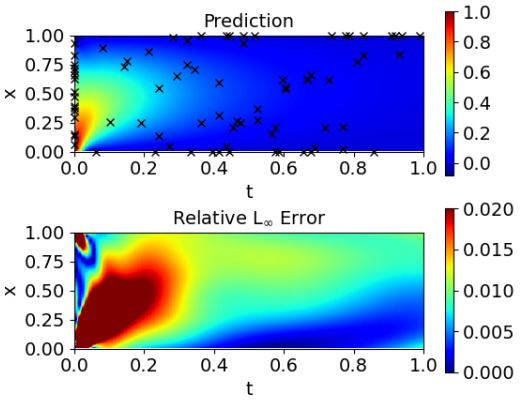


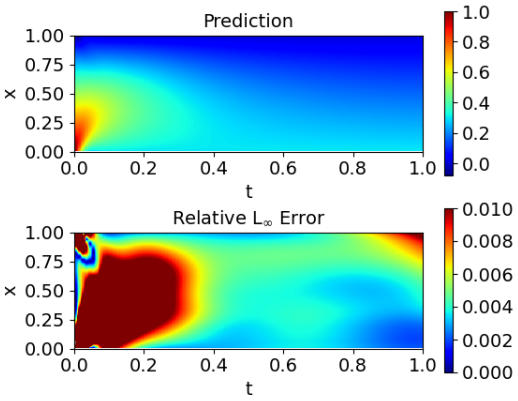
Figure 3.9: Train and test losses of models trained with the line $u_0(x) = -x+1$ as IC and boundary values in $\{0, 0.2, 0.4, 0.6, 0.8, 1\}$. The models were trained for sizes in $\{40, 60, 80\}$ and 20000 epochs with Adam as optimizer and a learning rate of 0.0001.

We consider results obtained by the model trained with size 80. As in the section before, we predict the solutions of the one-dimensional heat equation with BCs in $[0, 1]$ and stepsize 0.1 resulting in 121 cases. In Figure 3.10, some cases of predicted solutions with the respective relative L_∞ error are pictured. The corresponding maximal errors with their locations as well as the relative L_2 errors can be found in Table 3.8.

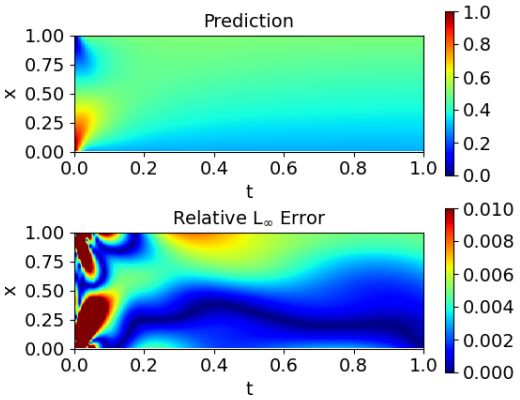
For all cases one can see a large error in those areas where the discontinuities occur.



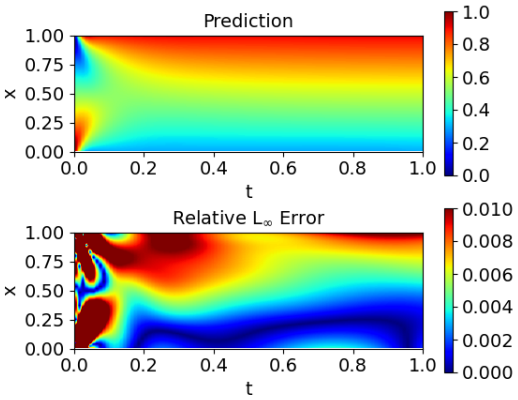
(a) l=0.0, r=0.0



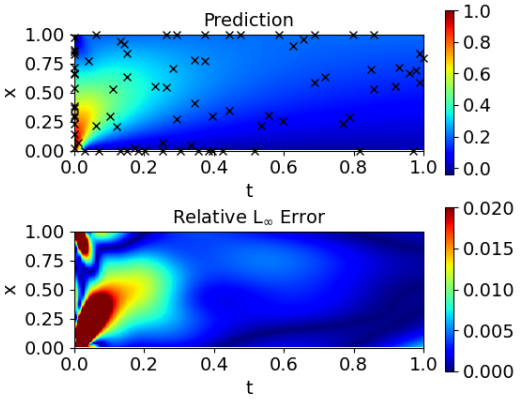
(b) l=0.3, r=0.0



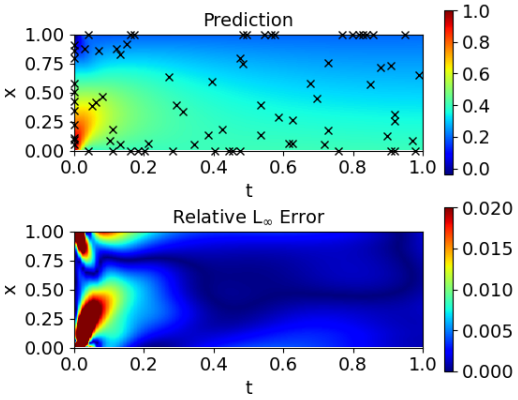
(c) l=0.3, r=0.5



(d) l=0.3, r=0.9

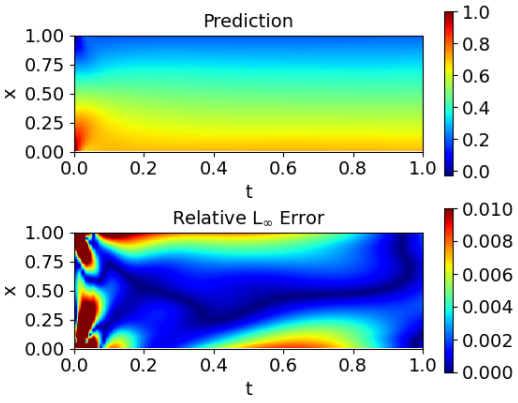


(e) l=0.0, r=0.2

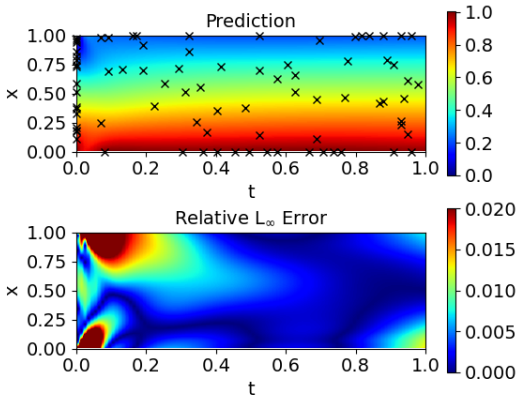


(f) l=0.4, r=0.2

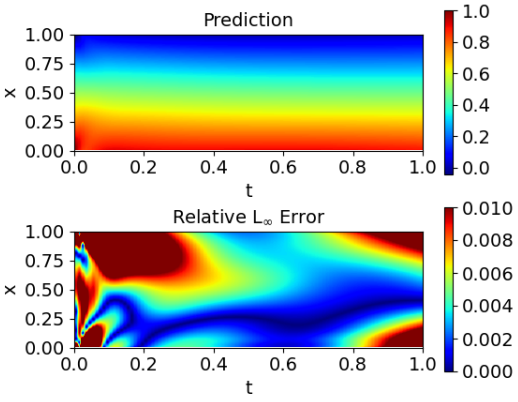
Figure 3.10: Predicted solutions of a PINN trained with the line $u_0(x) = -x + 1$ as IC and 80 training points for various left (l) and right (r) boundary values. The black markers represent the used training points. Cases where no markers are visible are not part of the training data. *Top*: Exact solution. *Middle*: Prediction. *Bottom*: Relative L_∞ error. (a): Largest maximal error (63.13%) and largest overall error (1.28×10^{-1}) of all cases. (b)-(d): Large errors (up to 40.98%) near the boundaries where the discontinuities occur. For growing r the error areas neart the initial condition decrease.



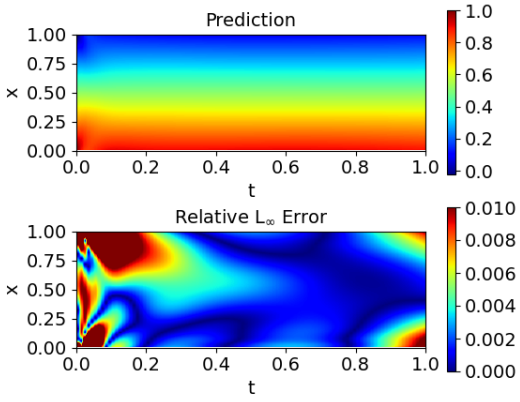
(g) l=0.7, r=0.2



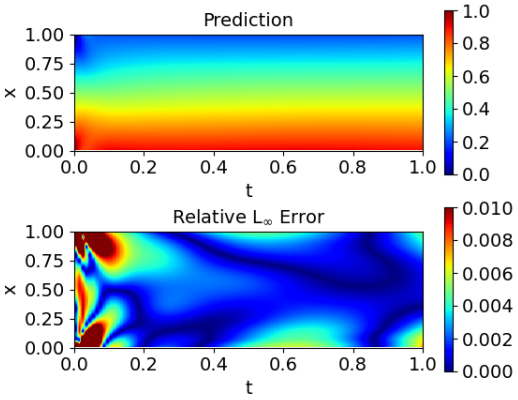
(h) l=1.0, r=0.2



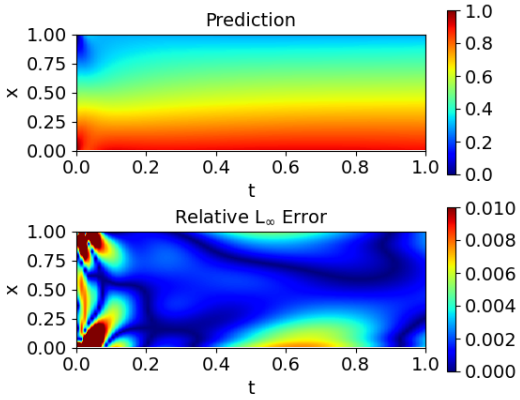
(i) l=0.9, r=0.0



(j) l=0.9, r=0.1

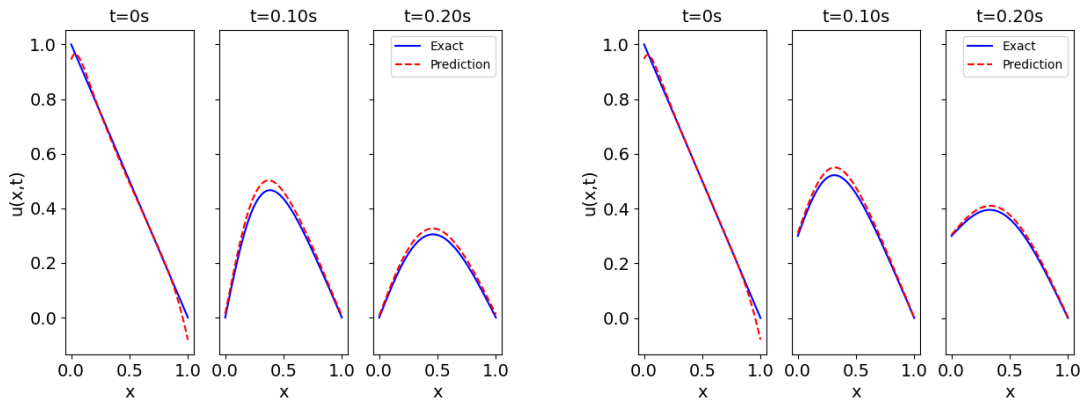


(k) l=0.9, r=0.2



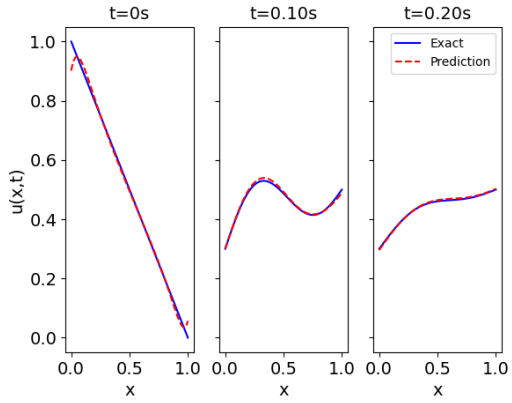
(l) l=0.9, r=0.3

Figure 3.10: Predicted solutions of a PINN trained with the line $u_0(x) = -x + 1$ as IC and 80 training points for various left (l) and right (r) boundary values. The black markers represent the used training points. Cases where no markers are visible are not part of the training data. *Top*: Exact solution. *Middle*: Prediction. *Bottom*: Relative L_∞ error. (i)-(l): Errors (up to 9.62%) are made near the boundaries where the discontinuities occur. For growing r the error areas decrease or vanish.

