Department of Civil, Geo and Environmental Engineering
Chair of Computational Modeling and Simulation
Prof. Dr.-Ing. André Borrmann

Exploring Physics-informed Neural Networks for the Heat Equation

Moritz Jokeit

Master's thesis
for the Master of Science program Computational Mechanics

Author:	Moritz Jokeit
Supervisor:	PD Dr.-Ing. habil. Stefan Kollmannsberger
	Davide D'Angella, M.Sc.
Date of issue:	01. February 2020
Date of submission:	01. August 2020

Involved Organisations

Chair of Computational Modeling and Simulation
Department of Civil, Geo and Environmental Engineering
Technische Universität München
Arcisstraße 21
D-80333 München

Declaration

With this statement I declare, that I have independently completed this Master's thesis. The thoughts taken directly or indirectly from external sources are properly marked as such. This thesis was not previously submitted to another academic institution and has also not yet been published.

München, July 31, 2020

Moritz Jokeit

Moritz Jokeit
Wendelsteinstr. 19
D-83071 Stephanskirchen
e-Mail: m.jokeit@tum.de

Abstract

At its essence, computational mechanics provides numerical solutions to problems that arise from observations of engineered systems or natural phenomena. Partial differential equations commonly govern the resulting mathematical descriptions. Discretization of the problem allows for the approximation of a solution using computational methods. The long-established finite element method and the method of finite differences are among the most popular approaches. However, with problems of increasing complexity, conventional methods often result in exponential growth in the computational effort, thus motivating the search for alternatives. Increased accessibility of deep learning techniques has inspired recent research into investigating their application in physics and engineering. Most publications have employed neural networks to approximate the hidden solution of problems described by partial differential equations. The distinct idea behind those approaches is to incorporate domain knowledge into the learning model to outweigh the usual data scarceness in physical systems. The first promising results have motivated the pursuit of this line of study in the context of computational mechanics. Therefore, this work elaborates on the fundamentals of machine learning and neural networks, and the current literature on learning-based methods in computational mechanics is reviewed. The focus lies on applications of physics-enriched surrogate models. Subsequently, a physics-informed neural network is employed to predict the solution of a heat transfer example. By documenting the implementation and related obstacles, this thesis intends to inform future research on the subject. A discussion on the advantages and drawbacks of learning-based algorithms in the engineering context concludes the thesis.

Contents

Chapter 1

Introduction

The pursuit of artificial intelligence dates back to the time computers in a modern sense were born [RND10]. Ever since, computer science has been partially dedicated to finding the key to intelligent machines. Over the last few years, rapid advancements in the field of machine learning have caused quite a sensation. Algorithms suddenly achieved competing results in tasks thought to be the reserve of human intelligence. Reasons for their success are the vast availability of data paired with a straightforward implementation and powerful graphics hardware. Nowadays, new classes of learning machines are confronted with various complex problems. Whether audio-visual signal processing, weather forecasting, or autonomous driving, data-driven algorithms offer tremendous potential and take computer science a step closer to the ideal of intelligent machines.

The unprecedented success of machine learning has not gone unheeded in other scientific fields. Especially in the presence of adequate data, the application of learning-based algorithms has shown promising results. For instance, image-based medical diagnosis already benefits from outstanding progress in visual object recognition [LRVL$^+$12]. At the same time, the demand for data constitutes a major obstacle for wider interdisciplinary adoption. Physics and engineering applications often lack sufficient information about the underlying problem. Thus, the use of data-driven approaches seems rather naive. Further, engineering sciences have an established, long-standing paradigm of computer-aided problem solving. Finding numerical solutions to problems that arise from observing natural or engineered systems describes the essence of computational mechanics. The solution process commences with formalizing the problem in terms of physical quantities. A set of partial differential equations typically governs the resulting mathematical description. Further, computational methods require a discretization of the problem into finite elements to allow the numerical approximation of continuous variables inside a prescribed domain. Most methods following this scheme have been developed and enhanced over decades, hence guaranteeing a high level of robustness and reliability.

Nevertheless, the generality and simplicity of learning algorithms has sparked interest in the scientific computing community and inspired early attempts toward the data-driven solution of partial differential equations. Incorporating domain knowledge into the learning model allowed it to compensate for the typical data sparsity in physical problems. The computational limitations at that time restricted the application beyond the scope of canonical examples. However, the availability of enhanced programming environments and potent hardware has encouraged researchers from various disciplines to revisit the proposed ideas over the past years. Current publications demonstrate the potential of physics-enriched surrogate models for the inference and identification of partial differential equations [RPK19, SAG$^+$19, NM19]. Most approaches deploy neural network architectures to approximate the hidden solution. Neural networks loosely mimic the neuronal structure of the brain and belong to the most successful group of contemporary learning algorithms. Their ability to learn highly non-linear representations is an excellent prerequisite for modeling physical phenomena.

The work at hand attempts to pursue this line of study by contributing to the adaption of learning-based methods in the domain of computational mechanics. In preparation for further investigations, the first part of the thesis contemplates the advantages and drawbacks of data-driven algorithms. Explaining machine learning basics paves the way for introducing more sophisticated architectures. Since neural networks are of particular interest for the application in engineering problems, their algorithmic framework is described in detail. A simple example elaborates on the implementation of neural networks and provides initial insights into their approximation capabilities. The introduced concepts allow a review of recent literature on data-driven algorithms in physics and engineering. What follows is a detailed study of the most notable publication toward learning solutions of partial differential equations in scarce data regimes [RPK19]. In the proposed method, so-called physics-informed neural networks approximate solutions of exemplary non-linear systems. The promising results are the basis for this thesis to advance the study of physics-enhanced learning models to a heat transfer example. The heat equation in general is of importance to various real-world applications such as the thermal analysis of metal-based additive manufacturing. The numerical modeling of these processes is an ongoing challenge due to the extensive computational effort involved. In cases where conventional methods fail to provide satisfactory approximations, learning-based algorithms might become an attractive alternative. As a preliminary step toward greater adoption, a physics-informed neural network is used to predict the solution to a nonhomogeneous heat equation with temperature-dependent material coefficients. By documenting the implementation process, the work intends to inform future research on the subject. The documentation includes a description of the obstacles encountered as well as possible improvements to the algorithm.

Following this introduction, Chapter 2 leads off with the fundamental concepts of machine learning. Subsequently, Chapter 3 elaborates on neural networks and their corresponding algorithms. After reviewing machine-learning applications in engineering and physics, Chapter 4 shifts the focus toward physics-enriched deep learning models. Chapter 5 documents the implementation of a physics-informed neural network on a heat transfer example. A discussion of results and future research directions concludes the work in Chapter 6.

Chapter 2

Fundamental Concepts of Machine Learning

2.1 Definition

Nowadays, machine learning is arguably the most successful and widely used technique to tackle problems that can not be solved with linearly written programs. In contrast to conventional algorithms following a predefined set of rules, a machine learning algorithm relies on a large amount of data that is observed in nature, handcrafted by humans or generated by another algorithm [Bur19]. A more formal definition by Mitchell states that "a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [Mit97]. Taking image recognition as an example, the task T is to classify previously unseen images, the performance measure P corresponds to the amount of correctly classified images and the experience E includes all images that have been used to train the algorithm. Most machine learning algorithms can be decomposed into the following features: a dataset, a cost function, an optimization procedure, and a parameterized model [GBC16]. Generally, the cost function defines an optimization criterion by relating the data to the model parameters. Further, the optimization procedure searches for the model parameters representing the provided data best. The key difference between machine learning and solving an optimization problem is that the optimized model is then used for predictions on previously unseen data.

2.2 Data Structure

A machine learning algorithm processes a dataset containing a collection of data points, often referred to as examples. Every example consists of one or more features describing the data point in a quantitative manner. In terms of notation, each example can be written as vector $\boldsymbol{x}$, where each entry x_j corresponds to a feature of that example. To take advantage of fast implementations of matrix vector calculus in modern programming languages, all examples are often arranged in a so-called design matrix

$$\boldsymbol{X} = \begin{array}{c} \\ \text{example } 1 \\ \text{example } 2 \\ \vdots \\ \text{example } m \end{array} \begin{array}{c} \begin{array}{cccc} \text{feature } 1 & \text{feature } 2 & \cdots & \text{feature } n \end{array} \\ \left[\begin{array}{cccc} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{array} \right] \end{array}.$$

Each row of the matrix represents an example and each column corresponds to a feature describing the examples [GBC16]. In the case of recognizing gray-scale images, every photo in the dataset is stored as one example vector in the design matrix. Assuming all pictures have a resolution of 50×50 pixels, then every example consists of 2500 features storing the gray-scale value for each pixel.

It is common practice to split the data into different subsets, namely a training set and a test set. The majority of the data, e.g ∼90%, is included in the training set and used for learning the optimal parameters of the model. The remaining part, e.g. ∼10%, is kept for the test set to get an estimate for the model's performance on unseen data. The given percentages are for guidance only, since machine learning literature does not prescribe specific figures and leaves it to the practitioner on how to subdivide the data [Ng20].

2.3 Types of Learning

There exist different types of learning, which are presented in the following sub-sections.

2.3.1 Supervised Learning

Most problems solved by machine learning algorithms fall into the category of supervised learning [Cho18]. In this context, "supervised" indicates, that the algorithm is processing a labeled dataset. Thus, next to the design matrix the dataset comprises a vector $\boldsymbol{y}$ with a label or target y_i for each example. For instance, in image classification tasks each image has previously been annotated receiving a certain category label. The supervised learning algorithm studies the dataset and learns to classify the images into the given categories by comparing its prediction with the given ground truth label.

2.3.2 Unsupervised Learning

The goal of unsupervised learning is to find a structure or more precisely, the probability distribution in the provided data. The data is not labeled and therefore no explicit prediction is possible. However, it can be very useful to apply unsupervised learning to large datasets in order to find inherent structures or repeating patterns in the data. For instance, anomaly detection algorithms are used to identify fraudulent credit card transactions that differ from the usual purchasing behavior of a customer [GBC16].

2.3.3 Semi-supervised Learning

As the term suggests, semi-supervised learning combines the two preceding concepts. In cases where only small samples of the data are labeled, unsupervised learning helps to improve the performance of the supervised learning algorithm [GBC16].

2.3.4 Reinforcement Learning

The basic idea of reinforcement learning is that an algorithm interacts with an environment to learn a certain decision behavior maximizing the expected average reward [Bur19]. It is used for problems involving sequential decision-making in order to fulfill a long-term goal. A group of Google researchers demonstrated the effectiveness of this technique on the example of playing the game of Go [SSS$^+$17], an old and very complex board game which originated in China. Solely by playing games against itself the program reached superhuman abilities and was capable of beating the European Go champion.

2.4 Machine Learning Tasks

The following subsections give a small overview of machine learning tasks with a short introduction of suitable algorithms. This list is by far not complete, however, many problems that arise in practice can be related to one of the following categories.

2.4.1 Regression

Regression is a supervised learning problem with the goal of predicting a numerical value. Basically, a regression algorithm outputs a function that maps a given input to an output, usually in form of a real number. An example is the prediction of house prices based on certain criteria like the area, number of rooms or the age of the house. A more detailed explanation of linear regression can be found in Section 2.5. Further, it is possible to extend the algorithm for the use with polynomials as well as for the prediction of multiple outputs, known as multivariate regression. Decision tree algorithms and neural networks are also used for regression problems. The latter will be introduced in Chapter 3.

2.4.2 Classification

Just like regression, classification tasks belong to the category of supervised learning. Instead of a numerical value, their output takes on a discrete form. In other words, a classification algorithm returns a function that assigns a category to the provided input. Again, the example of classifying the content of an image falls into this class of problems. Even though the name suggests otherwise, logistic regression is a classification algorithm generating a binary output based on the logistic (or sigmoid) function (cf. Chapter 3). Other important algorithms for categorization are the support vector machines and decision trees like random forest or the more advanced gradient boosting approach. In terms of performance at complex

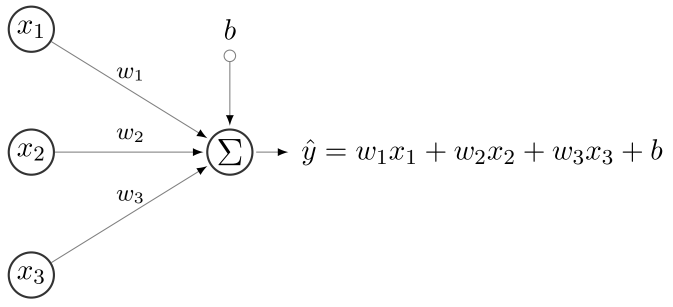


Figure 2.1: Linear regression for a single example $\boldsymbol{x}$ with three input features.

tasks like image recognition, those algorithm have been surpassed by neural networks, but are still used for simpler problems [GBC16].

2.4.3 Clustering

Clustering differs from classification and regression as it is an unsupervised learning task. The algorithm gives feedback about which parts of the data share similarities and therefore belong to the same cluster. A popular choice for clustering is the k-means algorithm that divides the incoming data into k different clusters of examples being close to each other [GBC16].

2.5 Example: Linear Regression

Even though linear regression is a very simple algorithm, it is well suited to explain concepts also applicable to more sophisticated machine learning algorithms. The goal of a regression model is to predict a scalar value $\hat{y} \in \mathbb{R}$ from an input vector $\boldsymbol{x} \in \mathbb{R}^n$. In general, linear regression can be written as

$$\hat{y} = \boldsymbol{w}^T\boldsymbol{x} + b = \sum_{j=1}^{n} w_j x_j + b, \tag{2.1}$$

where $\hat{y}$ represents the target, $\boldsymbol{w}$ is a vector containing the weights and $\boldsymbol{x}$ denotes the example vector (cf. Fig. 2.1). The constant b is called bias, referring to the case when either the weights or the input is close to zero, the output is biased toward b. Each weight w_j is a coefficient that gets multiplied with the corresponding feature x_j. The magnitude and sign of a weight decide about the feature's importance to the prediction $\hat{y}$. For instance, a feature x_j has a corresponding weight w_j which is large in magnitude, then even small changes in x_j alter the prediction $\hat{y}$ by a great amount. The weights and the bias are the parameters of the model and determine how well it performs on the task of predicting target values for new data. In order to get an estimate on the model's future performance a small part of the labeled data is held back to be used for evaluation. This fraction is called test set while the remaining part used for finding the optimal parameters is referred to as training set (cf. Section 2.2).

What is still missing is a way to measure the performance of the model. A common metric for this purpose is the squared error loss. If $\hat{y}$ is the prediction and y the ground truth, the squared error loss is simply defined as $(y - \hat{y})^2$. However, this only provides feedback for a single data point $(\boldsymbol{x}_i, y_i)$. To get one measurement for the performance on the whole data set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{m}$ the mean of the squared error (MSE) of all examples m is calculated as follows

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2. \tag{2.2}$$

The closer the prediction $\hat{y}_i$ gets to the value of y_i, the smaller the error and the better the model is expected to perform. The mean squared error is a popular choice for the cost function. It has a continuous derivative and naturally penalizes large differences between the true target and the prediction [Bur19]. In case of linear regression, the squared error loss even leads to a convex optimization problem, meaning the cost function has only one particular minimum.

Now the question is, how to find the optimal parameters $\boldsymbol{w}^*$ and b^* that yield a good prediction. This task can be interpreted as an optimization problem with the goal to minimize the mean squared error. Given Eq. (2.2) and inserting equation Eq. (2.1), a cost function $C(\boldsymbol{w}, b)$ dependent on the parameters $\boldsymbol{w}$ and b is obtained

$$C(\boldsymbol{w}, b) = \frac{1}{m}\sum_{i=1}^{m}(y_i - (\boldsymbol{w}^T\boldsymbol{x}_i + b))^2. \tag{2.3}$$

With the given cost function it is possible to formulate the optimization problem: Find the model parameters $\boldsymbol{w}$ and b that minimize the cost function $C(\boldsymbol{w}, b)$ or

$$\min_{\boldsymbol{w},b} C(\boldsymbol{w}, b) = \min_{\boldsymbol{w},b} \frac{1}{m}\sum_{i=1}^{m}(y_i - (\boldsymbol{w}^T\boldsymbol{x}_i + b))^2. \tag{2.4}$$

Finding the optimal parameters that lead to a good performance of the model can be interpreted as "learning".

The general concept that separates machine learning from solving an optimization problem is that the former adapts a model using the training examples and then evaluates it on the test set to emulate the future performance on unseen data. This leads to the important distinction between the training error $MSE_{(\text{train})}$

$$MSE_{\text{train}} = \frac{1}{m^{(\text{train})}}\sum_{i=1}^{m^{(\text{train})}}(y_i^{(\text{train})} - \hat{y}_i^{(\text{train})})^2, \tag{2.5}$$

and the test error $MSE_{(\text{test})}$

$$MSE_{\text{test}} = \frac{1}{m^{(\text{test})}}\sum_{i=1}^{m^{(\text{test})}}(y_i^{(\text{test})} - \hat{y}_i^{(\text{test})})^2. \tag{2.6}$$

To gain a better understanding of the whole concept, a one-dimensional linear regression example is given in Fig. 2.2. The goal is to fit the model to the data points by reducing the distance between the regression line and the training examples. Then, as the algorithm is fed with unlabeled data points x_{new}, the fitted line is used for the prediction of y_{new}.

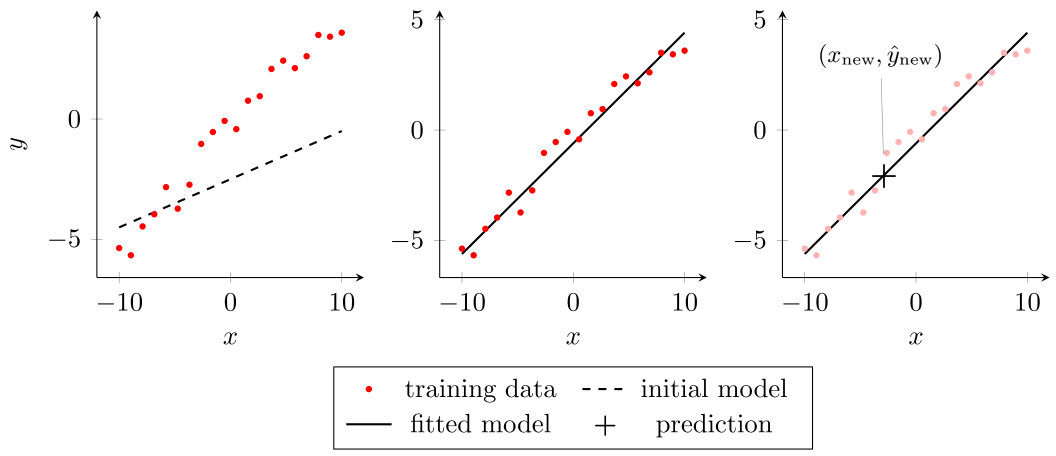


Figure 2.2: Linear regression example. Left: The untrained model is indicated by a dashed line. Middle: Shows the model properly fitted to the training data. Right: The fitted model is used to make a prediction $\hat{y}_{\text{new}}$ for an unknown example x_{new}. Illustrations are inspired by [GBC16].

In the case of linear regression it is possible to solve directly for the optimal model parameters w^* and b^* by setting the gradient of the cost function C to zero. This closed solution to the minimization problem is also known as normal equations [GBC16]. However, due to the fact that a closed solution only exists for very few simple algorithms it is favorable to use a more generally applicable optimization technique. Section 2.8 introduces the gradient descent approach that lays the foundation for many optimization algorithms used in modern machine learning problems. Gradient descent techniques allow the iterative search for a minimum considering a large amount of weight parameters.

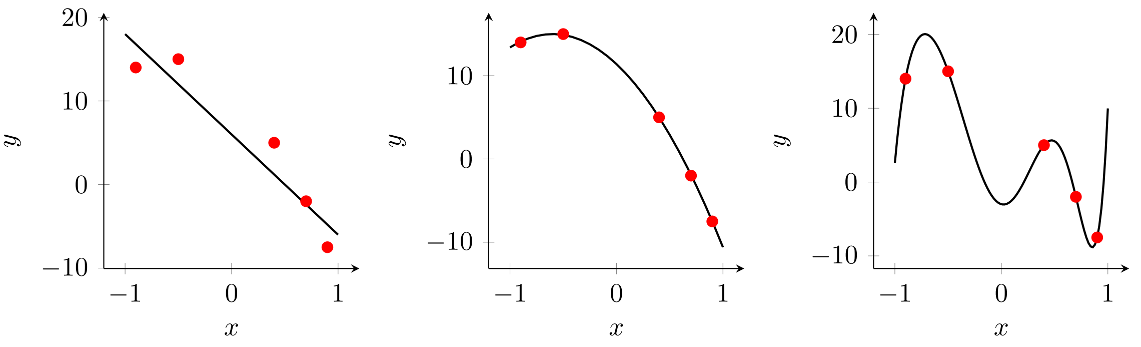
2.6 Overfitting vs. Underfitting

The previous section has introduced the test error, in particular the MSE_{test}, to measure how well a model is expected to perform when confronted with new inputs. The ability to generate good predictions for previously unseen data is called generalization and the associated error is called the generalization error. Therefore, the test error is also considered to be an estimate for the generalization error.

Supervised learning algorithms like linear regression are based on the idea that a low training error also leads to a small test error. Statistical learning theory provides some assumptions to support this idea [GBC16]:

1. The data contains all necessary information to solve the problem.
2. Examples in each dataset are independent of each other.
3. The train and test set are identically distributed, meaning each example is generated with the same probability distribution.

Assuming all given statements are true, the training error ought to be the same as the test error. Since in reality, the data generating probability distribution is unknown a priori, the



(a) Underfitting **(b)** Ideal fitting **(c)** Overfitting

Figure 2.3: Representation of the underfitting and overfitting problem on the example of a one-dimensional regression. Illustrations are inspired by [GBC16].

test error is usually higher than the training error. Figure 2.3 illustrates the two main reasons for that. Firstly, the capacity of a model can be too high, meaning the algorithm chooses a complex function that fits the training data perfectly, but fails to generalize, because it overestimates the importance of noisy data. This behavior is called overfitting (cf. Fig. 2.3c). Secondly, a model with a very low capacity is applied to a highly non-linear problem. For example, if the hypothesis space of a model only contains linear functions, it is not able to represent data that is following quadratic or cubic functions. This case describes underfitting (cf. Fig. 2.3a).

The capacity of a model, more precisely, its ability to fit a wide variety of functions, plays an important role in terms of performance. Choosing the capacity in a way that suits the complexity of the problem, or in other words, finding the balance between underfitting and overfitting is a central challenge in machine learning [GBC16]. The relationship between the training error and the generalization error with respect to the capacity is depicted in Fig. 2.4. As indicated, the optimal capacity is reached when the generalization error is as low as possible.

A way to tackle the problem of underfitting (cf. Fig. 2.3a), is to increase the set of functions an algorithm is allowed to select. Sticking with the example of linear regression, assuming a simple model with a single input x

$$\hat{y} = wx + b. \tag{2.7}$$

The model can be easily extended to include polynomials. Adding x^2 as a new feature a second order polynomial regression of the following form is obtained

$$\hat{y} = w_1 x^2 + w_2 x + b, \tag{2.8}$$

where $x_1 = x^2$ and $x_2 = x$. Introducing more polynomials to the model not only increases the number of features x_i, but also introduces the same amount of parameters w_i. Hence, the algorithm has more possibilities to tune and adapt the model to fit the target function appropriately (cf. Fig. 2.3b). Yet, if the polynomial degree and therefore the capacity becomes too large the model starts to overfit [GBC16]. This is shown in Fig. 2.3c for the case of

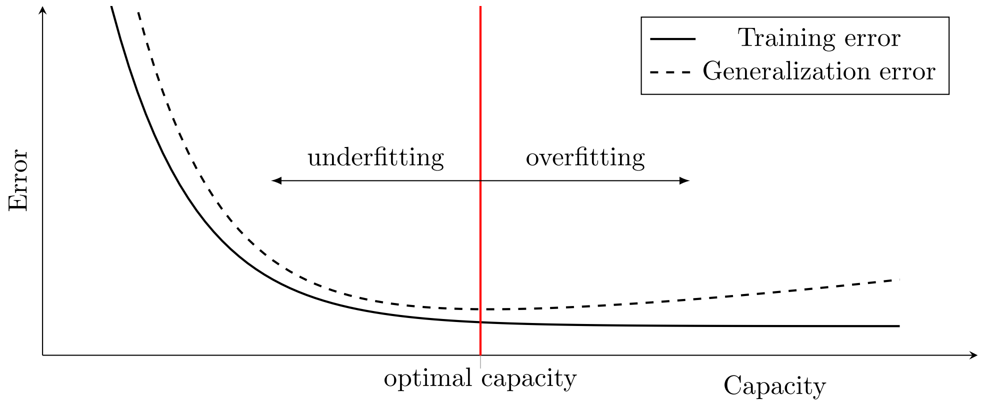


Figure 2.4: Generalization error and training error in relation with the capacity of a model. Illustration inspired by Goodfellow [GBC16].

regression with a polynomial degree of seven

$$\hat{y} = \sum_{i=1}^{7} \left(w_i x^i\right) + b. \tag{2.9}$$

In the literature over- and underfitting are often associated with the terms bias and variance. A high bias means the model produces many mistakes on the training set, so it is the equivalent to underfitting. Coming from statistics, the term variance describes the model's sensitivity to changes in the dataset. If the variance is high, small deviations in the training data result in a very different model. This behavior is closely related to overfitting [Bur19].

2.7 Regularization

Overcoming underfitting or overfitting, i.e. finding the appropriate capacity for a model, is one of the most challenging tasks in machine learning. The previous section showed a possibility to deal with underfitting by increasing the number of features x_i. In the same way overfitting can be reduced when features are removed from the model. However, this option is not very popular, since removing features means that information about the problem gets lost. Another countermeasure against overfitting is adding more examples to the training set [GBC16]. In most cases, it is not feasible or simply impossible to gather more data about the problem. Conversely, removing training data can help in case of underfitting. Yet again, this leads to a loss of information. The influence of the training set size on the the error measures is illustrated in Fig. 2.5.

Since the complexity of the underlying problem is usually unknown, it is hardly possible to choose a model with the appropriate capacity beforehand. So another approach to overcome overfitting is to keep a high capacity, but to introduce a control mechanism called regularizer that prefers the selection of certain functions over others. For instance, the cost function for linear regression (cf. Eq. (2.3)) can be modified to include an additional term that penalizes large weights

$$\tilde{C}(\boldsymbol{w}, b) = C(\boldsymbol{w}, b) + \lambda ||\boldsymbol{w}||_1, \tag{2.10}$$

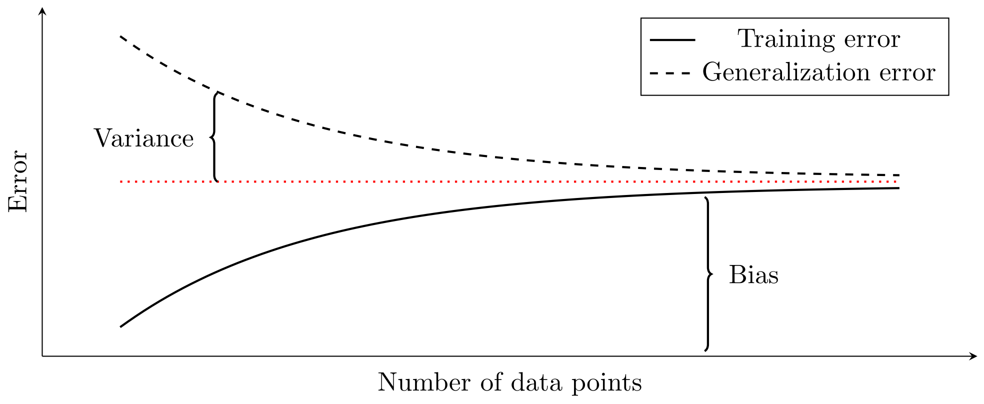


Figure 2.5: Influence of data set size on training and generalization error. The larger the amount of available data, the closer both error measurements get. The distance between the training error and the asymptote marked in red can be interpreted as the bias. In the same way the distance between die generalization error and the asymptote denotes the variance. Illustration inspired by Goodfellow [GBC16].

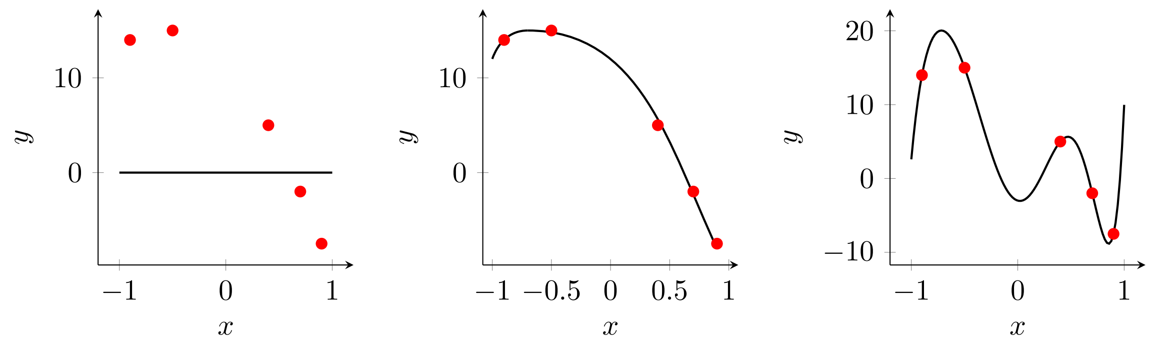
where $||\boldsymbol{w}||_1$ denotes the L^1-norm of the weight vector $\boldsymbol{w}$. Adding a term proportional to the magnitude of the weights forces the algorithm to select a model with smaller weights. In case of linear or polynomial regression, the weights can be understood as the coefficients determining the slope of the function. For instance, the model depicted in Fig. 2.6c exhibits large oscillations, which indicates an underlying function with large slope coefficients. Hence, penalizing large weights is a way to reduce the slope coefficients and the occurrence of oscillations. The influence of the penalty term is controlled by λ. If λ equals zero, the algorithm simply returns the initial linear regression model (cf. Fig. 2.6c). Contrarily, a large value for λ leads to extremely small weights, which basically eliminates the corresponding features resulting in a very simple or sparse model (cf. Fig. 2.6a). In practice, this technique called L^1 or lasso regularization is used for feature selection, a method to remove unimportant features from a model. A similar approach usually producing better results is L^2 or ridge regularization which adds the L^2-norm of the weights instead of their absolute magnitude. For linear regression the regularized cost function can be written as

$$\tilde{C}(\boldsymbol{w}, b) = C(\boldsymbol{w}, b) + \lambda \boldsymbol{w}^T \boldsymbol{w}. \tag{2.11}$$

Another advantage over L^2 regularization is that the penalty term is differentiable [Bur19]. This gains more importance when used in combination with the gradient-based optimization algorithms, which will be introduced in the following Section 2.8.