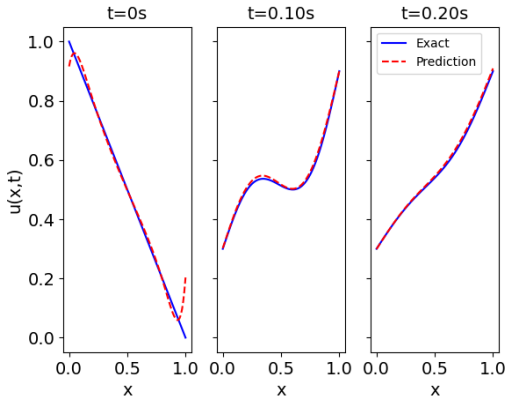


(a) l=0.0, r=0.0

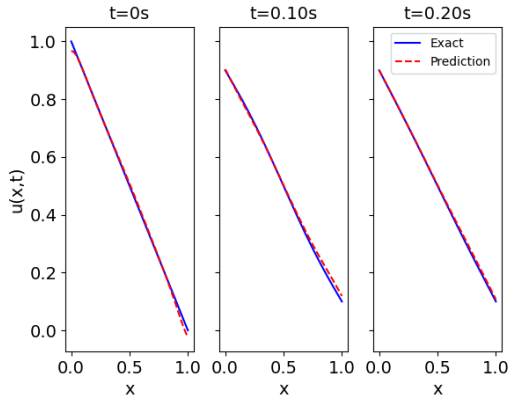
(b) l=0.3, r=0.0



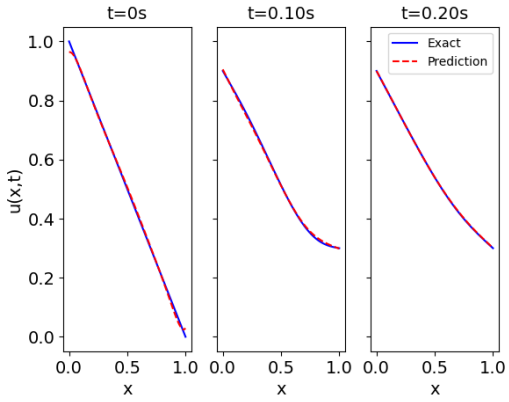
(c) l=0.3, r=0.5



(d) l=0.3, r=0.9



(e) l=0.9, r=0.1



(f) l=0.9, r=0.3

Figure 3.11: Line plots of exact and predicted solutions of the one-dimensional heat equation with the line $u_0(x) = -x + 1$ as IC and discontinuous boundaries depicted at time steps $t = 0, 0.1, 0.2$. (a)-(b): For $t = 0$ the prediction becomes negative at boundary $x = 1$. (c)-(f): The larger the difference between the line and BC, the larger the error at the corresponding boundary.

Table 3.7: Runtime of the models trained with the line $u_0(x) = -x + 1$ as IC and sizes in $\{40, 60, 80\}$ for 20000 epochs. Additionally, the train and test loss values of the best models are depicted.

size	runtime	train loss	test loss
40	5.9h	2.78e-03	3.67e-03
60	6.7h	3e-03	3.48e-03
80	7.2h	5e-04	1.46e-03

Table 3.8: Predictive errors of the model trained for solving the heat equation $u_t = \lambda u_{xx}$ with the line $u_0(x) = -x + 1$ as IC, boundary values in $\{0, 0.2, 0.4, 0.6, 0.8, 1\}$ and a training size of 80. The two left columns represent the left (l) and right (r) boundary values, respectively. Additionally, the maximal relative L_∞ error with locations and the overall relative L_2 error are depicted.

l	r	$(x,t)^T$	max. error	rel. L_2 error
0.0	0.0	$(0, 0.01)^T$	63.13%	1.28e-01
0.3	0.0	$(0, 0.01)^T$	40.98%	5.1e-02
0.3	0.5	$(0, 0.01)^T$	30.28%	2.3e-02
0.3	0.9	$(0, 0.01)^T$	31.11%	2.04e-02
0.9	0.0	$(1, 0.03)^T$	8.3%	1.66e-02
0.9	0.1	$(0, 0.03)^T$	6.24%	1.06e-02
0.9	0.2	$(1, 0.01)^T$	6.25%	8.1e-03
0.9	0.3	$(1, 0.01)^T$	9.62%	7.58e-03
0.0	0.2	$(0, 0.01)^T$	53.94%	7.39e-02
0.4	0.2	$(0, 0.01)^T$	29.41%	2.74e-02
0.7	0.2	$(0, 0.01)^T$	11.18%	1.15e-02
1.0	0.2	$(0, 0.02)^T$	9.49%	1.29e-02

The largest maximal error of 63.13% is obtained by case $l = 0.0$, $r = 0.0$ depicted in Figure 3.10a. Depending on the boundary condition, the error propagates further in positive t-direction. For a larger difference between the boundary values and the line, or in cases where the right boundary, i.e. $x = 1$, tends to zero, this propagation is more distinct. Note that for most cases, the maximal error is in other areas about 1% or smaller. As already mentioned, the largest maximal error is made in case of $l = 0.0$, $r = 0.0$. In addition to that, this case makes an overall error of 0.128 which is also the largest compared to the other cases. The prediction with the lowest relative L_2 error (7.58×10^{-3}) is obtained by case $l = 0.9$, $r = 0.3$ (Fig. 3.10l) as depicted in Table 3.8.

In Figure 3.11 we additionally display for some cases how the model tries to fit the BC at $t = 0$. Especially when comparing 3.11c and 3.11d, one can see that the larger the difference between the line and boundary value, the larger the L_∞ error at the respective boundary. This can be verified by looking at the corresponding values for fixed $l = 0.9$ or $r = 0.2$ in Table 3.8. An exception provide the cases where $r = 0.0$, i.e. $u(1, 0) = 0$. Although the difference tends to zero here, the prediction becomes negative in these cases leading to a larger error.

3.2.3 Additional Nonlinear Term: Radiative Loss

We investigate the heat equation with an additional nonliner term, i.e. $u_t = \lambda u_{xx} - \sigma u^4$. Here, we consider models trained with sine as initial condition and boundary values in M. The sizes of the training datasets are in $\{30, 40, 60\}$. In Figure 3.12 one can see the train and tess losses of the considered models. For all cases the loss curves proceed in the same order of magnitude. However, for epochs larger than 40000 the curves start departing from each other slightly. The best model is achieved for size 60. In Table 3.9 the runtimes of the models are displayed. The duration of the training process for size 60 is 19.6h which is similar to the case with sine as IC and without the additional nonlinear term.

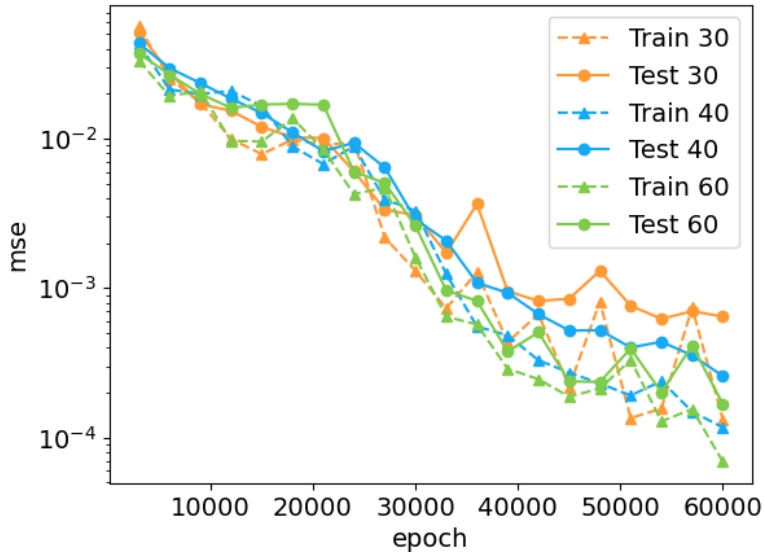


Figure 3.12: Train and test losses of models trained to solve the nonlinear heat equation $u_t = \lambda u_{xx} - \sigma u^4$ with sine as IC and boundary values in $\{0, 0.2, 0.4, 0.6, 0.8, 1\}$. The models were trained for sizes in $\{30, 40, 60\}$ and 60000 epochs with Adam as optimizer and a learning rate of 0.0001.

Table 3.9: Runtime of the models trained with an additional nonlinear term with sizes in $\{30, 40, 60\}$ for 60000 epochs. Additionally, the train and test loss values of the best models are depicted.

size	runtime	train loss	test loss
30	17.2h	1.6e-04	5.5e-04
40	18h	6e-05	2.3e-04
60	19.6h	7e-05	1.6e-04

We are looking at the results obtained by the model trained with size 60. As

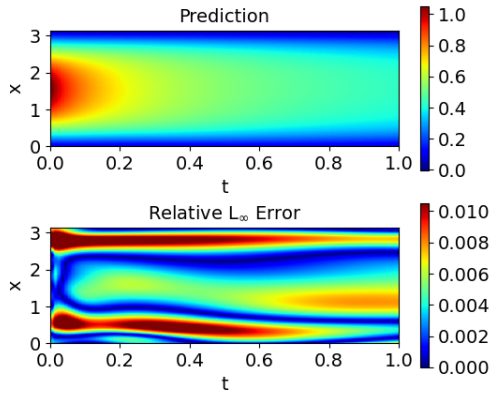
before, we predict the solution for boundary values in $[0, 1]$ with stepsize 0.1. In Figure 3.13 we have displayed some examples of the 121 cases. We further show the corresponding overall and maximal errors with the respective locations in Table 3.10.

Table 3.10: Predictive errors of the model trained for solving the nonlinear heat equation $u_t = \lambda u_{xx} - \sigma u^4$ with sine as IC, boundary values in $\{0, 0.2, 0.4, 0.6, 0.8, 1\}$ and size 60. The two left columns represent the left (l) and right (r) boundary values, respectively. Additionally, the maximal relative L_∞ error with locations and the overall relative L_2 error are depicted.

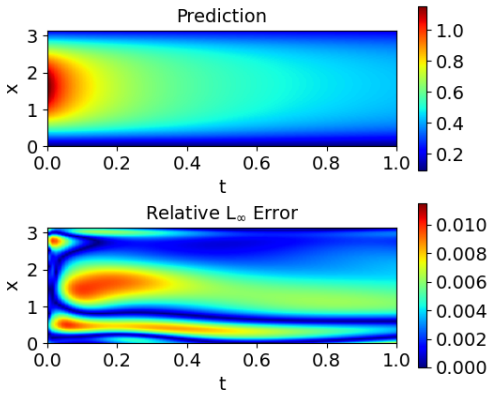
l	r	$(x,t)^T$	max. error	rel. L_2 error
0.1	0.0	$(2.79, 0.01)^T$	1.78%	1.24e-02
0.1	0.2	$(0.48, 0.05)^T$	0.99%	1.02e-02
0.1	0.6	$(1.68, 0.06)^T$	1.3%	9.66e-03
0.1	0.9	$(1.94, 0.03)^T$	1.96%	1.24e-02
0.3	0.1	$(2.79, 0.01)^T$	1.18%	8.29e-03
0.5	0.0	$(1.36, 0.06)^T$	1.24%	9.63e-03
0.8	0.0	$(1.33, 0.04)^T$	1.73%	1.09e-02
0.8	0.2	$(1.43, 0.04)^T$	2.14%	1.31e-02
0.8	0.6	$(1.55, 0.03)^T$	2.77%	1.66e-02
0.8	0.9	$(1.52, 0.02)^T$	3.28%	1.95e-02
1.0	0.0	$(0, 0)^T$	5.12%	1.26e-02
1.0	1.0	$(1.49, 0.02)^T$	4%	2.25e-02

It appears that for fixed $l \in \{0, 0.1, 0.2, 0.3, 0.4\}$ and $r = 0.0$, the prediction shows a larger error close to the boundaries compared to other regions of the domain. For growing r this error decreases. Instead, the error concentrates on the area around $(x,t)^T = (\frac{\pi}{2}, 0.05)^T$ as can be seen in Figures 3.13a-d. Figure 3.13f shows that this scheme is only barely present for $l = 0.5$. Here, one can already see the concentration of the error on the aforementioned area for $r = 0.0$ having a maximal error of 1.24%. For a fixed $l > 0.5$ and $r = 0.0$, the relative L_∞ error is largest in the same area as mentioned before. For growing r this area increases which is shown by Figures 3.13g-j. The largest relative L_2 error of 2.25×10^{-2} is obtained by case $l = 1.0$, $r = 1.0$. The smallest overall error of 8.28×10^{-3} is achieved by the case $l = 0.3$, $r = 0.1$ shown by Figure 3.13e.