Apart from model selection and adding a penalty term to the objective function there exist many other techniques to implicitly and explicitly express preferences for a certain solution. A few more will be explained in Chapter 3 about neural networks. Overall, these approaches can be summarized under the term regularization. A definition by Goodfellow states: "Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error (low variance) but not its training error (low bias) [GBC16]." Accordingly, regularization is also known as the bias-variance trade-off [Bur19].

When applying L^1 or L^2 regularization to a model, a new parameter λ is added to the objective function. In contrast to weights and bias, λ is not part of the optimization objective and thus has to be set manually by the user. Generally, all external settings controlling the



(a) Excessive regularization ($\lambda \to \infty$) **(b)** Proper regularization **(c)** No regularization ($\lambda \to 0$)

Figure 2.6: Regularization. Illustration inspired by [GBC16].

behavior of an algorithm are called hyperparameters [Bur19]. They give the opportunity to further improve the performance by doing a so-called hyperparameter tuning. For this purpose, a new subset from the training data has to be selected, since the test set can not be used for any parameter optimization. Therefore, the training data is split into a set for training the standard model parameters and a validation set used for tuning the hyperparameters.

2.8 Optimization Techniques

Optimization plays an essential role in many machine learning algorithms. Generally, the goal of an optimization is to minimize or maximize an optimization criterion or objective function $C(\boldsymbol{\Theta})$ by altering $\boldsymbol{\Theta}$ [GBC16]. In the context of machine learning, this function is often called cost function, and $\boldsymbol{\Theta}$ denotes the parameters of a model, normally the weights $\boldsymbol{w}$ and biases b. A lot of research is dedicated toward finding efficient methods, that help to determine the optimal parameters $\boldsymbol{\Theta}^*$. When the optimization criterion is differentiable, a popular choice is the iterative gradient descent algorithm [GBC16]. A gradient descent approach finds a local minimum of a function by taking steps proportional to the negative gradient of the function at a given point. The gradient points in the direction of steepest ascent, hence a small step in the opposite direction leads to a minimization of the function.

The algorithm is commonly used for neural networks but can be also applied to support vector machines or linear regression. In the latter case, the optimization criterion is convex meaning the function only has one global minimum. For neural networks the optimization problem is non-convex, though it is more likely to converge to a local minimum [GBC16].

As mentioned in Section 2.5, it is possible to analytically solve for the optimal parameters of linear regression. Nevertheless, it serves as a suitable example to explain the basic steps of gradient descent. To keep things simple, a linear regression model with a scalar input x, a corresponding weight w and a scalar bias b is chosen

$$\hat{y} = wx + b. \tag{2.12}$$

Again, the goal is to find the optimal parameters w^* and b^* that minimize the mean squared

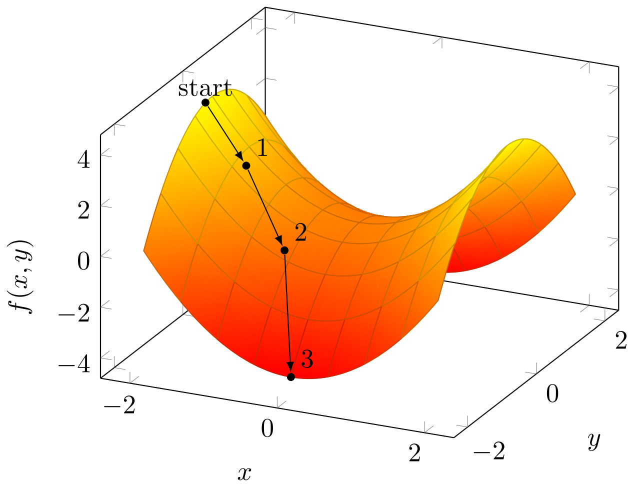


Figure 2.7: Gradient descent on a surface defined by $f(x, y) = x^2 - y^2$. Depending on the point of initialization gradient descent follows the steepest direction toward a minimum of the underlying function.

error. The resulting cost function can be written as

$$C = \frac{1}{m}\sum_{i=1}^{m}(y_i - (wx_i + b))^2. \tag{2.13}$$

Now the partial derivatives for each parameter have to be calculated

$$\begin{aligned}\frac{\partial C}{\partial w} &= \frac{1}{m}\sum_{i=1}^{m} -2x_i\left(y_i - (wx_i + b)\right),\\ \frac{\partial C}{\partial b} &= \frac{1}{m}\sum_{i=1}^{m} -2\left(y_i - (wx_i + b)\right).\end{aligned} \tag{2.14}$$

If more parameters are involved, it is useful to rewrite the partial derivatives in form of gradient ∇C. The gradient is nothing else than a vector storing all the partial derivatives with respect to all parameters of a function. During each iteration step or epoch, to use machine learning terminology, the parameters of the model get updated according to the following rules

$$\begin{aligned}w &\leftarrow w - \alpha\frac{\partial C}{\partial w},\\ b &\leftarrow b - \alpha\frac{\partial C}{\partial b}.\end{aligned} \tag{2.15}$$

In the first step, the parameters are initialized as zero and then updated in each epoch until convergence is reached. The learning rate α is a hyperparameter (cf. Section 2.7) and controls the step size of each update [Bur19].

A graphical representation of gradient descent is depicted in Fig. 2.7. It illustrates how the algorithm follows the direction of steepest descent toward a minimum. If the starting point is initialized differently, the algorithm might descent to the minimum on the other side of the saddle point. This shows the difficulties to find a global minimum in non-convex optimization problems.

A major drawback of gradient descent is its sensitivity to the choice of learning rate α. If α is too large, the algorithm starts to oscillate or even fails to converge at all. Conversely, a small α value leads to an extremely low convergence rate [Bur19].

Another issue is that with an increasing number of examples the computational costs for each epoch grow as well, because all partial derivatives are evaluated for the whole training set of size m. Since the whole training set is used, the method described above is often referred to as full batch gradient descent. Alternatively, the parameter update can be performed instantly after computing the gradient for a single example in the training data. This introduces stochasticity to the algorithm, because the gradient of a single example might indicate a substantially different direction than the gradient computed for the whole training batch [GBC16]. However, in practice, stochastic or sometimes called on-line gradient descent often shows better convergence properties, especially when combined with a adaptive learning rate. A popular and widely used compromise between full batch and stochastic gradient descent (SGD) is mini-batch gradient descent . The idea is to find an approximation of the gradient by evaluating the gradient just for a small sample of the data, a so-called mini-batch. Instead of evaluating the gradient for the whole training set, it is replaced by an estimator that was computed on a sample with fixed size. The differences between the implementation of the three possible optimization strategies are described in Section 3.5 on the example of neural networks.

Other improved versions of stochastic gradient descent include Adagrad, which automatically adapts the learning rate, and the momentum method that accelerates stochastic gradient descent by selecting the relevant direction and thus, reduces oscillations [Bur19].

Chapter 3

Neural Networks

For supervised learning tasks artificial neural networks (ANNs) are the state of the art algorithmic architecture [Mar19]. Inspired by the neurons of the brain, an early form, known as perceptron , was created by Frank Rosenblatt in 1958. After the late 1960s, the development stagnated due to the lack of computational power and efficient methods for network training. With the introduction of the backpropagation algorithm in 1986, learning capabilities of neural networks improved significantly. Nevertheless, their application only became practical in the early 2000s on the hardware available at this time. A major breakthrough leading to more attention was the success of a deep neural network by Krizhevsky et al. that won the image recognition challenge ImageNet in 2012 by a large margin [DBDJH14]. The success was also driven by developments in computer graphics that allowed the exploitation of graphical processing units (GPUs) for the training process.

Since then, artificial neural networks have been applied to solve a great variety of problems. Typical tasks include speech, image and natural language processing, autonomous driving, playing board and computer games, algorithmic trading or weather forecasting. Recently, also physicists and engineers started to investigate how conventional methods can benefit from the capabilities of neural networks [CCC+19]. Even though fundamentals in this still young area of research have been established, most results of modern neural networks are based on empirical studies and heuristics [MBW+19]. This chapter introduces the simplest form of a neural networks and explain its fundamentals on an illustrative example. Section 3.8.1 and Section 3.8.2 are devoted to more advanced neural network architectures that were designed to perform exceptionally well in specific tasks such as image or speech recognition, language translation or time series forecasting.

3.1 Feed-forward Neural Network

When seen as a black-box, a neural network is like any other supervised learning model just a parameterized function defining a mapping $y = f_{NN}(x)$. A particularity of neural networks is that they are typically composed of many nested functions. For instance, three functions f_1,f_2 and f_3 might form the mapping $f_{NN}(x) = f_3(f_2(f_1(x)))$, where each function f_l represents a layer of the network. The information in form of input x flows from the input layer through an arbitrary number of so-called “hidden” layers to the output layer, hence the name feed-

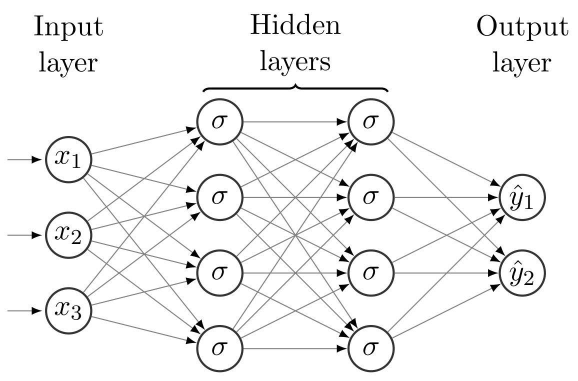


Figure 3.1: A fully-connected feed-forward neural network.

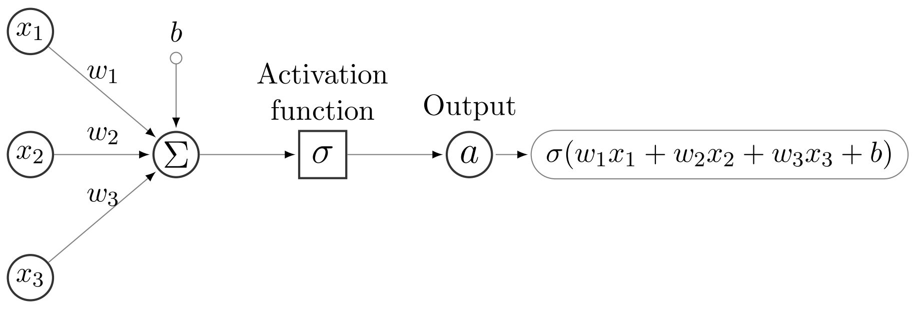


Figure 3.2: A single neuron with three inputs.

forward neural network. The number of layers defines the depth, whereas the amount of hidden neurons determine the width of a network. In this context the term "deep" learning often refers to neural networks with more than one hidden layer [GBC16]. A simple network architecture is depicted in Fig. 3.1. The circles represent the basic units called "neurons", whereas the connections can be interpreted as weights that control the importance of their inputs. If every neuron from the previous layer is connected with each neuron of the next layer, then the neural network is said to be "fully-connected". A single neuron takes a vector of features $\boldsymbol{x}$ and produces a scalar output $a(\boldsymbol{x})$, which serves as an input for the neurons in the next layer. The output of a neuron can be decomposed into two operations. First, the input vector $\boldsymbol{x}$ is transformed into

$$z = \boldsymbol{w}^T\boldsymbol{x} + b \tag{3.1}$$

with a neuron-specific weight $\boldsymbol{w}$ and bias b. Second, the neuron's output $a(\boldsymbol{x})$ is computed by applying the non-linear function σ to the resulting scalar z (cf. Fig. 3.2)

$$a(\boldsymbol{x}) = \sigma(z). \tag{3.2}$$

The function σ is called activation function and is usually chosen to be the same for all neurons [GBC16]. Without the use of a non-linear activation function, the model would not be able to represent any non-linearities in the data. The reason is, regardless of how many linear transformations are applied to an input, the output would still remain a linear function of the input. Typical candidates for the activation $\sigma(z)$ are the hyperbolic tangent, the sigmoid function and the rectified linear unit, short ReLU (see Fig. 3.3). A more detailed discussion about the properties of the different activation functions is provided in Section 3.4.

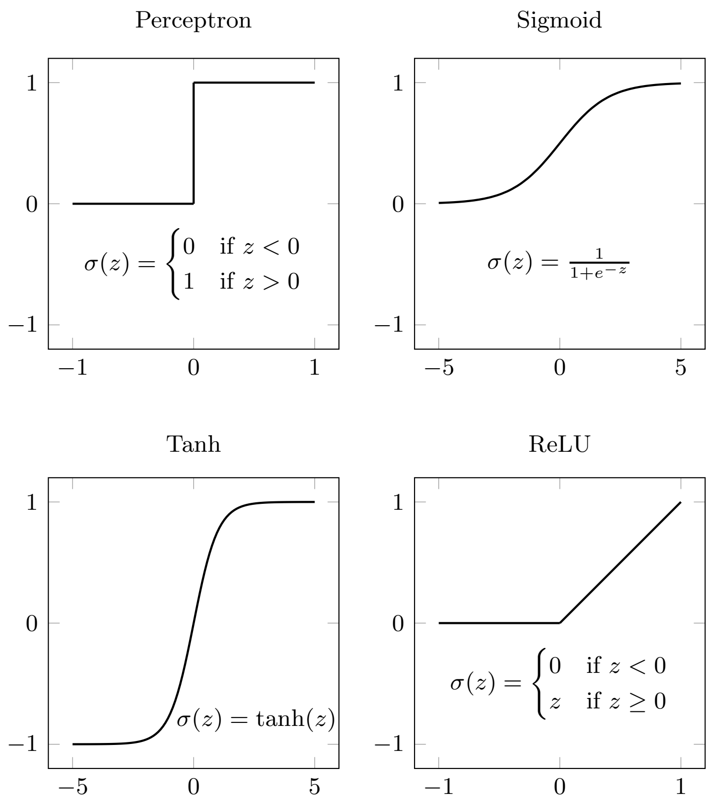
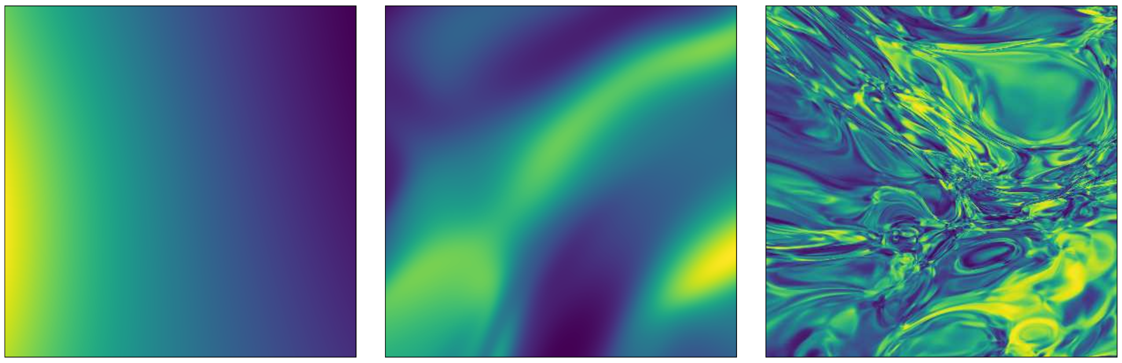


Figure 3.3: Common activation functions

Depending on the function chosen for the output neurons, neural networks are able to solve either regression or classification tasks. For regression, simply applying a linear function in the output layer leads to a result in form of a real number. In case of classification, a common choice is the sigmoid function to enforce a binary encoding of the output that can be easily transformed into labels.