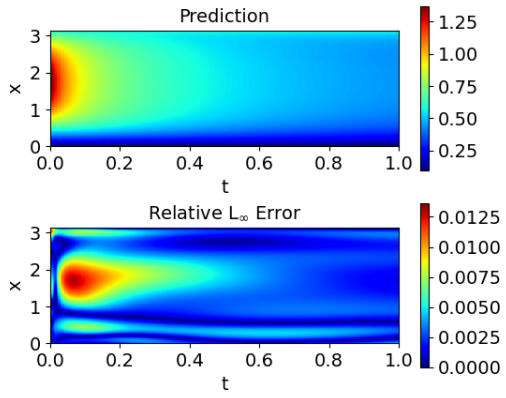
In Figure 3.14 we have pictured for various cases line plots of the exact and predicted solutions at time levels $t \in \{0, 0.1, 0.2\}$. The functions shown in Figures 3.14a-b assume the shape of a parabola. In these cases the errors are mainly made



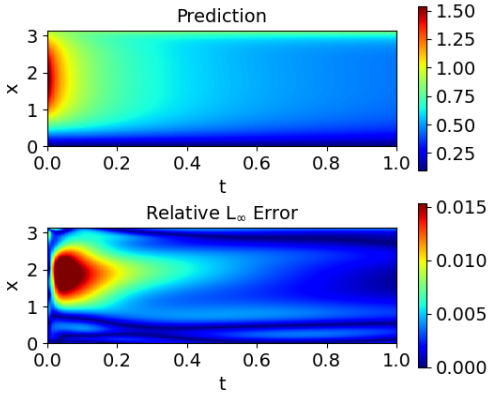
(a) l=0.1, r=0.0



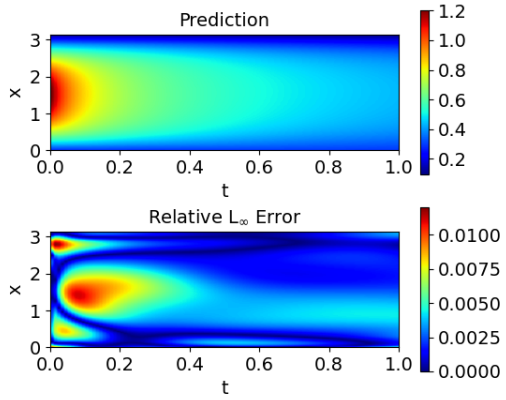
(b) l=0.1, r=0.2



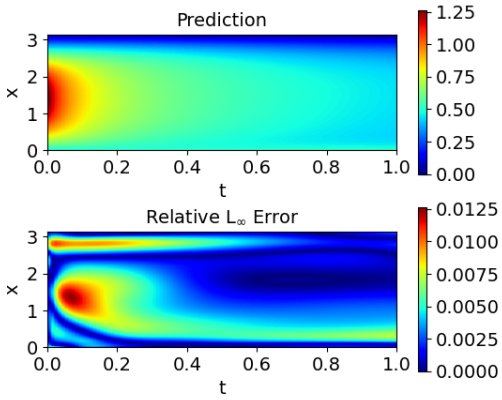
(c) l=0.1, r=0.6



(d) l=0.1, r=0.9

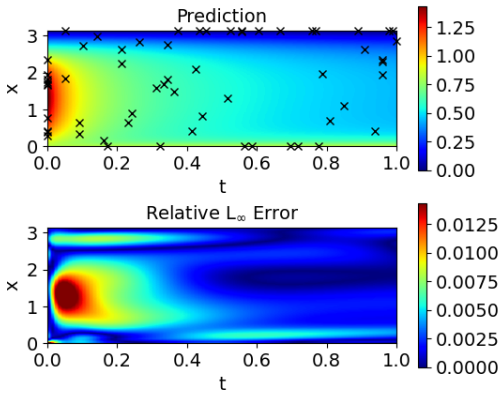


(e) l=0.3, r=0.1

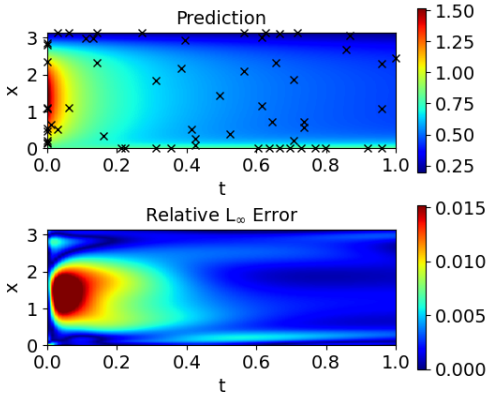


(f) l=0.5, r=0.0

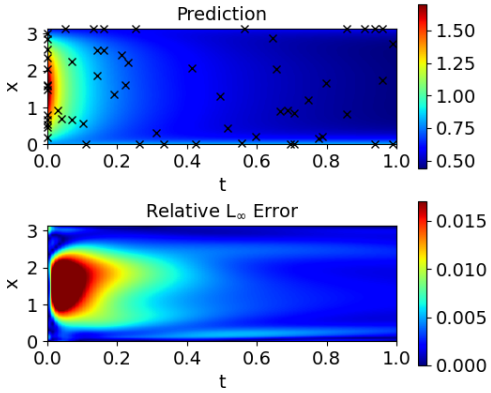
Figure 3.13: Predicted solutions of a PINN trained for solving the nonlinear heat equation $u_t = \lambda u_{xx} - \sigma u^4$ with sine as IC and 60 training points for various left (l) and right (r) boundary values. All depicted cases represent unseen data. *Top*: Prediction. *Bottom*: Relative L_∞ error. (a): Large errors (up to 1.78%) at the boundaries. (b)-(d): For growing r the error translocates to the increasing area around $(x,t)^T = (\frac{\pi}{2}, 0.05)^T$. (e): Case with smallest relative L_2 error of 8.29e-03. (f): Errors mainly near boundary $x = \pi$ and the area around $(x,t)^T = (\frac{\pi}{2}, 0.05)^T$.



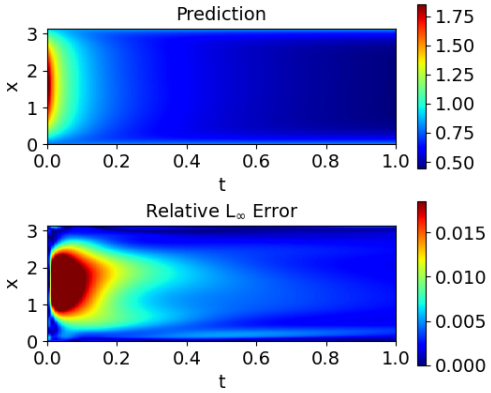
(g) l=0.8, r=0.0



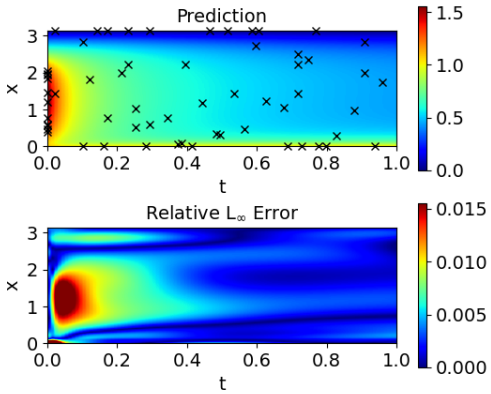
(h) l=0.8, r=0.2



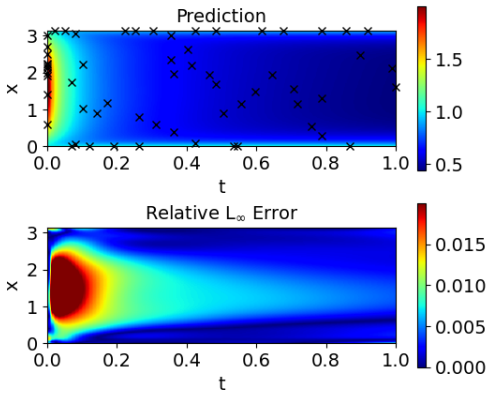
(i) l=0.8, r=0.6



(j) l=0.8, r=0.9

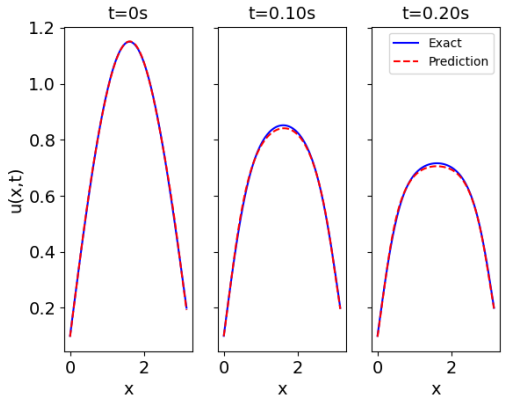


(k) l=1.0, r=0.0

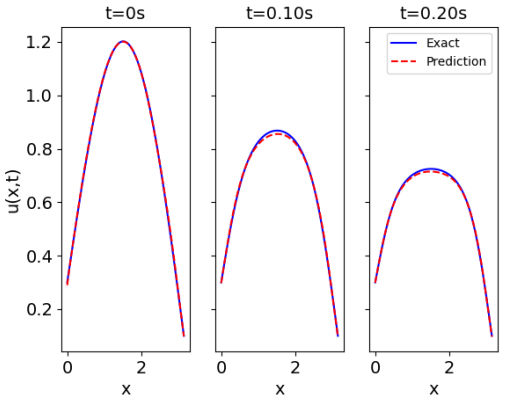


(l) l=1.0, r=1.0

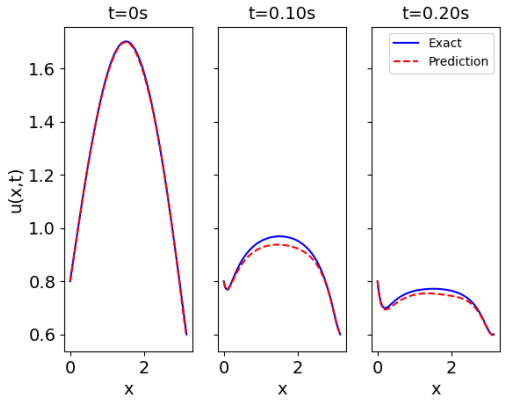
Figure 3.13: Predicted solutions of a PINN trained for solving the nonlinear heat equation $u_t = \lambda u_{xx} - \sigma u^4$ with sine as IC and 60 training points for various left (l) and right (r) boundary values. The black markers represent the used training points. Cases where no markers are visible are not part of the training data. *Top*: Prediction. *Bottom*: Relative L_∞ error. (g)-(j): Large error in the area around $(x,t)^T = (\frac{\pi}{2}, 0.05)^T$. Increasing area of error for growing r. (k): Case with largest relative L_∞ error (5.12%) made at $(0,0)^T$. (l): Case with largest relative L_2 error of 2.25×10^{-2}.



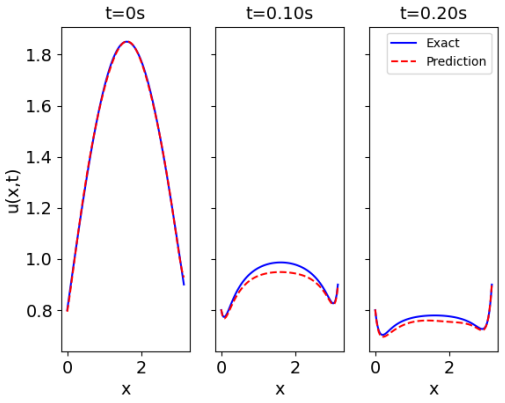
(a) l=0.1, r=0.2



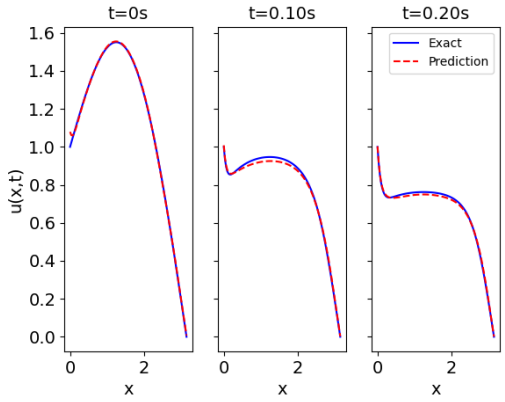
(b) l=0.3, r=0.1



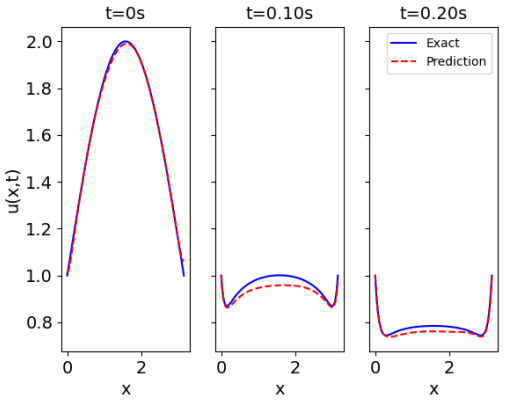
(c) l=0.8, r=0.6



(d) l=0.8, r=0.9



(e) l=1.0, r=0.0



(f) l=1.0, r=1.0

Figure 3.14: Line plots of exact and predicted solutions of the nonlinear heat equation $u_t = \lambda u_{xx} - \sigma u^4$ with sine as IC, boundary values in $\{0, 0.2, 0.4, 0.6, 0.8, 1\}$ and size 60 depicted at time steps $t = 0, 0.1, 0.2$. One can see that the more peaked the indentations at the boundaries, the larger the error at the vertex of the parabolic shaped functions. (a)-(b): Minor errors of less than 1% around $x = \frac{\pi}{2}$ for $t \in \{0.1, 0.2\}$. (c)-(d): Larger error of about 3% around $x = \frac{\pi}{2}$ for $t \in \{0.1, 0.2\}$. (e): Large deviation between exact and predicted solution at $(0, 0)^T$. (f): Large error around $x = \frac{\pi}{2}$.

at the vertex of the parabola. Note that these are only minor errors as mentioned above. In cases shown in Figures 3.14c-f, for $t > 0$ the shape of the parabola is still recognizable. However, indentations are visible at the boundaries. For 3.14d,f these indentations are at both boundary points and in 3.14c,e there are especially at the left boundary, i.e. $x = 0$. Note that in case of $l = 1.0$, $r = 0.0$ the prediction already tries to simulate this indent at $(0, 0)^T$ resulting in the largest maximal error of all cases (see Fig. 3.14e). Regarding these indents one can see that the more peaked the indentations, the greater the error between the exact and predicted solutions at the vertex of the parabola.