Generally, a fully-connected feed-forward neural networks with one hidden layer is capable of approximating any continuous multi-input/multi-output function with arbitrary precision, given that the hidden layer contains a sufficient amount of neurons. Known as the universal approximation theorem, this hypothesis was formally proven by multiple researchers independently, e.g. in "Approximation by superpositions of a sigmoidal function" by Cybenko, just to mention one of the first references [Cyb89]. A graphical and very intuitive explanation of the universality theorem can be found in chapter four of Nielsen's online-book [Nie15, Chapter 4]. The basic idea is that hidden neurons allow the generation of step functions with arbitrary offsets and heights that can be superpositioned to construct any arbitrary function. However, the whole concept is rather of theoretical importance. In practice, the usage of deep networks with multiple hidden layers is preferred. Multi-layer architectures exhibit the same representational power as comparable wide networks with a single hidden layer, while being computationally more efficient in the training process, as a paper by Mhaskar et al. suggested [MLP16]. Nevertheless, this topic is controversially discussed and continues to be an area of active research [MBW$^+$19].

When dealing with discontinuous functions, a feed-forward neural network can provide a sufficient continuous approximation [Nie15]. Nevertheless, their applicability for discontinuous



(a) 1 hidden layer with 100 neurons **(b)** 10 hidden layers with 100 neurons each **(c)** 20 hidden layers with 100 neurons each

Figure 3.4: Output visualization of a multi-layer neural network. It has two inputs corresponding to the x and y coordinates of a picture and one output defining the color at the pixel (x, y). The parameters of the network are randomly initialized and are not trained. The code used for generating the pictures was developed by Marquardt [Mar17].

problems is limited. The solution often exhibits oscillations around the discontinuities similar to the Gibbs phenomenon observed in Fourier series approximations of discontinuous functions. In "Constructive Approximation of Discontinuous Functions by Neural Networks", Llanas et al. propose a way to overcome this shortcoming and show an almost uniform approximation of a piece-wise continuous function by a single hidden-layer feed-forward neural network [LLS08].

In order to give an illustrative example of the representational capabilities, a neural network is used to generate color plots. The outputs for different numbers of hidden layers are compared in Fig. 3.4. All networks have two inputs, represented by the horizontal and vertical image axis as well as the corresponding output that controls the color value of the image. With an increasing number of layers, the network is able to show more complex functions. These images were generated with random weights and biases and the networks have not been trained for approximating a specific function or image yet.

3.2 Forward Propagation

Before proceeding with the training process of a neural network, it is important to get a better understanding of how the data is propagated from the input to the output layers. A very simple example network with two inputs x_1 and x_2, one hidden layer and a single output $\hat{y}_1$ is depicted in Fig. 3.5.

The goal is to compute the prediction $\hat{y}_1$ of the neural network for one given example

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \end{bmatrix}$$

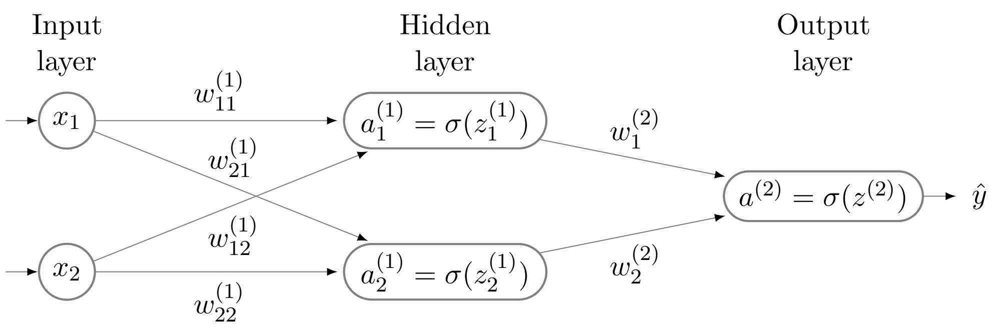


Figure 3.5: A simple feed-forward neural network example.

and weights

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix} = \begin{bmatrix} 1 & -3 \\ -2 & 1 \end{bmatrix}, \quad \boldsymbol{w}^{(2)} = \begin{bmatrix} w_1^{(2)} \\ w_2^{(2)} \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}.$$

For the sake of simplicity, the biases for each neuron are set to zero and thus are neglected in the following calculations. Furthermore, the activation function for all neurons is chosen to be $\sigma(\boldsymbol{z}) = (z_i)^2$. The input vector $\boldsymbol{x}$ is fed to the network and every entry is stored in the corresponding neuron of the input layer without any further modifications. As a first step toward computing the output of the hidden layer, the weighted sum of the inputs is calculated and stored in the variable $\boldsymbol{z}^{(1)}$

$$\boldsymbol{z}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{x} = \begin{bmatrix} 1 & -3 \\ -2 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ 3 \end{bmatrix} = \begin{bmatrix} -4 \\ -5 \end{bmatrix}.$$

Then, the activation function σ is applied element-wise to the resulting vector $\boldsymbol{z}^{(1)}$ to get the output $\boldsymbol{a}^{(1)}$ of the hidden layer

$$\boldsymbol{a}^{(1)} = \sigma(\boldsymbol{z}^{(1)}) = \begin{bmatrix} \sigma(-4) \\ \sigma(-5) \end{bmatrix} = \begin{bmatrix} 16 \\ 25 \end{bmatrix}.$$

Now, the output $\boldsymbol{a}^{(1)}$ serves as the input for the neurons of the next layer. Again, the linear transformation $\boldsymbol{z}^{(1)}$ is computed first

$$z_1^{(2)} = (\boldsymbol{w}^{(2)})^T \boldsymbol{a}^{(1)} = \begin{bmatrix} 2 & -1 \end{bmatrix} \begin{bmatrix} 16 \\ 25 \end{bmatrix} = 7.$$

Finally, the prediction of the neural network $\hat{y}$ is computed by applying the activation function to $z_1^{(2)}$

$$a_1^{(2)} = \sigma(z_1^{(2)}) = \sigma(7) = 49 = \hat{y}_1$$

To generalize the computations from the example, the output for the current neuron j in the l-th layer is calculated by the following formula [Nie15]

$$a_j^{(l)} = \sigma(z_j^{(l)}) = \sigma\left(\sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}\right), \tag{3.3}$$

where $a_k^{(l-1)}$ is the output of the k-th neuron from the previous layer $l-1$ and $w_{jk}^{(l)}$ is the weight connecting the k-th neuron with the current neuron j. After adding the neuron-specific bias $b_j^{(l)}$ to the weighted sum of all outputs k from the previous layer, the non-linear activation function is applied [Nie15].

Using only matrix and vector notation Eq. (3.3) yields [Nie15]

$$\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{z}^{(l)}) = \sigma(\boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}). \tag{3.4}$$

The images in Fig. 3.4 were generated in very similar fashion, namely by forward-propagating arbitrary inputs through a feed-forward neural network with random weights and biases. More complex images were achieved only by adding more hidden layers and neurons to the network architecture.

3.3 Backpropagation

Up to this point, the neural network was only able to transform an input into an output depending on the randomly initialized set of weights and biases. Meaning the network has not utilized any data to learn the optimal parameters w^* and b^* that generate a desired output. As for the example of linear regression, learning from data can be formulated as an optimization problem which requires the definition of a suitable cost function. Once again the mean squared error loss is chosen to quantify the prediction accuracy of the model. The cost function takes on the familiar form

$$C = \frac{1}{2m}\sum_i (y_i - \hat{y}_i)^2, \tag{3.5}$$

but introducing the factor $\frac{1}{2}$ to simplify the expression of the derivative is needed for later calculations. To find optimal network parameters, the cost function is minimized with the gradient descent method that was introduced in Section 2.8. Essential for the iterative parameter update (cf. Eq. (2.15)) is the computation of the gradient of the cost function with respect to the weights and biases. Backpropagation is an efficient technique for determining the partial derivatives of graph-structured functions and in particular neural networks. It belongs to the field of automatic differentiation that deals with the algorithmic computation of derivatives [GBC16]. In the following the general form of the back-propagation algorithm is derived and then illustrated on the example network from the previous section (cf. Fig. 3.5).

If the cost function C is seen as an average over the sum of cost functions C_i of the individual examples i

$$C = \frac{1}{m}\sum_i C_i, \tag{3.6}$$

then the cost C_i for one example i can be written as

$$C_i = \frac{1}{2}(y_i - \hat{y}_i)^2 = \frac{1}{2}(y_i - a_i^{(L)})^2, \tag{3.7}$$

where $a_i^{(L)}$ denotes the output of the last layer L and simultaneously the output $\hat{y}_i$ of the neural network. Further, the computation of the derivatives will be only performed for a single example. Thus, the index i in Eq. (3.7) is dropped to increase the readability. As mentioned the goal of the back-propagation algorithm is to compute the partial derivatives of the cost function with respect to the weights $\frac{\partial C}{\partial w_{jk}^{(l)}}$ and biases $\frac{\partial C}{\partial b_j^{(l)}}$.

Recalling Eq. (3.3) the output of a layer l is defined as

$$a_j^{(l)} = \sigma(z_j^{(l)}), \tag{3.8}$$

where

$$z_j^{(l)} = \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)} \tag{3.9}$$

is computed using the outputs of the previous layer $a_k^{(l-1)}$.

Now, rewriting $\frac{\partial C}{\partial w_{jk}^{(l)}}$ yields

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}, \tag{3.10}$$

with

$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \frac{\partial}{\partial w_{jk}^{(l)}} \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)} = a_k^{(l-1)}. \tag{3.11}$$

Similarly, $\frac{\partial C}{\partial b_j^{(l)}}$ is computed as follows

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)}, \tag{3.12}$$

since

$$\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \frac{\partial}{\partial b_j^{(l)}} \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)} = 1. \tag{3.13}$$

Equation Eq. (3.10) and Eq. (3.12) introduce a new variable $\delta_j^{(l)}$ that describes the sensitivity of the cost function toward a change in the neuron's weighted input $z_j^{(l)}$. If $\delta_j^{(l)}$ for each neuron j in layer l is known, the derivative $\frac{\partial C}{\partial w_{jk}^{(l)}}$ can be computed simply by multiplying $\delta_j^{(l)}$ of the j-th neuron of the current layer l with the output $a_k^{(l-1)}$ of the k-th neuron from the previous layer $(l-1)$. The derivative with respect to the bias $\frac{\partial C}{\partial b_j^{(l)}}$ directly equates to $\delta_j^{(l)}$.

In a next step, an expression for $\delta_j^{(L)}$ in the output layer L is derived

$$\delta_j^{(L)} = \frac{\partial C}{\partial z_j^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}) = -(y_j - \sigma(z_j^{(L)}))\sigma'(z_j^{(L)}), \tag{3.14}$$

recalling that $a_j^{(L)} = \sigma(z_j^{(L)})$ and inserting

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \frac{\partial}{\partial z_j^{(L)}} \sigma(z_j^{(L)}) = \sigma'(z_j^{(L)}). \tag{3.15}$$

What is still missing is a general term for $\delta_j^{(l)}$ in an arbitrary layer l. Utilizing the chain rule $\delta_j^{(l)}$ can be expressed in terms of $\delta_k^{(l+1)}$

$$\delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}} = \sum_k \frac{\partial C}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \delta_k^{(l+1)}. \tag{3.16}$$

With $a_j^{(l)} = \sigma(z_j^{(l)})$ and the definition of $z_k^{(l+1)}$ being

$$z_k^{(l+1)} = \sum_j w_{kj}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} = \sum_j w_{kj}^{(l+1)} \sigma(z_j^{(l)}) + b_k^{(l+1)}, \tag{3.17}$$

the derivative with respect to $z_j^{(l)}$ can be calculated as

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = w_{kj}^{(l+1)} \sigma'(z_j^{(l)}). \tag{3.18}$$

Substituting Eq. (3.18) into Eq. (3.16) finally yields

$$\delta_j^{(l)} = \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)} \sigma'(z_j^{(l)}). \tag{3.19}$$

With equations Eq. (3.10), Eq. (3.14) and Eq. (3.19) all expressions to calculate the partial derivatives of the simple example network are available. Since the biases in the example are chosen to be zero, the computation of the derivatives has to be executed with respect to the four entries of $\boldsymbol{w}^{(1)}$ and two entries of $\boldsymbol{w}^{(2)}$. This also implies that Eq. (3.12) is not needed. The single data point considered in this example consists of the input vector $\boldsymbol{x} = \left[\begin{smallmatrix}5\\3\end{smallmatrix}\right]$ and the target $y = 46$. In order to compute the delta for the output neuron $\delta^{(L)}$, the derivative of the activation function $\sigma'(z) = 2z$ and the results for $z_1^{(2)}$ from the previous section are inserted into equation Eq. (3.14) resulting in

$$\delta_1^{(2)} = -(y - \sigma(z_1^{(2)}))\sigma'(z_1^{(2)}) = -(46 - 49)\sigma'(7) = 42.$$

This intermediate result is then propagated backwards to compute $\delta_1^{(1)}$ and $\delta_2^{(1)}$ of the hidden layer using equation Eq. (3.19)

$$\delta_1^{(1)} = w_1^{(2)} \delta_1^{(2)} \sigma'(z_1^{(1)}) = 42 * 2 * \sigma'(-4) = -672,$$

$$\delta_2^{(1)} = w_2^{(2)} \delta_1^{(2)} \sigma'(z_2^{(1)}) = 42 * (-1) * \sigma'(-5) = 420.$$

Finally, the partial derivatives are calculated with equation Eq. (3.10) using the previously computed δ-values

$$\frac{\partial C}{\partial w_{11}^{(1)}} = \delta_1^{(1)} x_1 = -672 * 5 = -3360,$$
$$\frac{\partial C}{\partial w_{12}^{(1)}} = \delta_1^{(1)} x_2 = -672 * 3 = -2016,$$
$$\frac{\partial C}{\partial w_{21}^{(1)}} = \delta_2^{(1)} x_1 = 420 * 5 = 2100,$$
$$\frac{\partial C}{\partial w_{22}^{(1)}} = \delta_2^{(1)} x_2 = 420 * 3 = 1260,$$
$$\frac{\partial C}{\partial w_{1}^{(2)}} = \delta_1^{(2)} a_1^{(1)} = 42 * 16 = 672,$$
$$\frac{\partial C}{\partial w_{2}^{(2)}} = \delta_1^{(2)} a_2^{(1)} = 42 * 25 = 1050.$$

The partial derivatives can now be used to update the parameters according to the update rule from Eq. (2.15) in Section 2.8:

$$\boldsymbol{W}_{\text{new}}^{(1)} = \boldsymbol{W}^{(1)} - \alpha \nabla \boldsymbol{C}_{\boldsymbol{W}^{(1)}}, \tag{3.20}$$
$$\boldsymbol{w}_{\text{new}}^{(2)} = \boldsymbol{w}^{(2)} - \alpha \nabla \boldsymbol{C}_{\boldsymbol{w}^{(2)}}. \tag{3.21}$$

Updating the weights with a learning rate $\alpha = 0.001$ yields

$$\boldsymbol{W}_{\text{new}}^{(1)} = \begin{bmatrix} 1 & -3 \\ -2 & 1 \end{bmatrix} - 0.001 \begin{bmatrix} -3360 & -2016 \\ -2100 & 1260 \end{bmatrix} = \begin{bmatrix} 4.36 & -0.984 \\ 0.1 & -0.26 \end{bmatrix} \tag{3.22}$$

and

$$\boldsymbol{w}_{\text{new}}^{(2)} = \begin{bmatrix} 2 \\ -1 \end{bmatrix} - 0.001 \begin{bmatrix} 672 \\ 1050 \end{bmatrix} = \begin{bmatrix} 1.328 \\ -2.05 \end{bmatrix}. \tag{3.23}$$

More intuitive explanation of back-propagation can be found in the second chapter of Nielsen's online book "Neural Networks and Deep Learning" that was used as reference for the formal derivation of the algorithm [Nie15, Chapter 5]. Another illustrative approach is described in Christopher Olah's blog post [Ola15].

3.4 Activation Function

As observed in the preceding section, the activation function needs to be differentiable in order to train neural networks with gradient-based methods. Inspired by the biological archetype, the first neural networks, known as perceptrons, used a Heaviside step function to imitate an active (= 1) or inactive neuron (= 0) (see Fig. 3.3). However, the derivative of a step function is zero everywhere except at the jump and thus makes it impractical for training a network. Until recently, continuous functions like the hyperbolic tangent or the sigmoid, also known as the logistic function, were popular choices for the activation σ. Unfortunately, with increasing weights, those functions tend to saturate, resulting in very small gradients that prevent the network from further learning [GBC16]. To resolve this problem of "vanishing gradients", a new type of activation functions was introduced. Rectified linear units or short ReLU (cf. Fig. 3.3) are now the default recommendation for the use with most feed-forward neural networks. They help to reduce saturation, since the function grows linearly for positive inputs. The fact that the rectified linear function $\sigma(z) = \max\{0, z\}$ is not differentiable at $z = 0$, is not a major issue in practice. Depending on the software implementation either the left derivative 0 or right derivative 1 is chosen. Due to the numerical errors of computers, the occurrence of $z = 0$ is very unlikely and can be neglected regarding the training process of neural networks. Generally, the choice of the activation function has a considerable impact on the performance and especially on the learning rate of a model, but usually depends on the specific problem and the experience of the machine learning practitioner [Ng20].

3.5 Learning Algorithm

Within the previous sections all relevant information has been provided to build a neural network architecture. To summarize, the following aspects are required to train the network for a supervised learning task [Cho18]:

- input data $\boldsymbol{X}$ and corresponding targets $\boldsymbol{y}$, divided into a test, training and validation set,
- network topology, defined by the input, output and hidden layers, the corresponding number of neurons, and their connections,
- an activation function σ,
- a loss function C, which defines the feedback signal used for learning,
- a way to compute the gradients w.r.t the network parameters, e.g. backpropagation,
- an optimizer with learning rate α, which determines how learning proceeds.

Algorithm 1 roughly describes the learning algorithm for a feed-forward neural network that regresses a scalar value $\hat{y}$ from a given input $\boldsymbol{x}$.

Instead of using full-batch gradient descent, the parameters can be updated with a stochastic or mini-batch gradient descent step. In case of stochastic gradient descent, the parameter update is executed inside the loop over all examples. This means a gradient descent step is taken each time an example is propagated through the network (cf. Algorithm 2).

The most popular optimization approach in practice is mini-batch gradient descent. Instead of computing the gradients averaged over the whole training set, the gradients are evaluated just for a small part of the training data, the so-called mini-batch. This allows to process data-sets more efficiently, when they are too big to fit the memory as whole. Apart from that, mini-batch gradient descent is preferred, because it is said to introduce a regularizing effect [WM03].

Algorithm 1 Training a neural network with full-batch gradient descent. The inner loop is only displayed for a better understanding. Normally, the loop over the examples is vectorized for more efficient computations.

```
Require: training data $\boldsymbol{X}$, targets $\boldsymbol{y}$
  define network architecture (input layer, hidden layers, output layer, activation function)
  set learning rate $\alpha$
  initialize weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$
  for all epochs do
      for example $i \leftarrow 1$ to $m$ do
          apply forward propagation: $\hat{y}_i \leftarrow f_{NN}(\boldsymbol{x}_i; \boldsymbol{W}, \boldsymbol{b})$      ▷ cf. Section 3.2
          compute loss: $C_i \leftarrow (y_i - \hat{y}_i)^2$
          apply backpropagation for gradients $\partial C_i / \partial \boldsymbol{W}$ and $\partial C_i / \partial \boldsymbol{b}$      ▷ cf. Section 3.3
      end for
      compute full-batch cost function: $C \leftarrow \frac{1}{m} \sum_{i=1}^{m} C_i$
      compute full-batch gradients w.r.t. $\boldsymbol{W}$: $\frac{\partial C}{\partial \boldsymbol{W}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C_i}{\partial \boldsymbol{W}}$
      compute full-batch gradients w.r.t. $\boldsymbol{b}$: $\frac{\partial C}{\partial \boldsymbol{b}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C_i}{\partial \boldsymbol{b}}$
      update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial C}{\partial \boldsymbol{W}}$
      update biases: $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \frac{\partial C}{\partial \boldsymbol{b}}$
  end for
```

Algorithm 2 Training a neural network with stochastic gradient descent.

```
Require: training data $\boldsymbol{X}$, targets $\boldsymbol{y}$
  define network architecture (input layer, hidden layers, output layer, activation function)
  set learning rate $\alpha$
  initialize weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$
  for all epochs do
      for example $i \leftarrow 1$ to $m$ do
          apply forward propagation $\hat{y}_i \leftarrow f_{NN}(\boldsymbol{x}_i; \boldsymbol{W}, \boldsymbol{b})$      ▷ cf. Section 3.2
          compute loss: $C_i \leftarrow (y_i - \hat{y}_i)^2$
          apply backpropagation for gradients $\partial C_i / \partial \boldsymbol{W}$ and $\partial C_i / \partial \boldsymbol{b}$      ▷ cf. Section 3.3
          update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial C_i}{\partial \boldsymbol{W}}$
          updates biases: $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \frac{\partial C_i}{\partial \boldsymbol{b}}$
      end for
  end for
```

Algorithm 3 Training a neural network with mini-batch gradient descent.

```
Require: training data $\boldsymbol{X}$, targets $\boldsymbol{y}$
  define network architecture (input layer, hidden layers, output layer, activation function)
  set learning rate $\alpha$
  initialize weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$
  for all epochs do
      shuffle rows of $\boldsymbol{X}$ and $\boldsymbol{y}$ synchronously (optional)
      divide $\boldsymbol{X}$ and $\boldsymbol{y}$ into $n$ batches of size $k$
      for all batches do
          for example $i \leftarrow 1$ to $k$ do
              apply forward propagation: $\hat{y}_i \leftarrow f_{NN}(\boldsymbol{x}_i; \boldsymbol{W}, \boldsymbol{b})$          ▷ cf. Section 3.2
              compute loss: $C_i \leftarrow (y_i - \hat{y}_i)^2$
              apply backpropagation for gradients $\partial C_i/\partial \boldsymbol{W}$ and $\partial C_i/\partial \boldsymbol{b}$      ▷ cf. Section 3.3
          end for
          compute mini-batch cost function: $C \leftarrow \frac{1}{k}\sum_{i=1}^{k} C_i$
          compute mini-batch gradient w.r.t. $\boldsymbol{W}$: $\frac{\partial C}{\partial \boldsymbol{W}} \leftarrow \frac{1}{k}\sum_{i=1}^{k} \frac{\partial C_i}{\partial \boldsymbol{W}}$
          compute mini-batch gradients w.r.t. $\boldsymbol{b}$ : $\frac{\partial C}{\partial \boldsymbol{b}} \leftarrow \frac{1}{k}\sum_{i=1}^{k} \frac{\partial C_i}{\partial \boldsymbol{b}}$
          update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha\frac{\partial C}{\partial \boldsymbol{W}}$
          update biases: $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha\frac{\partial C}{\partial \boldsymbol{b}}$
      end for
  end for
```

3.6 Regularization of Neural Networks

As stated in the previous section on regularization (cf. Section 2.7), the task of making an algorithm perform well on new inputs and not only on the training data is one of biggest challenges in machine learning and is a field of extensive research. Regularization includes all strategies aiming to diminish the test error without increasing the training error. Ideally, they trade a significant reduction of variance for a slightly increased bias. There exist multiple approaches to regularize a machine learning model or in particular neural networks. One option is to formulate certain constraints, for example, by directly restricting the parameter values or by adding an extra term to the objective function that constrains the parameters indirectly. Some constraints and restrictions can alter an undetermined problem into a determined one, others prefer simpler models for better generalization properties. The idea to choose the simplest hypothesis among competing explanations stems back to the 14th century and is known as Occam's razor. Overall, these constraints and restrictions are often designed to encode prior knowledge about the problem and, if chosen properly, can help to reduce the generalization error. [GBC16] Furthermore, Section 2.7 introduced the three cases of underfitting (or high bias), overfitting (or high variance), and an ideal model capacity that matches the complexity of the underlying problem.

Most problems tackled by deep learning algorithms like image recognition or audio sequences are too complex to be modeled precisely. However, the practice has shown that building a large model with an appropriate regularization mechanism yields the best results in terms of minimizing the generalization error [GBC16].

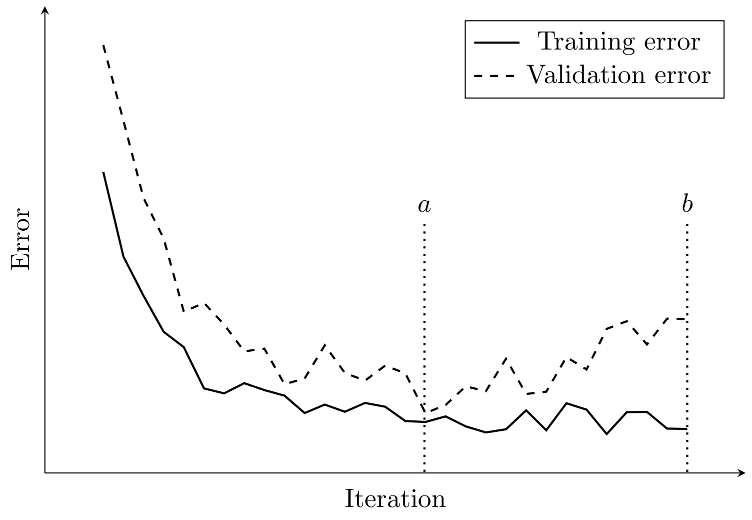


Figure 3.6: Early stopping: At point a the validation error reaches its minimum. At point b the optimization routine is terminated, since the validation error has not shown any improvement for the prescribed number of iterations. Illustration inspired by Demuth et al. [DBDJH14].

3.6.1 Early Stopping

When plotting the training and validation error for every iteration step, a common observation is that while the training error is steadily decreasing, the test error starts to rise again at certain point in time (cf. Fig. 3.6). This usually happens when the model has a sufficient capacity to overfit the problem. The basic idea of early stopping is to halt the training process as soon as the validation error rises and the model enters the overfitting regime (see point a in Fig. 3.6). In this way, the fitting of particular features of the training samples can be avoided. It is essential to monitor the error on a validation set and not the test set, because the number of training steps of the gradient-based algorithm has now become a hyperparameter of the model. Every time the validation error decreases, the weights and biases of the model are stored. The algorithm terminates when the validation error has not improved over a predefined number of iterations steps (see point b in Fig. 3.6). Then, the parameters at the point of the lowest validation error are returned.

Due to its simplicity and effectiveness, early stopping is a very popular regularization method often applied in practice. It can be seen as a hyperparameter selection algorithm determining the ideal number of training steps. The only additional costs are the evaluations of the validation set after each epoch and the memory used to store the parameters. Conversely, the computational costs are often reduced significantly, since the execution of unnecessary training steps is prevented.

3.6.2 L^1 and L^2 Regularization

The two main representatives from the family of parameter norm penalties have already been introduced in Section 2.8, namely L^1 and L^2 regularization. The methods have briefly been described for linear regression, and it is straightforward to extend them to regularize neural networks. Both regularization techniques aim to limit the capacity of a model by penalizing the parameters $\boldsymbol{\Theta}$ of the model with the help of a penalty term $\Omega(\boldsymbol{\Theta})$ that is added to the cost function C [GBC16]. In a general form this can be expressed as

$$\tilde{C} = C + \lambda\Omega, \tag{3.24}$$

where $\tilde{C}$ is the regularized and C the unregularized cost function. The coefficient λ is a hyperparameter weighting the relative contribution of the penalty term Ω. If $\lambda = 0$, no regularization is applied and larger values for λ result in a more regularized model. In case of L^1 regularization, the cost function takes the form

$$\tilde{C} = C + \lambda||\boldsymbol{w}||_1, \tag{3.25}$$

and

$$\tilde{C} = C + \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w} \tag{3.26}$$

for L^2 regularization, respectively, where C can be any cost function e.g. the mean squared error.

How these terms influence the training process, can be shown when deriving the update rules of the gradient descent algorithm for the regularized cost function $\tilde{C}$ [GBC16]. Beginning with computing the partial derivatives of Eq. (3.25)

$$\frac{\partial\tilde{C}}{\partial\boldsymbol{w}} = \frac{\partial C}{\partial\boldsymbol{w}} + \lambda\,\mathrm{sign}(\boldsymbol{w}), \tag{3.27}$$

where $\mathrm{sign}(\boldsymbol{w})$ is applied element-wise and Eq. (3.26)

$$\frac{\partial\tilde{C}}{\partial\boldsymbol{w}} = \frac{\partial C}{\partial\boldsymbol{w}} + \lambda\boldsymbol{w}, \tag{3.28}$$

the learning rule for the weights takes on the updated form

$$\boldsymbol{w} \to \boldsymbol{w}' = \boldsymbol{w} - \alpha\lambda\,\mathrm{sign}(\boldsymbol{w}) - \alpha\frac{\partial C}{\partial\boldsymbol{w}}, \tag{3.29}$$

for L^1 regularization and

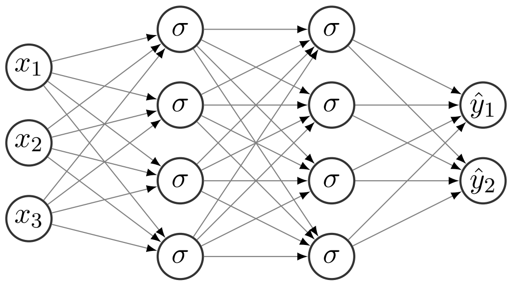
$$\boldsymbol{w} \to \boldsymbol{w}' = \boldsymbol{w}\,(1 - \alpha\lambda) - \alpha\frac{\partial C}{\partial\boldsymbol{w}}, \tag{3.30}$$

for L^2 regularization, respectively. In comparison, the update rule for an unregularized cost function is defined as

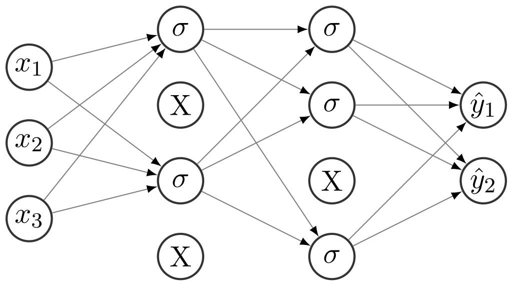
$$\boldsymbol{w} \to \boldsymbol{w}' = \boldsymbol{w} - \alpha\frac{\partial C}{\partial\boldsymbol{w}}. \tag{3.31}$$

Depending on the choice of the parameter norm different solutions are preferred [Nie15]. L1 regularization encourages the selection of few high-importance connections, while the other weights are forced toward zero. Looking at Eq. (3.29), the weights always shrink by a constant amount and eventually tend toward zero. In contrast, L2 regularization shrinks the weights proportional to w, so for small weights the reduction is much smaller compared to L1 regularization and thus, is rather seen as weight decay. One intuitive explanation why smaller weights are preferred is that they reduce the sensitivity toward changes in the inputs. If the weights are large, even a small variation of the input can drastically alter the output.