3.3 Various Boundary and Initial Conditions

We present the results obtained by training a PINN such that it is able to solve the one-dimensional heat equation $u_t = \lambda u_{xx}$ for arbitrary initial and boundary conditions. Here, we trained the models with several initial conditions represented by six of their corresponding Fourier coefficients and sizes in $\{40, 60, 80\}$. In Figure 3.15 we have plotted the losses of the considered models. One can see that all

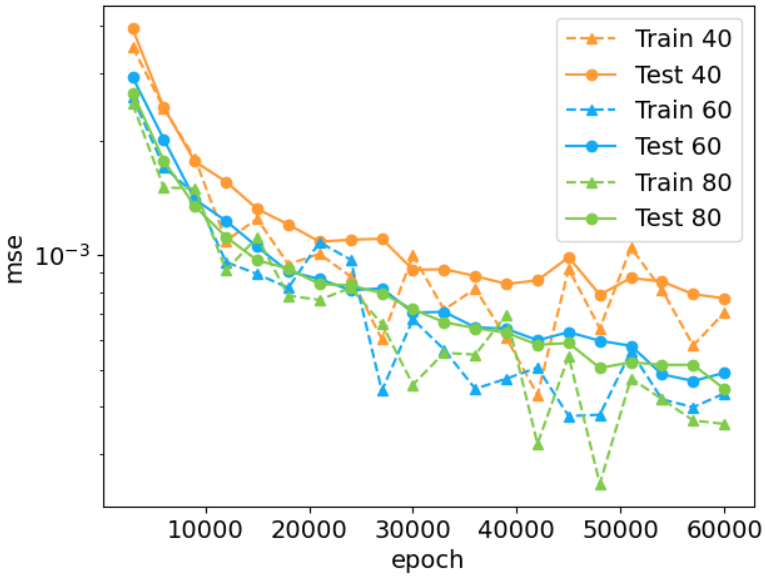


Figure 3.15: Train and test losses of models trained with several initial functions parameterized by six Fourier coefficients and boundary values given implicit by the corresponding initial functions. The models were trained for sizes in $\{40, 60, 80\}$ and 60000 epochs with Adam as optimizer and a learning rate of 0.0001.

curves still have potential to further decrease. The train and test curves of each considered size proceed close to each other. It appears, that the test curves of the models trained with size 60 and 80 almost lie on top of each other. The best model is achieved by size 80. In Table 3.11 we have displayed the runtimes of the considered

Table 3.11: Runtime of the models trained with several initial functions parameterized by Fourier coefficients and sizes in $\{40, 60, 80\}$ for 60000 epochs. Additionally, the train and test loss values of the best models are depicted.

size	runtime	train loss	test loss
40	10d 23h	8.4e-04	7.25e-04
60	16d 22h	4.45e-04	4.51e-04
80	19d 11h	3.6e-04	4.44e-04

models. This setting required a tremendous amount of time for training. The model trained with 80 points per function required about 19.5 days for 60000 epochs.

In the following we present results achieved by the model trained with size 80. In this setting we consider the training and test data, separately. Note, that the numbering of functions are independent from each other, i.e. No.1 from the training data is a different function than No. 1 from the test data.

Training Data

In Figures 3.17 and 3.18 we have displayed some of the 2065 initial functions we have used to train the model. In the former we collect some functions which the model managed to learn, in the sense that the prediction fits the shape of the exact solution, and in the latter we exemplarily show cases where the model fails to predict the correct solution. In Table 3.12 we depict the errors for some of the functions.

Table 3.12: Predictive errors of generalized model trained for solving the one-dimensional heat equation $u_t = \lambda u_{xx}$ with several initial functions parameterized by six Fourier coefficients and size 80. The model was evaluated on training data. The left column represents the function number. Additionally, the maximal relative L_∞ error with locations and the overall relative L_2 error are depicted.

No.	$(x,t)^T$	max. error	rel. L_2 error
69	$(0,0)^T$	6.74%	1.03e-02
280	$(0,0)^T$	12.12%	1.97e-02
314	$(0.49,0)^T$	25.86%	7.99e-02
369	$(0.51,0)^T$	21.37%	2.4e-02
384	$(0.19,0)^T$	5.54%	1.2e-02
432	$(0.83,0.29)^T$	29.8%	1.21e-01
794	$(0,0)^T$	4.7%	5.6e-03
1085	$(0,0)^T$	1.85%	3.75e-03
1166	$(0.66,0.01)^T$	0.72%	4.82e-03
1581	$(0.59,0.01)^T$	1.86%	4.46e-03
1711	$(0,0)^T$	1.7%	4.41e-03
1946	$(1,0)^T$	1.71%	5.12e-03
2050	$(0.7,0)^T$	1.57%	4.51e-03

Regarding the functions shown in Figure 3.17, it appears that for odd functions the prediction at $t = 0$ fits more the exact solution than for even functions. Larger errors are especially made at the boundaries here. This can be seen in Figures

3.17a, b, e, h and also in Figures 3.18a-d where even functions are depicted whose prediction error is large. Function No. 432 (Fig. 3.18c) even exhibits the largest overall error of 0.121. The smallest relative L_2 error of 3.75×10^{-3} is achieved by function No. 1085 (see Fig. 3.17k).

For better visualization of the model's performance on the training data we show a histogram depicted in Figure 3.16. About 90% of all 2065 functions achieved a L_2 error of less than 2%. The remaining functions have at least an overall error of less than 10%.

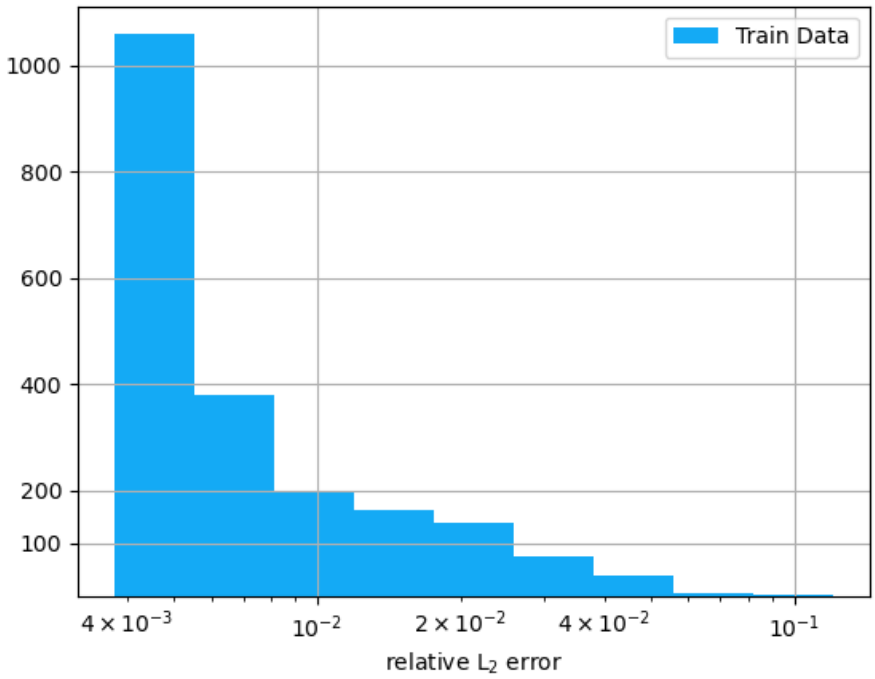
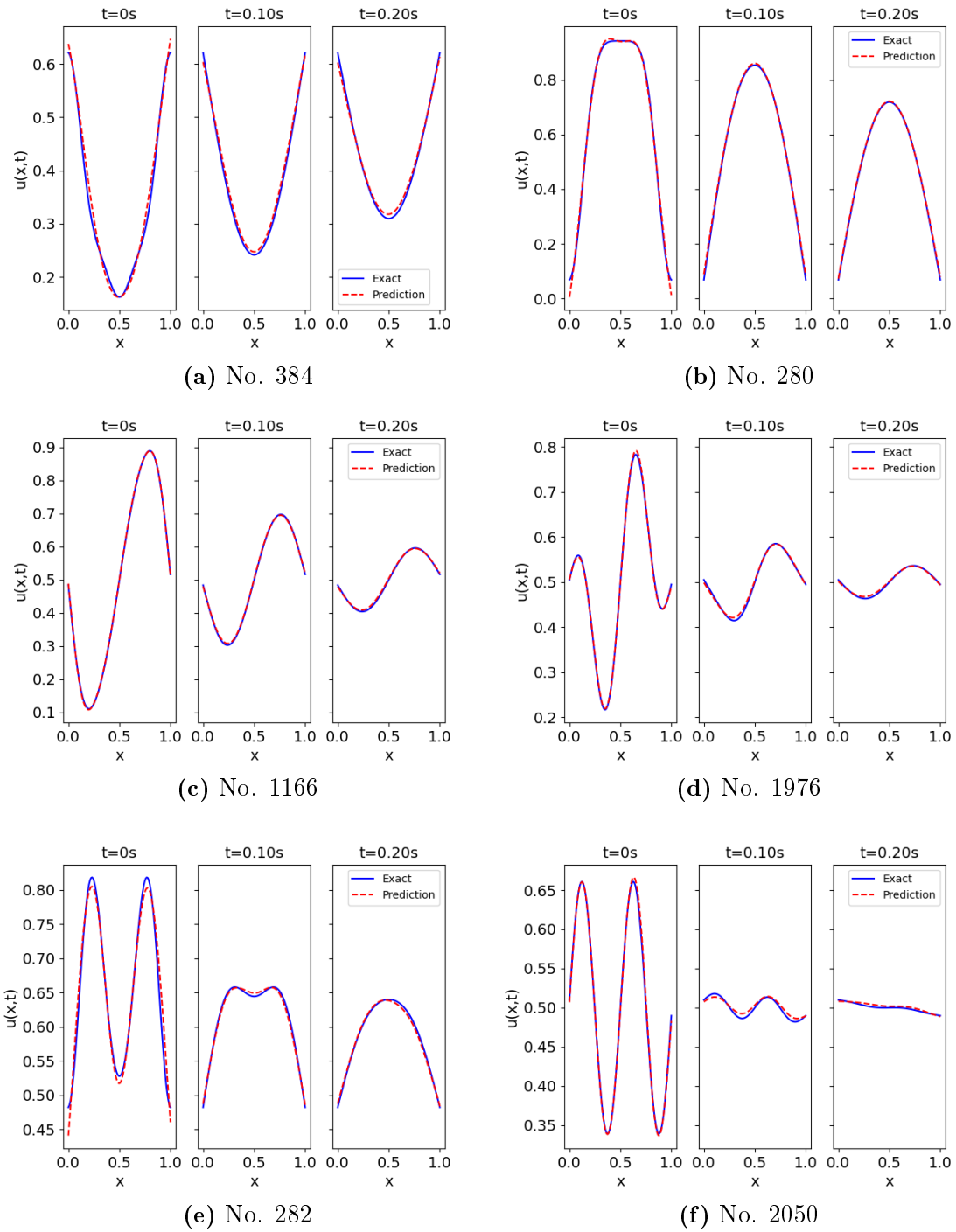


Figure 3.16: Histogram representing the frequency distribution of the (relative L_2) error. Considered are the predictions of the model evaluated at the 2065 functions that were used as initial conditions for training.

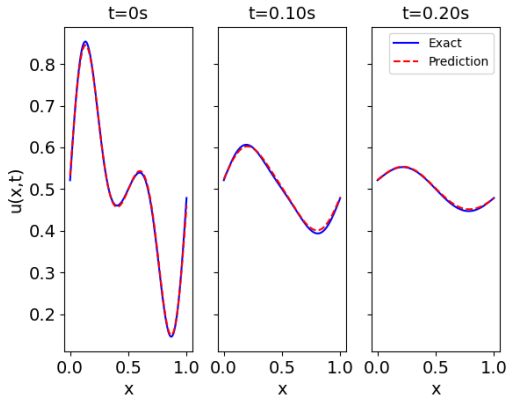


(a) No. 384 **(b)** No. 280

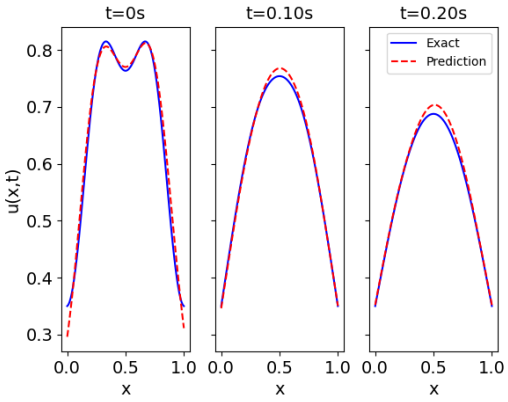
(c) No. 1166 **(d)** No. 1976

(e) No. 282 **(f)** No. 2050

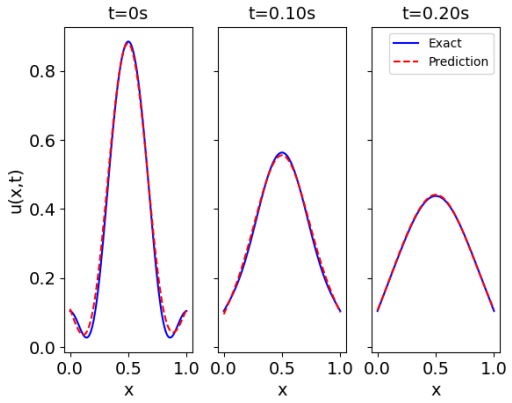
Figure 3.17: Line plots of exact and predicted solutions of the model trained for solving the one-dimensional heat equation with various ICs parameterized by six Fourier coefficients and size 80. Depicted are functions at time levels $t = 0, 0.1, 0.2$ which were used for training. One can see relatively small prediction errors. (a),(b),(e): Even functions provide larger errors, especially at the boundaries.



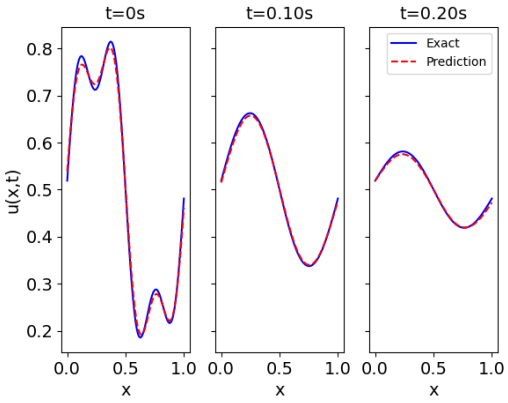
(g) No. 1852



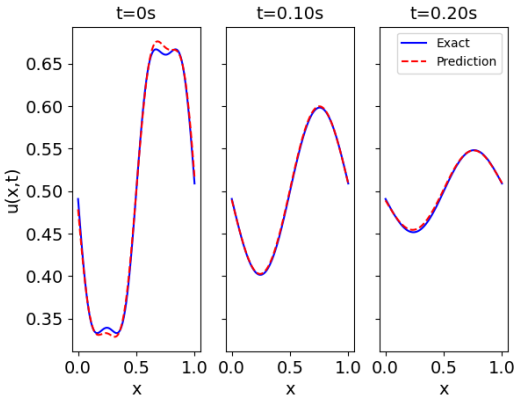
(h) No. 289



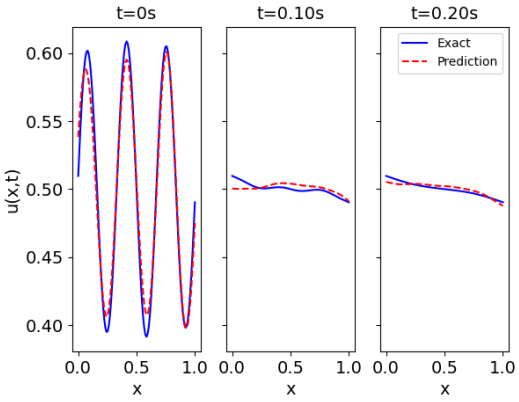
(i) No. 345



(j) No. 743

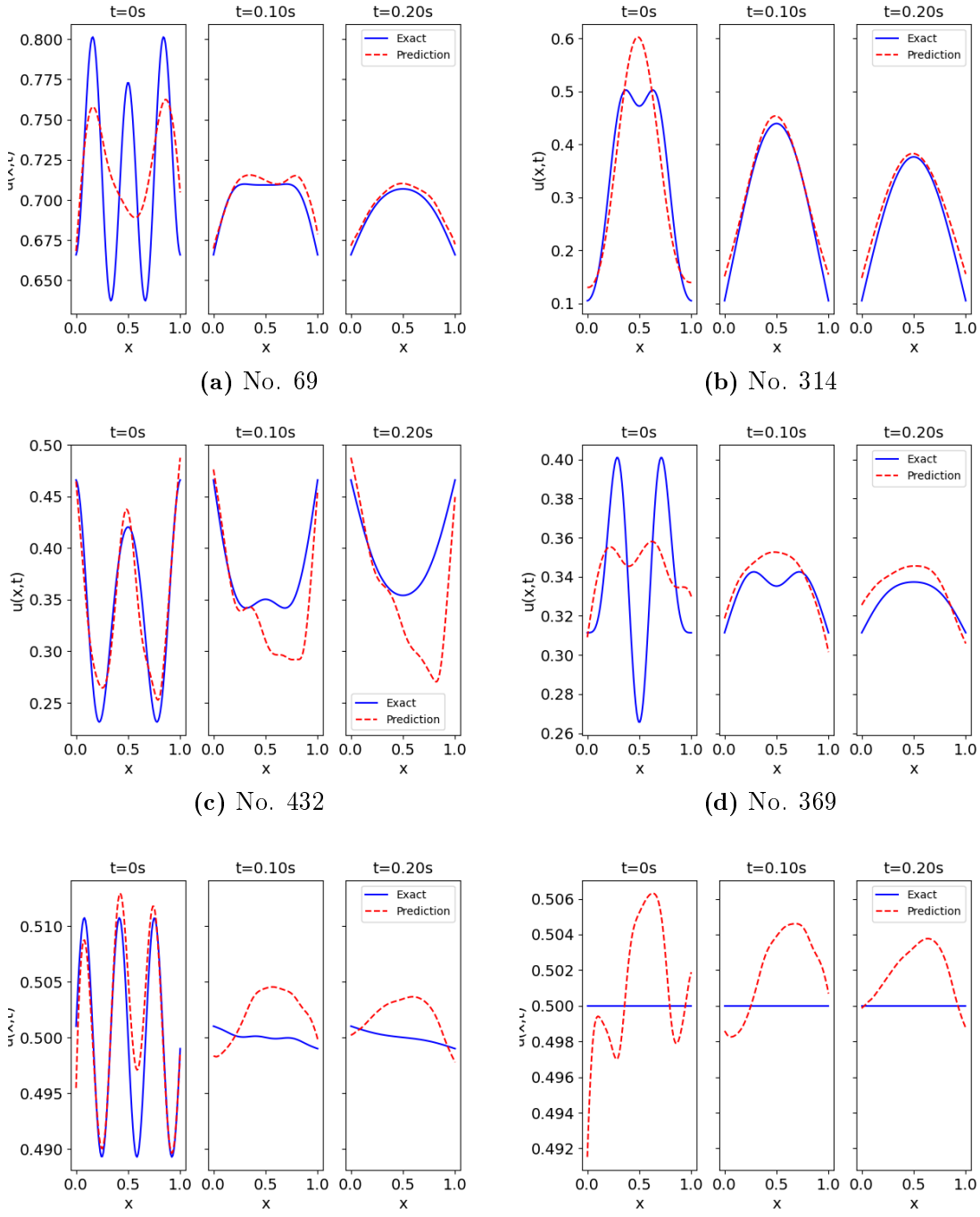


(k) No. 1085



(l) No. 794

Figure 3.17: Line plots of exact and predicted solutions of the model trained for solving the one-dimensional heat equation with various ICs parameterized by six Fourier coefficients and size 80. Depicted are functions at time levels $t = 0, 0.1, 0.2$ which were used for training. (g)-(j): Even functions make larger errors than odd functions. (k): Case with the smallest realtive L_2 error (3.75×10^{-3}).



(a) No. 69 **(b)** No. 314

(c) No. 432 **(d)** No. 369

(e) No. 1581 **(f)** No. 1711

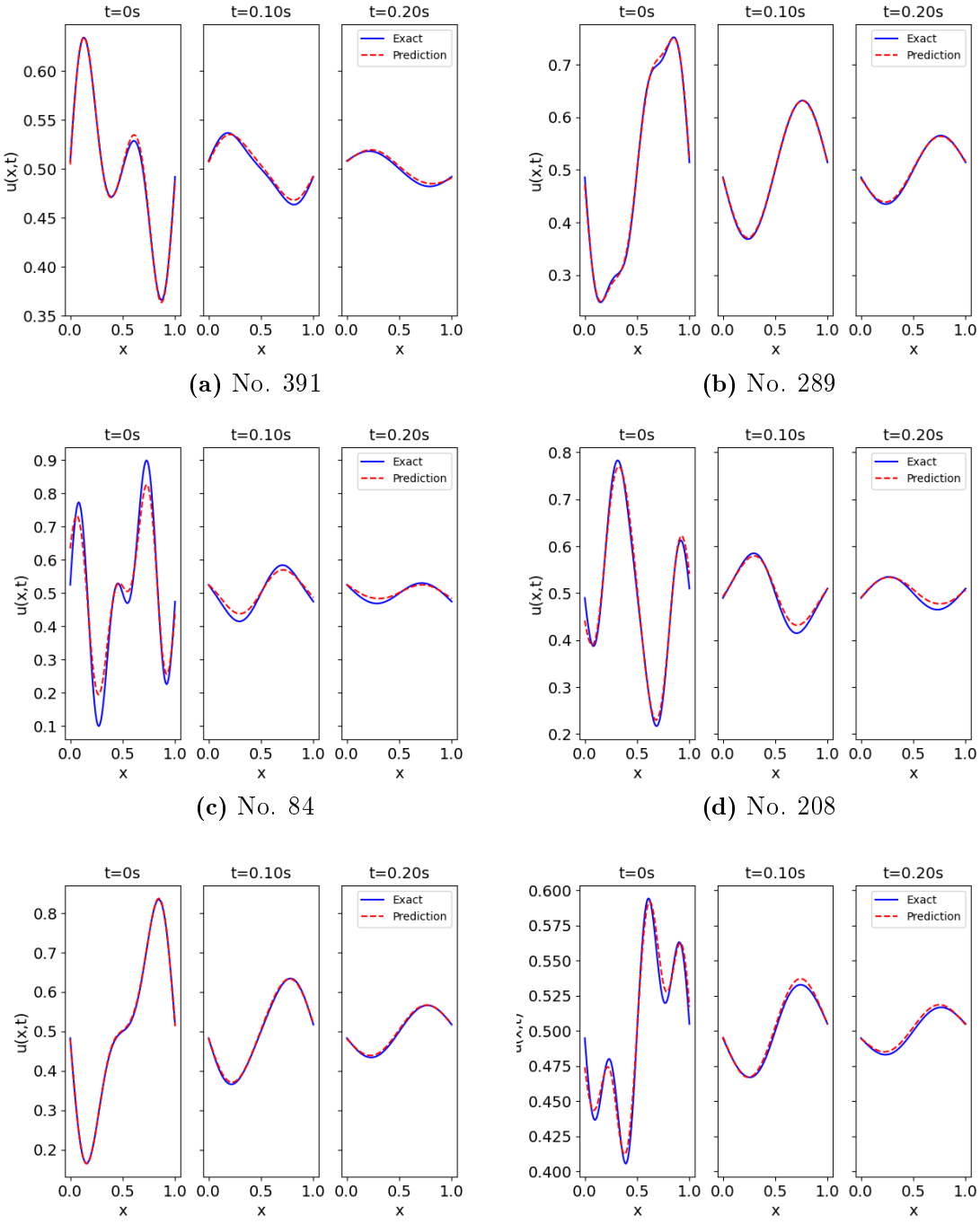
Figure 3.18: Line plots of exact and predicted solutions of the model trained for solving the one-dimensional heat equation with various ICs parameterized by six Fourier coefficients and size 80. Depicted are functions at time levels $t = 0, 0.1, 0.2$ which were used for training. One can see large prediction errors.