The given formulas only consider the weights, because regularizing the biases can have un-



(a) Standard neural network



(b) Neural network with dropout

Figure 3.7: Dropout. Illustration inspired by Nielsen [Nie15].

wanted effects like leading to severe underfitting. Theoretically, the hyperparameter λ could be chosen individually for each layer, but usually it is set equally for the whole network to reduce the search space during hyperparameter tuning [GBC16].

Overall, regularization with parameter norm penalties can effectively reduce the generalization error of neural networks and can therefore improve their performance on a multitude of tasks [Nie15]. Nevertheless, the effectiveness is usually demonstrated empirically, rather than with a mathematical proof.

3.6.3 Dropout

In contrast to the parameter norm penalties, dropout regularization alters the network itself instead of modifying the cost function [Nie15]. Before initializing the usual training workflow, half of the hidden neurons get randomly and only temporarily dropped, while input and output neurons stay untouched. Then, forward- and back-propagation as well as the parameter update are executed. Importantly, dropout works with stochastic gradient-based methods in which the cycle is only applied to one mini-batch of the training data (see Section 2.8). Afterwards, the dropout neurons are restored and the procedure is repeated for the next mini-batch deactivating another random subset of hidden neurons (cf. Fig. 3.7). Thus, the weights and biases are learned with only half of the hidden neurons activated. Conversely double the amount of hidden neurons is active, when the network is eventually used for making predictions. To compensate for that, the outgoing weights of every hidden neuron are halved. Other quotas than dropping 50% of hidden neurons are also possible and are usually defined with the hyperparameter $p \in [0, 1]$.

In order to understand why dropout helps the model to generalize better, it is worth to look at a related topic known as ensemble method [Nie15]. If several different neural networks are trained with the same training data, they most probably produce different results due to their varying initial states. To select the preferred output, a voting or averaging scheme is applied to the results. Assuming all networks overfit the data in a different way, taking an average over the outputs can help to prevent this type of overfitting. Since training multiple networks is only possible under extensive computational effort, dropout imitates this approach at a much reduced cost. Dropping a set of neurons during each mini-batch update is similar to training different neural networks. The dropout method can be considered as averaging the effects of a large number of various neural networks and thus may reduce overfitting.

3.6.4 Dataset Augmentation

The best way to increase the generalization abilities of a model is to train it on more data. In most cases, obtaining more training data is not feasible, but sometimes the creation of "fake" data can be an option. A good example is the task of image classification, where a high dimensional input is mapped to a single classifier. This implies that the model has to be invariant to a wide range of transformations. For the example of image recognition, it is fairly easy to introduce small variations like translation, rotation, or scaling to the input images.

Another data augmentation technique is the inclusion of noise [GBC16]. Most classification and regression tasks should still be solvable, even when a small amount of random noise is added to the input data. However, neural networks seem not to be very robust to noisy inputs. A possibility to increase the robustness is to actually train the network with noise injected data.

3.7 Example: Approximating the Sine Function

As stated in Section 3.1, fully-connected feed-forward neural networks with at least one hidden layer are capable of approximating any continuous function with arbitrary precision given a sufficient number of hidden neurons. On the example of the sine function, this section investigates some of the introduced characteristics of machine learning and neural networks. The sine is a suitable choice, since it is continuous function and its periodicity offers enough complexity to serve as an academic example. All following results are generated using Python and Tensorflow, the latter being a widely used and highly optimized library for training deep neural networks [AAB+15].

The basic architecture of the fully-connected feed-forward network is depicted in Fig. 3.8. It consists of one input unit, two hidden layers with 100 hidden neurons each as well as one output unit. The goal is to approximate

$$f(x) = \sin(2\pi x), x \in [-1, 1]$$

in the interval $[-1, 1]$. The neural network approximation can be written as

$$f_{\text{NN}} = \boldsymbol{w}_3^T \boldsymbol{\sigma} \left(\boldsymbol{W}_2 \left(\boldsymbol{\sigma}(\boldsymbol{w}_1^T \boldsymbol{x} + \boldsymbol{b}_1) \right) + \boldsymbol{b}_2 \right) + b_3 = \hat{y}, \tag{3.32}$$

with the sigmoid $\sigma(z_j) = \frac{1}{1+e^{-z_j}}$ serving as the activation function for the hidden neurons.

The training set containing 40 samples are generated by computing $y = \sin(2\pi x) + \epsilon$ for a uniform random distribution of values x across the interval $[-1, 1]$. By adding noise with the term ϵ, noisy measurements often occurring in real-world applications are simulated. The noise term is computed as

$$\epsilon = 0.1 \cdot \mathcal{U}(-1, 1),$$

where $\mathcal{U}(-1, 1)$ denotes values drawn from a uniform random distribution. Similarly, a validation set is generated by sampling 40 randomly chosen points in the interval $[-1, 1]$. The test set is simply represented by the analytical solution of the sine function.

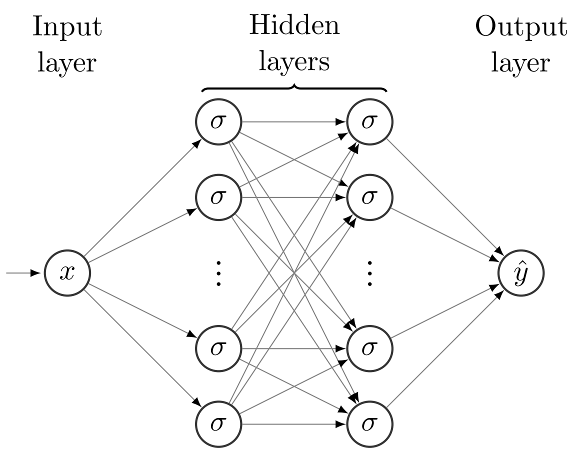


Figure 3.8: Feed-forward neural network architecture used for approximating the sine function. The network takes x as an input and outputs the corresponding $\hat{y}$ coordinate. Each of the two hidden layers consists of 100 neurons with sigmoid activation functions. The output neuron uses a linear activation.

Since the prediction of the sine function is a regression task, the mean squared error loss is chosen as the cost function C

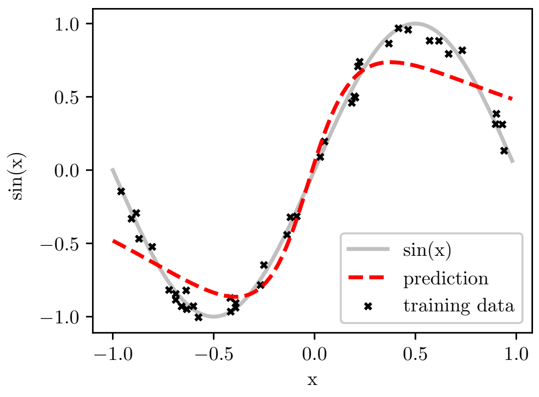
$$C = \text{MSE}_{\text{train}} = \frac{1}{m^{(\text{train})}} \sum_{i=1}^{m^{(\text{train})}} (y_i^{(\text{train})} - \hat{y}_i^{(\text{train})})^2. \tag{3.33}$$

Using the cost function C and the respective gradient ∇C, the loss on the training data is minimized by applying full-batch gradient descent, in particular the commonly used optimizer Adam (short for: Adaptive Moment Estimation), with a learning rate of $\alpha = 0.01$. The computation of the gradient is handled by Tensorflow, which provides a very fast automatic differentiation algorithm to compute the partial derivatives with respect to all parameters of the network.

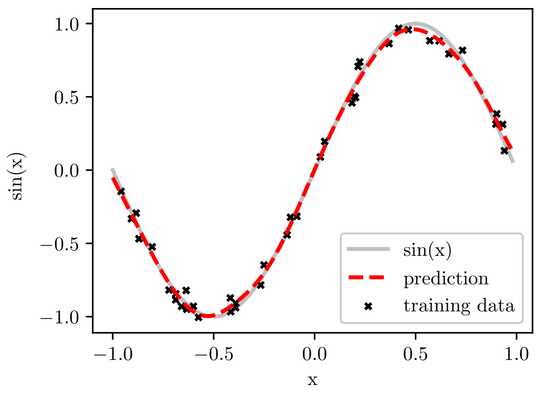
The weights of the network are initialized using a Glorot uniform initialization, as it often shows better results and faster training with gradient descent methods [GB10]. Glorot uniform initialization draws samples from a uniform distribution within certain bounds. The bounds $[-l, l]$ are defined as $l = \sqrt{\frac{6}{n_{in}+n_{out}}}$, with n_{in} being the number of input units in the weight tensor $\boldsymbol{w}^{(l)}$ and n_{out} the number of output units, respectively. The choice of initial weights can have an impact on the results, since the gradient based algorithm may descent into a completely different minimum of the cost function (cf. Section 2.8). For the biases, the initial values are simply set to zero.

First results at different epochs of the training process are shown in Fig. 3.9. While the number of training iterations is still low (cf. Fig. 3.9a), the model struggles to fit the training data and the underlying sine function. As soon as the network has learned the parameters for a sufficient amount of time (cf.Fig. 3.9b), it fits the training data and also generalizes well on the test data across the sine. If the training continues for too long (cf. Fig. 3.10a), the model starts to fit every example in the training set. This includes also the noisy data points resulting in overfitting and bad predictions for the test data.

Early stopping, a straight-forward approach to prevent the model from overfitting has already

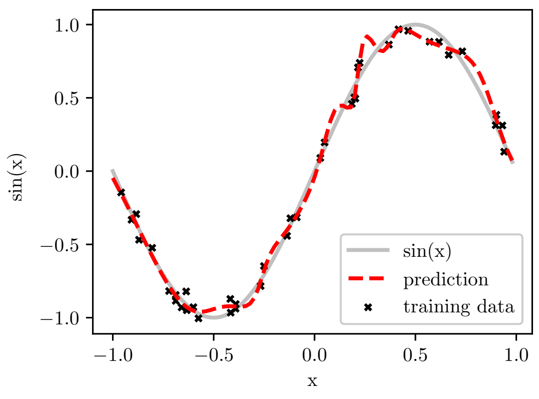


(a) Neural network still showing signs of underfitting after 500 training epochs.

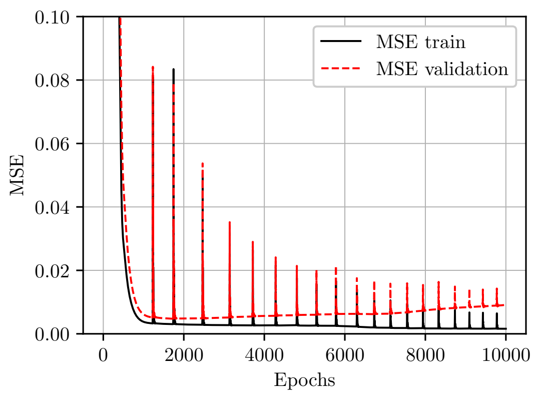


(b) Neural network properly fitting the underlying sine function after 2000 iterations.

Figure 3.9: Predictions of the example network at different training epochs.

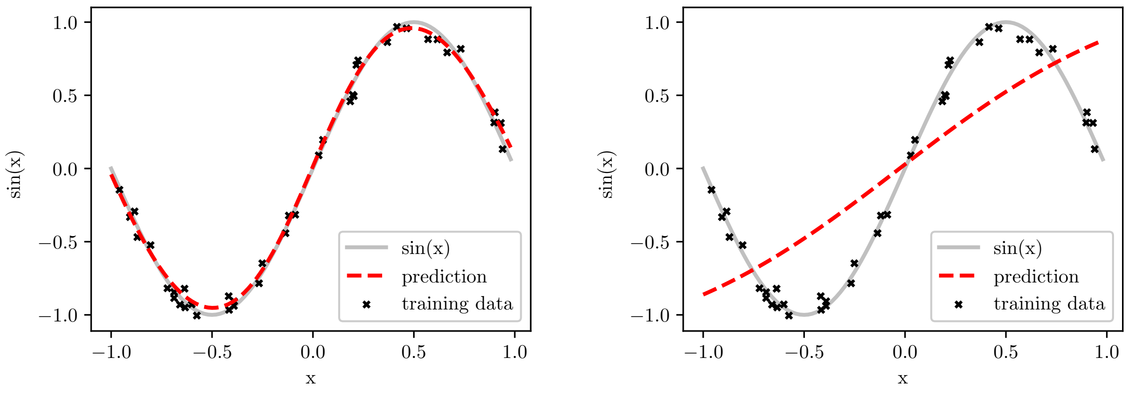


(a) Neural network prediction after 10 000 epochs of training and no regularization. Clearly showing signs of overfitting.



(b) Training and validation error plotted for each training iteration. The spikes are caused by the Adam optimizer adapting the learning rate.

Figure 3.10: The example network trying to predict the sine function after 10 000 training iterations.



(a) Prediction after 10 000 epochs of training and L^2 regularization ($\lambda = 0.0001$).

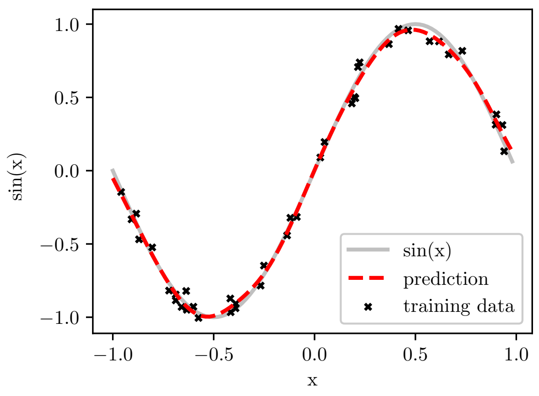
(b) Prediction after 10 000 epochs of training and L^2 regularization ($\lambda = 0.01$).

Figure 3.11: Regularization of sine example.

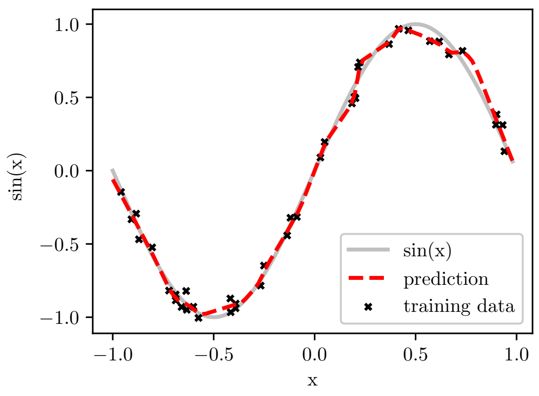
been introduced in the previous Section 3.6.1. When the validation error can not be improved within a predefined number of epochs, the early stopping algorithm simply returns the model parameters at the point of the lowest validation error. For the example network, Fig. 3.10b shows the training and validation errors plotted against the number of epochs. After reaching epoch 2000, the validation error begins to rise again. In the given example, this patience parameter for early stopping is set to 1000 epochs. The training stops after 3243 iterations, meaning the lowest validation error was reached at epoch 2243.

As discussed in Section 3.6.2, another way to resolve the problem of overfitting is to use parameter norm penalties like L2 regularization, which adds a term to the cost function penalizing large weights. The influence of the penalty term is controlled by the hyperparameter λ. The plots in Fig. 3.11 show the results with applied L2 regularization for different values of λ. For large values of λ the model exhibits underfitting, meaning it fails to fit the training data as well as the test data (cf. Fig. 3.11b). In contrast, when the ideal value for λ is found, the network is able to fit the data points of the training set and also a generalization to the test data is achieved (cf. Fig. 3.11a). If no regularization ($\lambda = 0$) is applied, the model shows the typical typical of overfitting (cf. Fig. 3.10a). So, the function exactly passes through almost every noisy data point in the training data, but fails to represent the underlying sine wave.

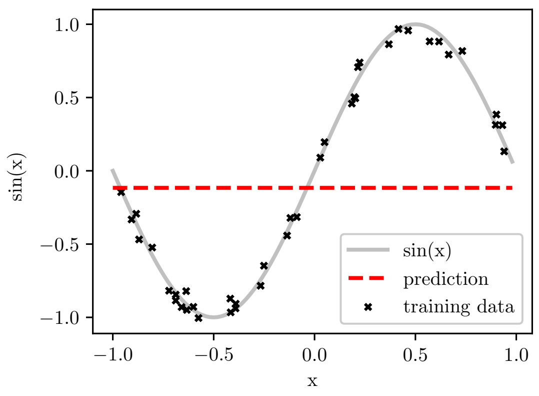
Investigating the seemingly simple example of approximating the sine function reveals that the results are sensitive to a great variety of factors. All the different kind of hyperparameters and possible settings determined by the user have a great influence on the output and the learning speed of the model. For example, choosing a different activation function, a different learning rate or even another optimizer can drastically change the outcome. Since conducting a detailed parameter study exceeds the scope of this work, Fig. 3.12 displays a few representative results that demonstrate the influence of the three aforementioned settings and hyperparameters.



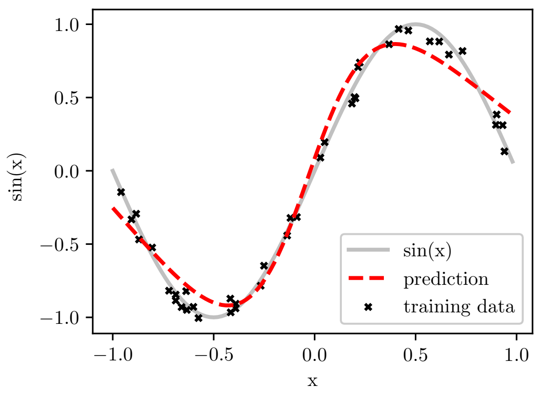
(a) Reference neural network with sigmoid activations properly fitting the underlying sine function after 2000 iterations. The network parameters are learned by applying the Adam optimizer with a learning rate of $\alpha = 0.01$.



(b) Prediction generated using ReLU as the activation function. With ReLU activations the network already starts to overfit at 2000 epochs.



(c) Prediction after 2000 epochs with learning rate $\alpha = 0.2$. This example shows that a large learning rate can prevent the network from learning.



(d) Prediction after 2000 epochs using stochastic gradient descent (SGD) instead of the Adam optimizer. A drawback of SGD is the slower convergence rate, since the parameters are updated after the evaluation of each single example (cf. Section 3.5).

Figure 3.12: Parameter study for the example network predicting the sine function. Except for the one parameter in discussion, all parameters are kept the same as in the reference example on the top left.

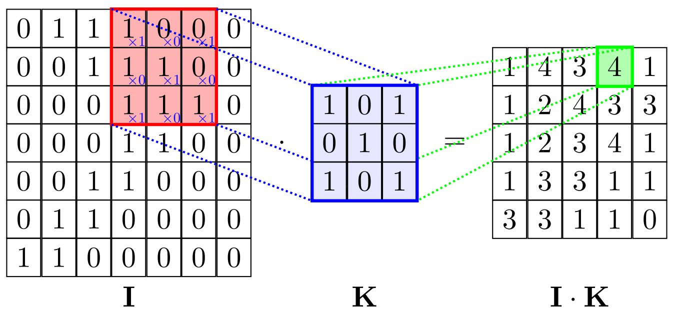


Figure 3.13: Convolution layer: The local receptive field or filter **K** is applied to the input layer **I**. Image by Veličković [Vel20].

3.8 Advanced Architectures

The following sections introduce two network architectures that have been very successful in their domains of use. Adapted to a particular data structure, they achieve superior results in various tasks when compared to fully-connected feed-forward neural networks.

3.8.1 Convolutional Neural Network

Convolutional neural networks (CNNs) are specifically designed for image processing [LBBH98]. Their distinct architecture allows them to account for the spatial structure of a picture and thus makes convolutional networks the preferred choice for image classification tasks [Nie15]. When trying to recognize an object in an image, it can be assumed that features describing the object are found in close proximity to each other. Furthermore, the exact location in the image is not of importance for the identification of an object, just the relative position of features to each other matters. CNNs exploit these two realities about physical objects, namely locality and translational variance. Like an ordinary neural network, a CNN consists of neurons that have learnable weights and biases. What differs is the arrangement of the processed data and the mathematical operation employed in the hidden layers, which are separated into convolutional and pooling operations. The input layer is not longer a vector, but has the shape of a matrix to account for the grid-like structure of pixels in an image. For instance, a convolutional network is processing a square image of 7×7 pixels where each pixel corresponds to a gray-scale value. Then, a hidden neuron in the convolutional layer is connected with a small region of the input pixels, e.g a 3×3 window, called the local receptive field of the hidden neuron (see Fig. 3.13). The size of the local receptive field and the distance between the fields, also referred to as stride length, define the number of neurons of the convolutional layer. In the example shown in Fig. 3.13, the local receptive field of size 3×3 moves over the input layer with a stride length of 1. Since the input layer consists of 7×7 pixels, the corresponding convolutional layer requires 5×5 neurons. Together with the size of the local receptive field, the stride length is a hyperparameter of the convolutional network. A specialty of CNNs is that all hidden neurons in the convolutional layer share the same weights and bias, often referred to as the filter **K**. As a result, all neurons detect the same feature at different locations in the input image, which implies the characteristic of translational invariance. The output of the convolution operation is called feature map. A convolutional layer normally consists of more than one feature map in order to detect

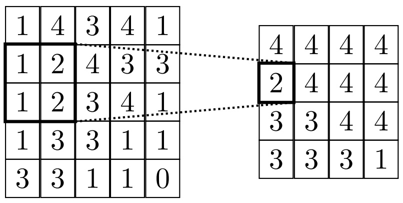


Figure 3.14: Max pooling with a 2×2 filter and stride length 1. Illustration inspired by Burkov [Bur19] and Veličković [Vel20].

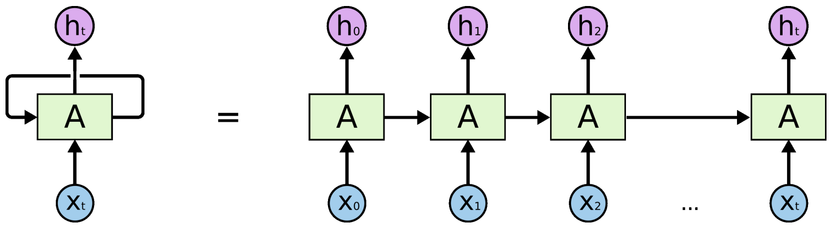


Figure 3.15: An unrolled recurrent neural network [Ola15].

multiple localized features. Every feature map is generally followed by the application of a pooling operation that condenses the output of the convolution (see Fig. 3.14). For example, each unit in the pooling layer summarizes a region of 2×2 neurons in the previous layer. A common pooling technique is max-pooling, where simply the maximum value of the respective region is taken. Pooling can be interpreted as a query that checks if a feature has been detected somewhere neglecting the exact position in the image. At the end, a fully-connected layer is added to perform the classification task. The arrangement of multiple convolutional and pooling layers allows very deep and expressive network architectures. Sharing weights and biases in combination with the pooling operation reduces the amount of trainable parameters significantly. Thus, convolutional networks learn much faster than fully-connected neural networks with comparable expressive power [Nie15]. The backpropagatoin algorithm for calculating the gradients needs only small changes to adapt to the convolution and pooling operations in the network .

3.8.2 Recurrent Neural Network

Conventional neural networks are only able to evaluate a current state described by the fixed-size input vector. So information from previous states can not persist nor be passed on to later ones. Recurrent neural networks (RNNs) tackle this issue by adding loops that allow the persistence and propagation of information over time. A recurrent neural network can be seen as multiple copies of the same network, each forwarding a message to their respective successor. The sketch of an unrolled recurrent neural network is depicted in Fig. 3.15 [Ola15]. As shown, RNNs are chain-like structures that take in sequences of data $\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$ and generate a sequential output $\hat{\boldsymbol{y}}_0, \hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_t$. Sequential data often occurs in tasks like speech recognition, language modeling and translation, making recurrent neural networks the preferred choice for these kind of problems. One shortcoming of standard RNNs is that they are only able to connect very recent information with the current task. An illustrative example is the prediction of words in text sequences. For instance, the final word in the sentence "the color of a lemon is *yellow*" can directly be derived from the preceding words. Often the relevant information needed for the prediction is found much earlier in the sequence. In the text excerpt "We used too many lemons for our lemonade... We did not like the lemonade, it was too *sour*", the gap between the information and the place of prediction is much larger and RNNs usually fail to learn such long-term dependencies [Hoc91]. This problem has been solved by introducing a special implementation of recurrent neural networks, called long short-term memory networks (LSTMs) that often achieve outstanding results in the aforementioned tasks. The key concept of LSTMs is their cell state, where information has to pass several gates before it gets passed on. Three different gates control and protect the cell state by deciding if information is added, kept or discarded. In this way, LSTMs are capable to remember information and storing long term dependencies. Since recurrent neural networks employ a sequential structure, the backropagation algorithm is adapted to take into account the temporal component resulting in a method called "backpropagation through time" [Ola15].

For more in-depth content about recurrent neural networks, the reader is referred to chapter 10 of Goodfellow's "Deep Learning" book [GBC16, Chapter 10] and Christopher Olah's blog post "Understanding LSTMs" [Ola15].

Chapter 4

Literature Review

The access to enormous quantities of data combined with rapid advances in machine learning in recent years yielded outstanding results in the fields of computer vision, recommendation systems, medical diagnosis, or financial forecasting [AMMIL12]. Nonetheless, the impact of learning algorithms reaches far beyond and already found its way into many scientific disciplines [AJZS18]. While machine learning frameworks are already able to support radiologists in medical diagnostics [LRVL^{+}12], do scientists from other fields only begin to explore the immense potential of data-driven algorithms.

The first part of this chapter provides a short overview of developments in physics and engineering with an emphasis on applications in the domain of computational mechanics. The second part is dedicated to a more detailed review of a paper by Raissi et al., that introduces the class of so-called physics-informed neural networks (PINNs) [RPK19]. Lastly, this chapter ends with an outline of several works, that build up on the findings of the aforementioned paper.

4.1 Machine Learning in Physics and Engineering

Even before the unprecedented success of deep learning, a handful of academics identified the potential of neural networks in scientific computations during the 1990s and early 2000s. For instance, Rico-Martínez and Kevrekidis used a neural-network-based approach for the identification of non-linear systems, that are continuous in time [RMK93]. With the help of a neural network, Milano and Koumoutsakos demonstrated the reconstruction of a near-wall field in turbulent flow [MK02]. An example from the field of chemical engineering is the process modeling of a fed-batch bioreactor. Psichogios and Ungar described a hybrid network architecture that incorporates additional information about the problem [PU92]. The idea of enriching a neural network architecture with prior knowledge is also found in the work of Lagaris et al., who propose an artificial neural network for solving ordinary and partial differential equations. The last two examples inspired the paper by Raissi et al. [RPK19], which will be discussed in more detail in Section 4.2.

Solving scientific problems with the help of machine learning techniques is a very broad and active area of research. The following section gives a short introduction to the topic and it provides the reader with an overview of current literature.

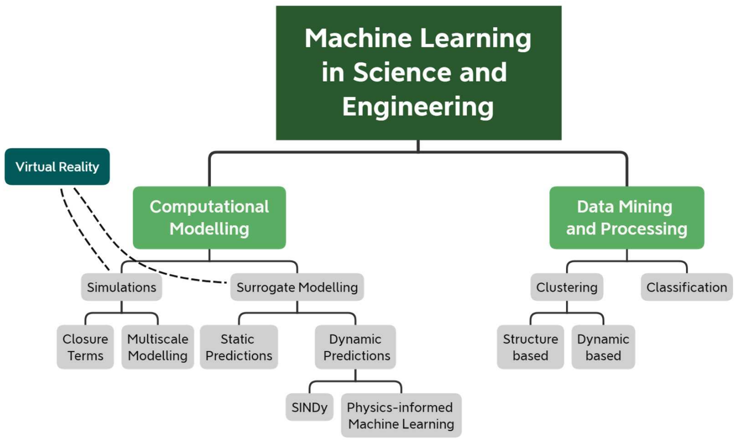


Figure 4.1: Machine Learning in Science and Engineering. The image was taken from Frank et al. [FDC20].

4.1.1 General Reviews

In order to validate existing theories or new hypothesis, scientists heavily rely on empirical studies. This means a great part of scientific work is dedicated to data analysis. For instance, the sub-field of statistical physics deals with the derivation of physical models from experimental data. According to Mehta et al., statistical physics shares a lot of common ground with statistical learning theory, the fundamental concept of learning probability distributions from data [MBW+19]. Their recommendable introduction to the world of algorithmic data analysis is addressed to interested readers with a background in physical sciences. Carleo et al. take on a similar perspective [CCC+19]. They discuss the interface between machine learning and physics and further highlight how both fields could equally benefit from each other. According to the authors, one example of this potential symbiosis is quantum computing. Machine learning can support the building process and analysis of quantum computers. On the other hand, the execution of learning algorithms on quantum computers could lead to a significant speed-up of the training process.

A review article about the applications of machine learning in natural sciences by Frank et al. puts a greater emphasis on recent developments within computational sciences and engineering [FDC20]. Next to scientific data analysis supported by well-established learning algorithms, they discuss machine learning in the context of computational simulation and modeling. More specifically, they distinguish between algorithms that help to improve conventional computational methods and different classes of surrogate models that can replace those methods completely in specific situations (cf. Fig. 4.1).

Conventional methods in the context of computational mechanics usually refer to the finite element method (FEM) or other discretization methods such as the finite difference or finite volume method. The core idea of discretization methods is to subdivide a large domain into simpler and smaller parts, that can be easily processed by a computer. These methods allow the numerical solution of partial differential equations and they are applied to a variety of static and dynamic problems. Due to its robustness, the finite element method is the most popular approach for solving problems in the fields of structural analysis, heat transfer, multiphysics applications as well as fluid mechanics [All07]. The application of finite differences and finite volumes is more prevalent in problems of thermodynamics and computational fluid dynamics [PTA12].

Time-dependent partial differential equations are of major significance in the field of fluid dynamics. Fluids in motion often exhibit highly non-linear behavior, which makes their solution with conventional numerical methods very costly. In the opinion of Brunton et al. fluid mechanics could benefit from the application of machine learning due to its ability to cope with non-linear relations [BNK20]. Their review dealt with recent applications in problems of flow modeling and optimization as well as experimental flow control. While highlighting the success of machine learning in critical tasks like model order reduction or feature extraction, they pointed out current obstacles that demand future research.

Inspired by recent results in reinforcement learning (cf. Section 2.3.4) Garnier et al. assessed first approaches that apply deep reinforcement learning in the context of fluid dynamics [GVR$^+$19]. On the examples of flow control and shape optimization, new ideas are compared to classical methods.

Computationally challenging tasks in mechanics do not only arise in structural or fluid dynamics. Also the domain of material mechanics could possibly benefit from the introduction of machine learning techniques. A collection of articles published by Huber et al. present data-driven approaches contributing to the advancement of continuum material mechanics [HKKC20].