Test Data

We now illustrate the performance of the model on unseen data. The test data contains functions that are similar to the functions contained in the training data except that they have a different scaling. As in case of the training data, we show test functions that are well-fitted and functions with a large prediction error, separately. In Figure 3.19 some of the functions which show a low prediction error are depicted. It is noticeable that this collection includes especially odd functions. We already observed in the training data that smaller prediction errors are made for odd functions compared to even ones. The best prediction results are achieved by functions No. 289 and 391 (see Figs 3.19b, a and Tab. 3.13). No. 391 has only an overall error of 4.24×10^{-3} whereas No. 289 has the smallest relative L_2 error of 3.75×10^{-3}. Note that this is the same smallest overall error that was achieved in the training data as well. In Figure 3.20 one can see some of the functions with a poor prediction result. Particularly function No. 140 (see Fig. 3.20d) has a large prediction error with a L_2 error of 264%. Unlike the previous presented functions, No. 122 and 98 (see Figs. 3.20a, b) have their maximal error not near the initial condition. For function No. 122 it is near $t = 0.67$ with a maximal error of 86.11% and in case of No. 98 it is at $t = 1$.

Table 3.13: Predictive errors of generalized model trained for solving the one-dimensional heat equation $u_t = \lambda u_{xx}$ with several initial functions parameterized by six Fourier coefficients and size 80. The model was evaluated on test data. The left column represents the function number. Additionally, the maximal relative L_∞ error with locations and the overall relative L_2 error are depicted.

No.	$(x,t)^T$	max. error	rel. L_2 error
18	$(1,0)^T$	141.21%	5.33e-01
84	$(0,0)^T$	12.12%	2e-02
98	$(0.01,1)^T$	21.78%	3.3e-01
120	$(1,0)^T$	110.74%	6.63e-01
122	$(0.85,0.67)^T$	86.11%	7.62e-01
140	$(0,0)^T$	401.17%	2.64e-00
176	$(0.76,0)^T$	29.76%	5.37e-02
208	$(0,0)^T$	6.11%	1e-02
280	$(0.63,0.02)^T$	1.25%	5.49e-03
289	$(0,0)^T$	1.92%	3.75e-03
314	$(0,0)^T$	3.49%	4.5e-03
391	$(0.65,0)^T$	0.99%	4.24e-03

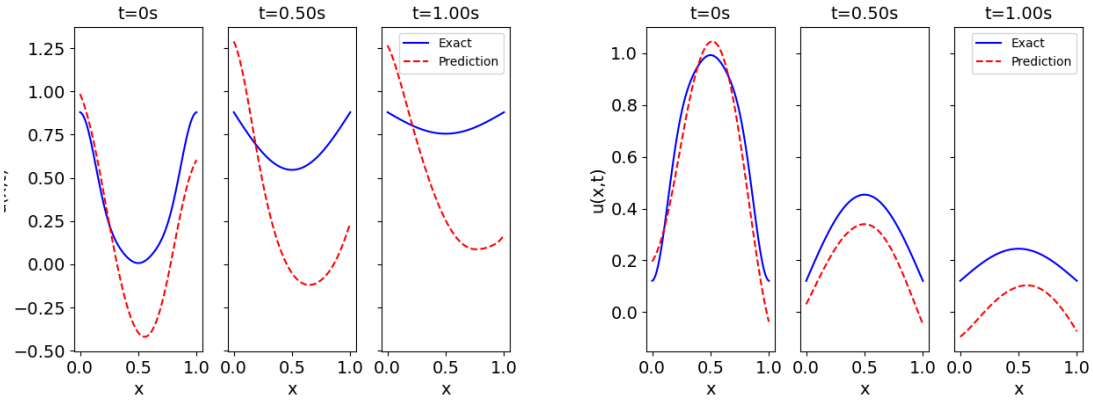


(a) No. 391 **(b)** No. 289

(c) No. 84 **(d)** No. 208

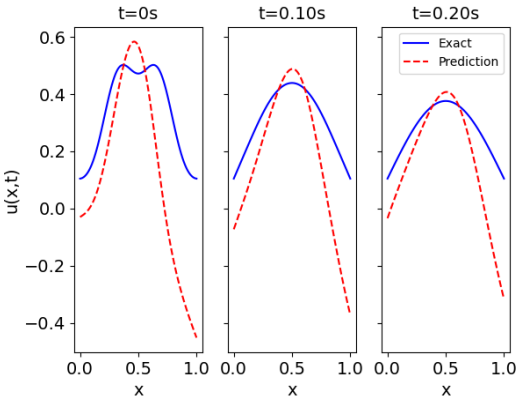
(e) No. 280 **(f)** No. 314

Figure 3.19: Line plots of exact and predicted solutions of the model trained for solving the one-dimensional heat equation with various ICs parameterized by six Fourier coefficients and size 80. The model was evaluated on test data. Depicted are functions at time levels $t = 0, 0.1, 0.2$ whose prediction is close to the exact solution.

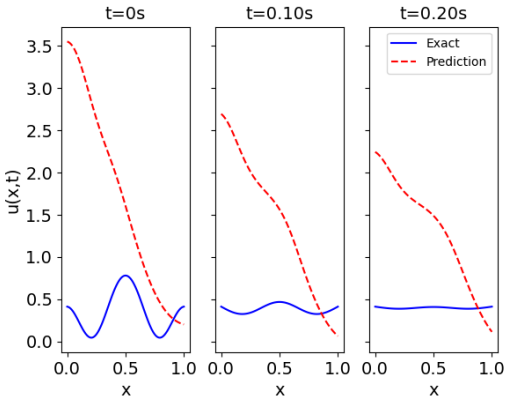


(a) No. 122

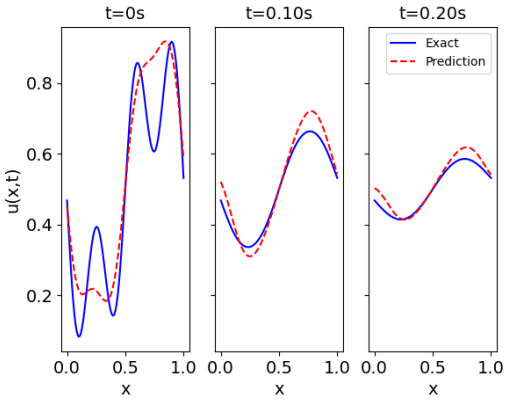
(b) No. 98



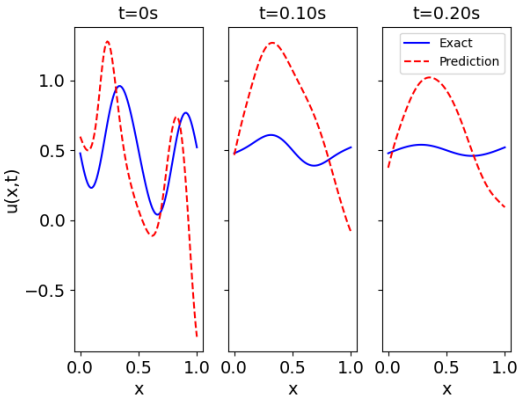
(c) No. 120



(d) No. 140



(e) No. 176



(f) No. 18

Figure 3.20: Line plots of exact and predicted solutions of the model trained for solving the one-dimensional heat equation with various ICs parameterized by six Fourier coefficients and size 80. The model was evaluated on test data. One can see large prediction errors. (a)-(b): Functions depicted at time levels $t = 0, 0.5, 1$ which have a large error at areas different from $t = 0$. (c)-(f): Functions depicted at time levels $t = 0, 0.1, 0.2$.

For the test data, we have also a histogram depicted in Figure 3.21 which shows the amount of functions having a certain relative L_2 error. Here, about 75% of the functions show an error of less than 2%. However, more than 12.5% of all functions show an error of larger than 10%. Of these functions 1.5% have even an error larger than 100%.

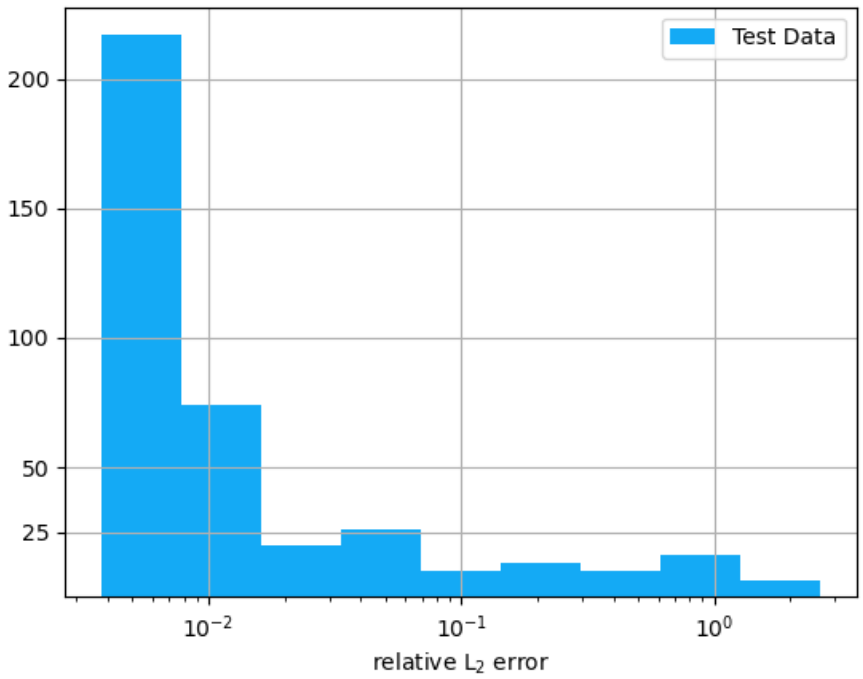


Figure 3.21: Histogram representing the frequency distribution of the (relative L_2) error. Shown are the prediction errors of the model evaluated at 392 test functions.

4 Discussion

The main purpose of this thesis is to investigate on the question if PINNs can be generalized such that they are able to solve several PDE problems, i.e. to solve a PDE with various boundary and initial conditions. The obtained results demand a partial answering of this rather general question. Besides, encapsulated from the generalization task, we examined what the added value of a PINN is compared to a standard NN. To this end, we evaluate and interpret the key findings of the afore presented results by answering these two main questions by six separate subquestions.

1) How many training points are required in order to gain a reliable model? For size 15 (see Fig. 3.2a), the large gap between train and test loss for both the PINN and NN indicate an unsufficient regularization which could lead to a poor generalization. Here, the smaller gap for the PINN intimates that better regularization has been achieved than for the NN. This was to be expected since the PINN loss contains the PDE residual which acts as a regularization mechanism, as mentioned in section 2.2.1 and in [4]. The corresponding solution plots (see Fig. 3.3a-b) and errors depicted in Tables 3.1 and 3.2, allow the conclusion that for the considered problem already 15 training points are sufficient for a PINN to learn the problem. This can be justified regarding the fact that the prediction was performed on a 100×100 mesh, i.e. 10000 data points, from which only 15 were used for training and nevertheless a small (relative L_2) error of 5.7×10^{-3} can be achieved. It should be noticed that no boundary or initial data was used for training which causes the NN to make large errors at the boundaries (up to $\approx$11%). Still, the PINN achieves good prediction accuracy at the boundaries with a maximal L_∞ error of only 2%.

It should be mentioned that this question is highly problem dependent. We considered the one-dimensional heat equation, a linear PDE, with sine as IC and zero Dirichlet BCs. From the mathematical point of view, this problem can be regarded as an "easy" solvable problem due to the linearity of the PDE and since the initial

condition is smooth and infinitely often differentiable. Also, the boundary conditions are simple compared to Neumann or Robin BCs which require derivative evaluations themselves. It can be assumed that for more complex problems the required number of training samples increases. This assumption is supported by Lu et al. [16] where they mentioned that in case of Burger's equation, a nonlinear PDE, it should be considered to put more training points near the sharp front which is characteristic for this PDE such that the model is able to learn the discontinuity well. Also, Raissi, Perdikaris, and Karniadakis note in [4] that "the total number of collocation points needed [...] will increase exponentially" for higher dimensional problems.

2) Do PINNs have better extrapolation capabilities than NNs? At first glance, Figures 3.4a, b suggest that PINNs extrapolate better than NNs due to the smaller error and the fact that the shape of sine is mostly maintained in case of the PINN. However, this suggestion is not generally valid. This becomes clear for increasing size of the training dataset, especially when looking at Figure 3.5 where the relative L_2 error is depicted for all considered sizes. The zick zack pattern showcase that neither NN nor PINN performs better in terms of extrapolation such that no valid answer can be given here. This was to be expected however since in general, NNs do not boast proper extrapolation capabilities [40] and due to PINNs being a special variant of NNs.

3) When are PINNs superior over NNs? In question 1 we already pointed to the difference betweeen PINN and NN for size 15, shown by the losses, solution plots, and the corresponding errors. In this case, the PINN is superior over the NN due to the observed better generalization. This observation can also be seen for case 30 (see Fig. 3.2b), where a still relatively large distance between the PINN and NN test loss is visible. Also, the solution plots (see Fig. 3.3c-d) and errors (Tabs. 3.1, 3.2) lead to a similar conclusion as for size 15. However, from size 40 on, one can see that the PINN and NN perform similar. This is indicated by the losses (see Fig. 3.2c) and undergird by the corresponding values (test loss, rel. L_2 error, maximal rel. L_∞ error) which only differ slightly from each other. For the considered problem (PDE), a size of 40 training samples represents the threshold where the PINN provides no significantly advantage against the respective NN anymore. Although the solution plots and error values for sizes larger than 30 are more promising in case of PINNs, a severe bottleneck can be observed in terms of computational runtime. Already in case of 40 training points, the training process of the NN is about three times faster. This can be tied to the fact that for PINNs the gradient has to be evaluated in each iteration. Figure 3.1b displays that for PINNs the computational time increases

linearly with growing size of the training dataset compared to the NN which shows no significantly increase of time for sizes up to 200. This leads to the conclusion that PINNs are superior over NNs for small sizes of the training dataset where the term "small" has to be seen relatively to the considered problem. This statement is also confirmed by Raissi, Perdikaris, and Karniadakis [4] where they mentioned that one of the key properties of PINNs is that they are able to achieve good prediction results even though it may be only a small amount of data available where standard NNs usually perform worse.

Can a PINN be generalized to solve the 1D heat equation for ...

4) ... fixed initial and various boundary conditions? The results from section 3.2 indicate with the aid of three cases that if the initial condition is fixed, it is indeed possible to generalize a PINN to solve the one-dimensional heat equation for various BCs. In the following, we explain this statement on the basis of these cases in more detail.

Initial Condition: Sine
The solution plots (see Fig. 3.8) and error values depicted in Table 3.6 allow the conclusion that for sine as IC it is possible to generalize a PINN when varying the boundary conditions. This is due to the fact that for all possible boundary values only minor errors are made, especially for cases where no training data was used at all, e.g. l=1.0, r=0.9, as seen in Figure 3.8l. This can also be verified by looking at the relative L_2 errors (see Tab. 3.6) which are all of order 10^{-3}, and the maximal L_∞ errors which for all cases are less than 1%. This bevavior fits the expection made beforehand, regarding the fact that NNs have proper interpolation capabilities in general [40]. The boundary values for which training data was used are in $\{0, 0.2, \ldots, 1\}$. For future work, it would be interesting to consider larger distances between the values, e.g. $\{0, 0.5, 1\}$, and look at the evolution of the resulting generalization errors.

Initial Condition: Line with Discontinuous Boundaries
Due to the discontinuity at the boundaries, large errors up to 63.13% (see Tab. 3.8) occur in the region around the IC. This indicates that too few training points have been selected in the areas where the discontinuities occur and thus the PINN is not able to learn the discontinuity well. It can be assumed that adding more points in those areas can improve the performance since it has already been shown with the example of Burger's equation [4] that a PINN is able to learn discontinuities. A possible reason might also be the fact that no form of discontinuity appears in the

architecture of the network. The hyperbolic tangent which serves as activation function is continuous and infinitely often differentiable. In some future work, one can investigate if this behavior changes when using an activation function which is also not differentiable in some point, e.g. the ReLU function which is not differentiable at $x = 0$. However, although the large errors near the boundaries lead to a larger overall error, the relative L_2 error stays within a reasonable order such that we can argue that in this case the generalization is possible as well.

Additional Nonlinear Term: Radiative Loss
We compare this case (Case 3) to the case with sine as IC (Case 1) since the only difference between these two cases is the nonlinear term. Thus, we can expose appropriately the influence of the nonlinearity on the model. Comparing the solution plots of Case 1 (see Fig. 3.8) and Case 3 (see Fig. 3.13), one can see the nonlinearity causes larger errors in general, especially near the initial condition. A reason for that could be the fact that in these areas there is a great change of the graph of the function as can be seen in Figures 3.14i,j,l. This means that in these areas the gradient is steep and therefore more training points should be set at these locations which is not the case as pictured in Figures 3.13i,l where the used training points are displayed. This was already mentioned in question 1 where we emphasized that for areas with steep gradients more training points are required. However, this can only be done if the shape of function is known in advance, which is generally not the case in practical applications. Nevertheless, the PINN provides good prediction accuracy (see Tab. 3.10) such that it can be stated that for this setting it is possible to generalize the model.

5) ... various boundary and initial conditions? Considering the line plots and corresponding errors it seems that odd functions are easier for the model to learn than even functions. Especially Table 3.12 shows that on the training data the relative L_2 error is for even functions at least one order of magnitude higher than for odd functions. This behavior can be observed on the test data as well. However, this is not a statement that holds in general. Looking for example at Figure 3.20f, one can see an odd function which shows large discrepancies between the prediction and exact solution. Since a similar function contained in the test data (Fig. 3.19d) achieved a L_2 error of 1%, the model architecture seems to be sufficient for the model to learn this kind of function. Thus, it can be assumed that further training can improve the prediction accuracy in this case. Additionally, it appears that very large errors are made for functions whose shape seems to be "easy", e.g. the shape of a parabola (see Fig. 3.20a). Specifically, function No. 140 (Fig. 3.20d) shows a

prediction result which is not even close to optimal providing a L_2 error of 264%, although a similar function contained in the training data (see Fig. 3.17i) has been learned well. A possible reason for that could be the frequency distribution of the functions within the training data. Tendentially, the training data include more odd functions than even. For possible future work, it should be considered to choose a training dataset with an equal distribution of odd and even functions.

The histogram representing the frequency distribution of the (relative L_2) error on the train data (see Fig. 3.16) shows that at least 90% of all initial functions achieve an error of less than 2%. Assuming that an error of 2% can be regarded as a measure of "well-learned", one can say that the model is able to learn these kind of functions. At first glance, this is a promising result. However, regarding the histogram representing the test functions (Fig. 3.21), one can see that 1.5% of all test functions (6 out of 392) have a L_2 error larger than 100%. Although these few functions are not representative for the overall performance of the model, this causes the model to be not reliable. Apart from the fact that further training could improve the overall prediction accuracy, by the obtained results we cannot give a profound answer here concerning the generalization question of this thesis.

6) Is it recommendable to use PINNs for a generalization task as proposed in this thesis? Independent from the fact if PINNs can be generalized in general, the computational cost for the training process is tremendous. Certainly, there is still room for optimization, e.g. improve efficiency of the implementation code or optimizing the training dataset. Still, a duration of several weeks for the training process cannot be ignored. An important point to mention here is, that we only investigated problems in one dimension. For higher dimensions the runtime will grow exponentially which makes this whole process not applicable in practice. For real-world applications we recommend to consider alternative methods concerning the generalization of PDEs for different instances of initial and boundary conditions, respectively.

4.1 Alternative Method: Learning Nonlinear Operators

Due to the necessary gradient evaluation in each iteration the training process is exceedingly time-consuming. Although the presented methods could be further improved in terms of accuracy and runtime, it should be considered to attempt alternative approaches. Dealing with the generalization capabilities of PINNs, we came

across another method tackling a similar generalization task. We briefly introduce the method of *learning nonlinear operators* in order to solve a (partial) differential equation for several instances of the equation. Like standard NNs, only data is required instead of taking the considered equation into account. The basis forms an approximation result which states that a neural network with a single hidden layer can approximate accurately any nonlinear operator [41]. Lu, Jin, and Karniadakis present in [42] how to use this result in practice and propose *deep operator networks* (DeepONets), a data-driven method "to learn (nonlinear) operators accurately and efficiently from a relatively small dataset" [42].

We explain the idea of DeepONets by reference to Figure 4.1. Let X be a Banach space and $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ be two compact sets in X and $\mathbb{R}^d$, respectively. V is a compact set in $\mathcal{C}(K_1)$ and G is a nonlinear continuous operator which maps V into $\mathcal{C}(K_2)$. The network to learn an operator $G\colon u \mapsto G(u)$ takes an input function $u \in V$ and a point $y \in K_2$, and outputs a real number $G(u)(y) \in \mathbb{R}$ (see Fig. 4.1A). In order to work with the input function u numerically, it has to be represented discretely. A straightforward way to do this is to evaluate the function at sufficient but finite many locations $\{x_1, \dots, x_m\}$ which in [42] are called *sensors*. The only requirement they claim is the consistency of the sensors for all input functions u (see Fig. 4.1B). The DeepONet consists of two subnetworks. The architecture is shown in Figure 4.1C. There are p branch networks each taking $(u(x_1), u(x_2), \dots, u(x_m))^T$ as the input and outputting a scalar $b_k \in \mathbb{R}$ for $k = 1, \dots, p$. In addition to the branch networks, there is the trunk network which takes y as the input and outputs $(t_1, \dots, t_p)^T \in \mathbb{R}^p$. The outputs are merged together such that

$$G(u)(y) \approx \sum_{k=1}^{p} b_k t_k.$$

In order to increase the performance by reducing the generalization error [42], biases are added to the last layer of each branch network b_k and also to the last stage

$$G(u)(y) \approx \sum_{k=1}^{p} b_k t_k + b_0.$$

The authors note that in practice, p is at least of the order 10 which results in a network that is computationally expensive. To reduce the number of branch networks, they merge them together into one single branch network (see Fig. 4.1D) which outputs a vector $(b_1, \dots, b_p)^T \in \mathbb{R}^p$. The authors refer to the two types of DeepONets as *stacked* (Fig. 4.1) and *unstacked* (Fig. 4.1) DeepONet. All versions of DeepONets are implemented in DeepXDE [17], a scientific machine learning library

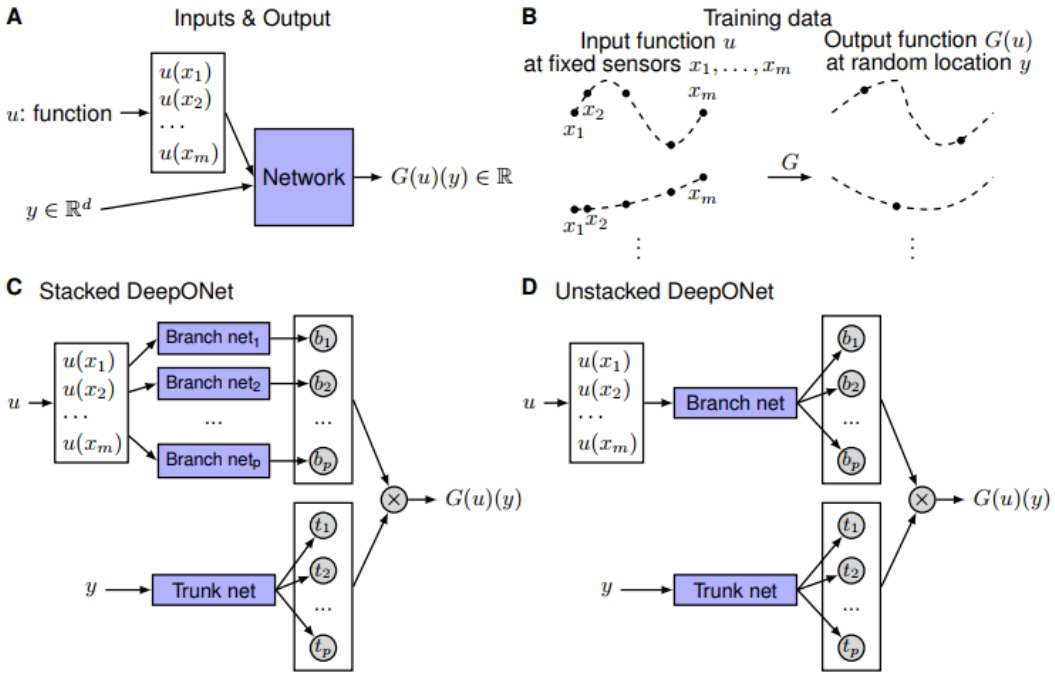


Figure 4.1: Illustration of the problem setup and architectures of DeepONets. (**A**): The network to learn an operator $G\colon u \mapsto G(u)$ takes two inputs $(u(x_1), u(x_2), \ldots, u(x_m))^T$ and y. (**B**): Illustration of the training data. For each input function u we require that we have the same number of evaluations at the same scattered sensors $x_1, x_2, \ldots, x_m$. However, there are no constraints enforced on the number of locations for the evaluation of output functions. (**C**): The stacked DeepONet has one trunk network and p stacked branch networks. (**D**): The unstacked DeepONet has one trunk network and one branch network. Source [42].

which we already introduced in section 2.2.2.

In [42] they have applied DeepONets to four ordinary/partial differential equation problems and showed that by introducing this inductive bias DeepONets can achieve smaller generalization errors compared to fully connected networks. They observed "even exponential error convergence with respect to the training dataset size". Also, the theoretically dependence of approximation error on different factors are derived. Note that the authors emphasize that this method achieves good prediction results when only small data is available. Thus, it can be promising to use this as an alternative to PINNs for the generalization task.

5 Conclusion

We investigated on physics-informed neural networks (PINNs), i.e. neural networks (NNs) which incorporate the considered PDE into the training process. Specifically, we examined the question if PINNs can be generalized in order to solve PDE problems for various instances of boundary and initial conditions.

Since PINNs are NNs which only differ in their training process, we first explained the method of NNs and elucidated the regularization mechanism which is used during training in order to avoid overfitting. Subsquently, we introduced the method of PINNs and illustrated how a PDE is involved within the training process. The PINN loss function is divided into a PDE and a boundary loss. Only the boundary loss considers ground truth data consisting of boundary and initial data. The PDE loss contains the considered PDE and evaluates the required derivatives via AD. It acts as a regularization mechanism such that PINNs can be used when only partial data is available. In this context we also pointed to the existence of the scientific machine learning library DeepXDE which provides several advantages when investigating PDE problems for fixed instances of boundary and initial conditions. It is user-friendly and has several features such as the flexible monitoring of intermediate results due to so-called callback functions. Also, the method of RAR increases the training efficiency by adding more training points in those areas where steep gradients occur. To be able to evaluate our investigations, we produced reference solutions of our problem using the software library FEniCS. FEniCS uses the finite element method, the conventional method to solve PDEs numerically. The user basically has to provide the considered problem in weak form, also called variational formulation.

We used a PINN to solve the one-dimensional heat equation with Dirichlet boundary conditions. In addition to the generalization task proposed in this thesis, we compared a PINN to a NN with the same prerequisites for solving one instance of the PDE, i.e. for fixed initial and boundary conditions. As it turned out, PINNs are superior over NNs in terms of accuracy for a small number of training samples. However, we observed that a PINN requires more computational time than a NN

due to the gradient evaluations in each iteration. This can be a severe bottleneck for higher dimensional problems as the runtime increases exponentially for growing number of input features.

In order to be able to answer the generalization question properly, we processed this task in two steps. First, we fixed the initial condition and trained the model with various instances of boundary conditions. Secondly, we trained the model with a large amount of initial conditions and implicit given boundary conditions. Our numerical results indicate that a PINN can be generalized in the first setting. Due to the long runtime in case of various initial conditions, it is not possible to give an appropriate answer to the generalization question in this case. The training took several weeks and it seems as if further training is still required because of the partially poor predictive performance on both training and test data. Still, it looks promising that PINNs can be generalized in this case considering the results for fixed initial conditions and the fact that there are indeed functions being part of the test data where the model achieved good prediction accuracy. However, if many instances are involved in the training data, one has to take into consideration that this comes along with a massive amount of gradient evaluations which slows down the overall training process. To improve this, one could optimize the selection of functions contained in the training dataset in order to reduce the number of gradient evaluations. Another approach for some future work could be adapting the source code of DeepXDE such that it can handle various instances of initial and boundary data. The implemented RAR method could reduce the required amount of computational resources. Concerning practical applications, it should be considered to attempt other methods when it comes to the generalization of PDEs for various boundary and initial conditions. The method of learning nonlinear operators is a data-driven concept which also aims to achieve good prediction results when only partial data is available.

From the machine learning perspective it is rather unintuitive that for certain parts of the input data no ground truth is actually used or even needed. However, from the scientific point of view it is promising that a well-known data-driven concept, the neural network, is augmented in a way that it takes the analytical representation of the problem into account. PINNs reveal their strengths in settings where traditional solving methods fail to provide reliable predictions due to a lack of data, e.g. unknown boundary or initial data. However, for applications where the training requires a lot of input features or gradient evaluations our results show that it should be consdiered to use different approaches. Nonetheless, for fixed initial conditions it is possible to use PINNs to solve general PDEs for various BCs.

Bibliography

[1] Alexander Maedche et al. „AI-Based Digital Assistants“. In: *Business & Information Systems Engineering* 61.4 (2019), pp. 535–544. ISSN: 1867-0202. DOI: 10.1007/s12599-019-00600-8. URL: https://doi.org/10.1007/s12599-019-00600-8.

[2] Sorin Grigorescu et al. „A survey of deep learning techniques for autonomous driving“. In: *Journal of Field Robotics* 37.3 (2020), pp. 362–386. DOI: https://doi.org/10.1002/rob.21918. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21918.

[3] Akira Taniguchi et al. „Autonomous planning based on spatial concepts to tidy up home environments with service robots“. In: *Advanced Robotics* 35.8 (2021), pp. 471–489.

[4] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. „Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations“. In: (Nov. 2017). arXiv: 1711.10561 [cs.AI].

[5] Atilim Gunes Baydin et al. „Automatic differentiation in machine learning: a survey“. In: *Journal of Machine Learning Research* 18 (2018), pp. 1–43.

[6] Allan Pinkus. „Approximation theory of the MLP model in neural networks“. In: *Acta numerica* 8 (1999), pp. 143–195.

[7] Yeonjong Shin, Jerome Darbon, and George Em Karniadakis. „On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type PDEs“. In: *arXiv preprint arXiv:2004.01806* (2020).

[8] Maziar Raissi et al. „Deep learning of vortex-induced vibrations“. In: *Journal of Fluid Mechanics* 861 (2019), pp. 119–137. DOI: 10.1017/jfm.2018.872.

[9] Zhiping Mao, Ameya D. Jagtap, and George Em Karniadakis. „Physics-informed neural networks for high-speed flows“. In: *Computer Methods in Applied Mechanics and Engineering* 360 (2020), p. 112789. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2019.112789. URL: https://www.sciencedirect.com/science/article/pii/S0045782519306814.

[10] Xiaowei Jin et al. „NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations“. In: *Journal of Computational Physics* 426 (2021), p. 109951. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2020.109951. URL: https://www.sciencedirect.com/science/article/pii/S0021999120307257.

[11] Oliver Hennigh et al. „NVIDIA SimNet™: an AI-accelerated multi-physics simulation framework“. In: (2020). arXiv: 2012.07938 [physics.flu-dyn].

[12] *Heat Transfer Prediction With Unknown Thermal Boundary Conditions Using Physics-Informed Neural Networks*. Vol. Volume 3: Computational Fluid Dynamics; Micro and Nano Fluid Dynamics. Fluids Engineering Division Summer Meeting. V003T05A054. July 2020. DOI: 10.1115/FEDSM2020-20159. eprint: https://asmedigitalcollection.asme.org/FEDSM/proceedings-pdf/FEDSM2020/83730/V003T05A054/6575747/v003t05a054-fedsm2020-20159.pdf. URL: https://doi.org/10.1115/FEDSM2020-20159.

[13] Tongsheng Wang et al. „Reconstruction of natural convection within an enclosure using deep neural network“. In: *International Journal of Heat and Mass Transfer* 164 (2021), p. 120626. ISSN: 0017-9310. DOI: https://doi.org/10.1016/j.ijheatmasstransfer.2020.120626. URL: https://www.sciencedirect.com/science/article/pii/S0017931020335626.

[14] Didier Lucor, Atul Agrawal, and Anne Sergent. *Physics-aware deep neural networks for surrogate modeling of turbulent natural convection*. 2021. arXiv: 2103.03565 [cs.LG].

[15] Shengze Cai et al. „Flow over an espresso cup: inferring 3-D velocity and pressure fields from tomographic background oriented Schlieren via physics-informed neural networks“. In: *Journal of Fluid Mechanics* 915 (2021).

[16] Lu Lu et al. „DeepXDE: A deep learning library for solving differential equations“. In: (July 2019). arXiv: 1907.04502 [cs.LG].

[17] *DeepXDE, a library for scientific machine learning*. https://github.com/lululxvi/deepxde.

[18] Shengze Cai et al. „Physics-informed neural networks for heat transfer problems“. In: *Journal of Heat Transfer* 143.6 (2021), p. 060801.

[19] Shengze Cai et al. „Physics-informed neural networks (PINNs) for fluid mechanics: A review“. In: *arXiv preprint arXiv:2105.09506* (2021).

[20] Liu Yang, Xuhui Meng, and George Em Karniadakis. „B-PINNs: Bayesian physics-informed neural networks for forward and inverse PDE problems with noisy data“. In: *Journal of Computational Physics* 425 (2021), p. 109913. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2020.109913. URL: https://www.sciencedirect.com/science/article/pii/S0021999120306872.

[21] Mark Craven, Johan Kumlien, et al. „Constructing biological knowledge bases by extracting information from text sources.“ In: *ISMB*. Vol. 1999. 1999, pp. 77–86.

[22] Violetta Schäfer. „Automatic Differentiation for High-Dimensional Optimization Problems“. Bachelor Thesis. 2019.

[23] Christopher J.C. Burges Yann LeCun Corinna Cortes. *The MNIST Database of Handwritten Digits*. http://yann.lecun.com/exdb/mnist/. 1998.

[24] Niklas Rottmayer. „Deep Learning with Neural Networks“. Bachelor Thesis. 2019.

[25] Chigozie Nwankpa et al. „Activation Functions: Comparison of trends in Practice and Research for Deep Learning“. In: *CoRR* abs/1811.03378 (2018). arXiv: 1811.03378. URL: http://arxiv.org/abs/1811.03378.

[26] cloud4science. *Notes on Deep Learning and Differential Equations*. June 2020. URL: https://cloud4scieng.org/2020/06/10/notes-on-deep-learning-and-differential-equations/.

[27] Ya-xiang Yuan. „A NEW STEPSIZE FOR THE STEEPEST DESCENT METHOD“. In: *Journal of Computational Mathematics* 24.2 (2006), pp. 149–156. ISSN: 02549409, 19917139. URL: http://www.jstor.org/stable/43694074.

[28] Diederik P. Kingma and Jimmy Ba. „Adam: A Method for Stochastic Optimization“. In: (Dec. 2014). arXiv: 1412.6980 [cs.LG].

[29] Jason Brownlee. Oct. 2021. URL: https://machinelearningmastery.com/gradient-descent-with-momentum-from-scratch/.

[30] Christopher M.Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[31] René Pinnau. „Partial Differential Equations: An Introduction“. Lecture Notes.

[32] Isaac E Lagaris, Aristidis C Likas, and Dimitris G Papageorgiou. „Neural-network methods for boundary value problems with irregular boundaries“. In: *IEEE Transactions on Neural Networks* 11.5 (2000), pp. 1041–1049.

[33] Richard H Byrd et al. „A limited memory algorithm for bound constrained optimization“. In: *SIAM Journal on scientific computing* 16.5 (1995), pp. 1190–1208.

[34] G. N. Wells et al. A. Logg K.-A. Mardal. *Automated Solution of Differential Equations by the Finite Element Method*. Springer. doi.org/10.1007/978-3-642-23099-8. 2012.

[35] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python - The FEniCS Tutorial I*. Springer, 2016.

[36] Bernd Simeon. „Numerical PDEs I“. Lecture Notes.

[37] Oct. 2021. URL: http://www-m15.ma.tum.de/foswiki/pub/M15/Allgemeines/IllposedProblems/WebHome/board4.pdf.

[38] Gabriele Steidl and Ronny Bergmann. „Fundamentals of Mathematical Image Processing“. Lecture Notes. 2017.

[39] André Neubauer. *DFT - Diskrete Fourier-Transformation*. Springer Vieweg, 2011.

[40] Etienne Barnard and LFA Wessels. „Extrapolation and interpolation in neural network classifiers“. In: *IEEE Control Systems Magazine* 12.5 (1992), pp. 50–53.

[41] Tianping Chen and Hong Chen. „Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems“. In: *IEEE Transactions on Neural Networks* 6.4 (1995), pp. 911–917.

[42] Lu Lu, Pengzhan Jin, and George Em Karniadakis. „Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators“. In: *arXiv preprint arXiv:1910.03193* (2019).