The cited reviews agree on the great potential of machine learning in applications of physical sciences and engineering. No less could machine learning profit from a wider adoption in the scientific community. For instance, the interface between fluid mechanics and machine learning lets hope for a fruitful exchange of ideas between the two fields [BNK20]. However, some authors noted that machine learning introduces new uncertainties and drawbacks. Data-driven algorithms often lack robustness and can not guarantee convergence [GVR$^+$19]. Further, the interpretability and explainability of results are compromised, when algorithms are simply applied as “black-boxes” [BNK20].

4.1.2 Combined Methods

After a more general introduction on potential uses of machine learning in science and engineering, the focus shifts on attempts that augment existing methods for modeling and simulation of physical problems.

Solving partial differential equations while maintaining small-scale features of the solution becomes computationally infeasible for large time-scales. In these cases, equations that represent a coarse-grid approximation of the underlying problem are derived. However, it is not always possible to find such a suitable approximation function analytically. Motivated by this circumstance Ben-Sinai et al. proposed a neural network that learns an effective approximation from actual solutions of the underlying partial differential equations [BSHHB19]. This data-driven approach allows accurate results for a much coarser time discretization when compared to the standard finite-differences method. Nevertheless, the computational overhead due to the convolutional operations of the neural network is much higher. Additionally, the scalability needs further investigation, since the method was only demonstrated on examples with one spatial dimension.

The finite element method requires the element-wise calculation of integrals [All07]. A standard approach used for the numerical integration is the Gaussian quadrature, that approximates an integral with a finite sum [Ise08]. Oishi et al. proposed a method, that optimizes the quadrature rule for computation of the finite element stiffness matrix using a deep neural network [OY17]. The resulting quadrature rule was more accurate than the standard Gauss-Legendre quadrature for the same amount of integration points. Improving the performance of numerical integration is of general importance to computational mechanics and not only in the context of the finite element method. Having said that, the computational costs of training a neural network for each problem individually does not justify the gain in accuracy so far.

Material modeling is an essential part of simulating physical entities. The material models are usually derived from experimental data and are then calibrated for further calculations. According to Kirchdoerfer et al. the additional step of empirical material modeling is a non negligible source of error to the solution of complex systems [KO16]. Their work introduced a method, that replaces the empirical material modeling process with data-driven computations. The proposed solver directly utilizes experimental material data in combination with essential constraints and conservation laws. The first results were conducted on the examples of non-linear three-dimensional trusses and linear elastic solids showing good convergence properties. Even though their approach was formulated in the context of quasi-static mechanics, the authors believe that an extension to dynamic problems is possible.

4.1.3 Surrogate Models

The previously discussed examples followed the idea of improving existing numerical methods by means of data-driven algorithms. A different approach to exploit the predictive power of machine learning algorithms is to use them as surrogate models for physical simulations. The following publications introduced the first attempts of replacing costly and time-consuming computations with data-driven predictions.

Image-guided interventions are exemplary for clinical applications, that demand immediate feedback to practitioners. Due to the high complexity of biomechanical models standard numerical methods, such as the finite element method, are not suitable for the application in time-sensitive tasks. On the way to providing real-time results on patient-specific geometries Liang et al. proposed a deep learning framework that can directly estimate the stress distribution in the wall of an aorta [LLMS18]. The stress distributions used to train the deep neural network were generated by finite element analysis of 729 patient-specific geometries. Martínez-Martínez et al. provided another example from the field of medical applications [MMRMMS$^+$17]. In contrast to the preceding approach, they made use of tree-based methods to simulate the biomechanical behavior of breast tissues during image-guided interventions. Again, the training data was based on finite element simulations of ten real breast models.

In the early design phases of engineering structural components, it is crucial to run multiple iterations, e.g. for the shape optimization of an airfoil. When the problem involves fluid dynamics, the computation in each iteration is costly. Even with the use of efficient solvers, the engineer's workflow is delayed by long waiting times since the solution has to be computed for each and every change in design. The concept of Afshar et al. is to estimate the pressure field and velocity of dynamic problems using a convolutional neural network (cf. Section 3.8.1), that learns from pixelated solutions [ABP$^+$19]. In particular, they demonstrated the pre-

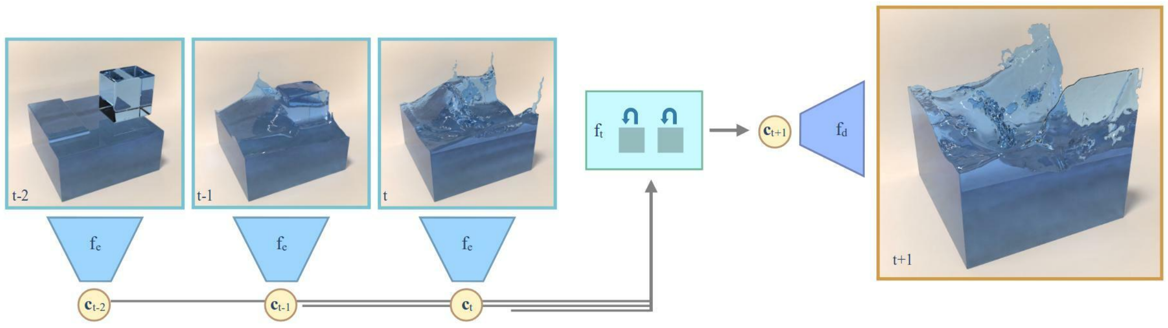


Figure 4.2: A combination of long-short term memory and convolutional neural network for fluid flow prediction. Here the trapezoidal shapes denote the encoding and decoding of spatial information by the convolutional layers and the rectangle in the middle represents the LSTM network predicting the temporal evolution in the reduced space. Image by Wiewel et al. [WBT19].

diction of a two-dimensional flow field around different airfoil geometries under variable flow conditions and compared the performance to a classical Reynolds-averaged Navier–Stokes solver. The accuracy of the results was sufficient for the early design stage while the computational time decreased by a factor of four on the available hardware. Three years earlier Guo et al. proposed a very similar idea [GLI16]. Instead of computing the solution of airflow around an obstacle with conventional methods, they trained a convolutional neural network to predict the resulting velocity field. The experiments were conducted on a greater variety of shapes compared to the first approach including simple three-dimensional geometries. In terms of speed-up, they claimed to accelerate the computations by two orders of magnitudes in comparison to classical numerical methods.

Motivated by the goal of simulating physics in real-time, progress is made in the field of computer graphics. Wiewel et al. introduced an interesting data-driven framework for the prediction of fluid flows [WBT19]. In a first step, they generated data-sets with a classical Navier-Stokes solver. Then, a convolutional neural network (CNN) was trained to learn a mapping from the three-dimensional problem into a smaller spatial representation. At the same time, the network learned the corresponding inverse mapping. The reduced model was then fed to an LSTM neural network (cf. Section 3.8.2) that predicted the temporal evolution in the reduced space. Finally, the earlier learned reverse mapping transformed the output of the LSTM network back into the three-dimensional space. The whole process is also depicted in Fig. 4.2 [WBT19]. Due to the efficient compression of the CNN, this method allows significant speed-ups compared to conventional fluid flow simulations according to the authors. Furthermore, the proposed work shows good generalization capabilities. From an engineering viewpoint, it is important to note that the accuracy of the results is mainly judged on visual comparison to the reference computation.

All presented works demonstrate the possibility of replacing physically complex simulations with data-driven computations. Using a learned surrogate model may lead to a significant speed-up, which enables the on-line application in time-sensitive tasks. As a consequence, the problem of expensive data generation and time-consuming training is shifted to preliminary computations. A major drawback of this approach is the limited generalization ability. The machine-learning frameworks are trained for one specific task and can not be used for arbitrary problems. In order to achieve better generalization properties incredibly large data-sets and deep learning architectures would be needed.

4.2 Physics-informed Neural Networks

Generating an accurate surrogate model of a complex physical system usually requires a large amount of data about the problem at hand. However, data acquisition from experiments or simulations is often infeasible or too costly. With this in mind, Raissi et al. proposed an approach, that augments surrogate models with existing knowledge about the underlying physics of a problem [RPK19]. In many cases, the governing equations or empirically determined rules defining the problem are known a priori. For instance, an incompressible flow has to satisfy the law of conservation of mass. By incorporating this information, the solution space is drastically reduced and, as a result, training less data are needed to learn the latent solution. In particular, the goal is to solve problems which can be described by parameterized nonlinear partial differential equations of the form

$$u_t + \mathcal{N}[u;\lambda] = 0,\ x \in \Omega,\ t \in [0,T]. \tag{4.1}$$

Here, the latent solution $u(t,x)$ depends on time t and a spatial variable x, $\mathcal{N}[u;\lambda]$ represents the nonlinear operator with parameter λ, and Ω refers to a space in $\mathbb{R}^D$. This description covers a wide range of problems ranging from advection-diffusion-reaction of chemical or biological systems to the governing equations of continuum mechanics.

The idea of adding prior knowledge to a machine learning algorithm is not completely new. As mentioned before, the studies by Raissi et al. were inspired by papers of Psichogios and Ungar [PU92], Lagaris et al. [LLF98], and more recent developments by Kondor [Kon18], Hirn et al. [HMP17] and Mallat [Mal16]. Nevertheless, the solutions proposed by Raissi et al. extended existing concepts and introduced fundamentally new approaches like a discrete time-stepping scheme, that efficiently exploits the predictive power of neural networks. Furthermore, they demonstrated their method on a variety of examples that are of interest in a physics and engineering context. Their code was written in Python and utilizes the popular GPU-accelerated machine learning framework Tensorflow. Additionally, the code is publicly available on GitHub allowing others to explore physics-informed neural networks and contribute to their development [Rai20].

The paper was referenced in different reviews [BNK20, FDC20] and inspired further research on physics-enriched surrogate models as outlined in Section 4.3. The wide-spread recognition by the scientific community also motivated this thesis to establish a deeper understanding of physics-informed neural networks and to investigate their possible applications.

The main article this thesis is referring to was published in 2019 in the Journal of Computational Physics [RPK19]. It can be seen as the summary of a two-part series of pre-prints already available since 2017 [RPK17d, RPK17e]. Following the structure of the papers, this section begins with an introduction to data-driven inference of partial differential equations, followed by a description of data-driven identification of partial differential equations. Each of these sections is further subdivided into descriptions of continuous and discrete-time models. If not indicated differently, the contents in this section refer to the three aforementioned papers by Raissi et al. [RPK17d, RPK17e, RPK19].

4.2.1 Data-driven Inference

The problem of inference can be phrased as: find the hidden solution $u(t, x)$ for fixed model parameters λ. Since λ is known Eq. (4.1) simplifies to

$$u_t + \mathcal{N}[u] = 0, \quad x \in \Omega, \quad t \in [0, T]. \tag{4.2}$$

Continuous-time Model

As an introductory example the authors chose the initial-boundary value problem of a one dimensional Burgers' equation. The governing partial differential equation along with the initial condition and Dirichlet boundary conditions is defined as

$$\begin{aligned} u_t + uu_x - (0.01/\pi)u_{xx} &= 0, \qquad\qquad x \in [-1, 1], \quad t \in [0, 1], \\ u(0, x) &= -\sin(\pi x), \\ u(t, -1) = u(t, 1) &= 0. \end{aligned} \tag{4.3}$$

Now, a deep neural network is used to approximate the unknown solution $u(t, x)$. So far this approach does not differ from the surrogates introduced in Section 4.1.3. However, these models have to rely on a tremendous amount of labeled training data closely related to the solution being approximated. Instead of having labeled training data in the whole domain, the solution for the example at hand is only known at initial time $t = 0$ and on the boundaries. Additionally, Eq. (4.3) has to be satisfied for every point inside the domain. In order to verify that every input fulfills this condition, the network is extended to compute the left-hand-side of equation Eq. (4.3)

$$f := u_t + uu_x - (0.01/\pi)u_{xx}. \tag{4.4}$$

Since a neural network is fully differentiable, it is not only possible to compute the derivatives with respect to the parameters necessary for training. Likewise, the automatic differentiation capabilities of libraries such as Tensorflow allow the fast computation of derivatives with respect to the input variables x and t. To demonstrate the simplicity of computing the derivatives u_t, u_x, and u_{xx} needed for $f(t, x)$, the authors show a code snippet where `u(t,x)` represents the neural network approximating $u(t, x)$:

```
def f(t,x):
u = u(t,x)
u_t = tf.gradients(u,t)[0]
u_x = tf.gradients(u,x)[0]
u_xx = tf.gradients(u_x,x)[0]
f = u_t + u*u_x - (0.01/tf.pi)*u_xx
return f
```

All together, this extended network architecture can be interpreted as a physics-informed neural network with outputs $u_{NN}(t, x)$ and $f_{NN}(t, x)$. As depicted in Fig. 4.3 the first part simply approximates the solution $u(t, x)$ with a feed-forward fully connected neural network. The second part represents the code snippet from above for the computation of $f(t, x)$ and the corresponding partial derivatives u_t, u_x, and u_{xx}. It should be noted that both networks

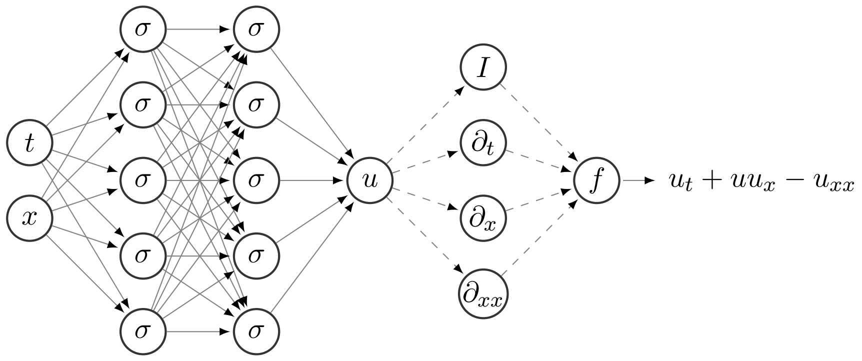


Figure 4.3: Conceptual physics-informed neural network for the Burgers' equation. The left part shows the feed-forward neural network and the right part represents the physics-informed neural network. The dashed lines denote non-trainable weights.

$u_{NN}(t,x)$ and $f_{NN}(t,x)$ depend on the same set of parameters, namely the weights and biases of the first network. The parameters of the second part are not trainable. The connections can be interpreted as constant weights with value one and the biases of the neurons remain at zero.

Like with any other machine learning algorithm, the parameters are learned by minimizing a suitable loss function. In case of the physics-informed neural network this custom loss function is assembled from three mean squared error losses

$$C = MSE_0 + MSE_b + MSE_f, \tag{4.5}$$

where

$$MSE_0 = \frac{1}{N_0}\sum_{i=1}^{N_0}\left(u_{NN}\left(0, x_0^i\right) - u_0^i\right)^2, \tag{4.6}$$

$$MSE_b = \frac{1}{N_b}\sum_{i=1}^{N_b}\left(u_{NN}\left(t_b^i, x_b^i\right) - u_b^i\right)^2, \tag{4.7}$$

and

$$MSE_f = \frac{1}{N_f}\sum_{i=1}^{N_f}\left(f_{NN}\left(t_f^i, x_f^i\right)\right)^2. \tag{4.8}$$

The known solution at the initial time $t = 0$, is represented by N_0 labeled data points $\{0, x_0^i, u_0^i\}_{i=1}^{N_0}$ and the solution on the boundary by N_b labeled data points $\{t_b, x_b^i, u_b^i\}_{i=1}^{N_b}$, respectively. The losses MSE_0 and MSE_b are simply computed by comparing the approximation u_{NN} with the labels u_0 and u_b of the training data. In order to enforce Eq. (4.2) in the whole spatio-temporal domain, a set of N_f collocation points $\{t_f, x_f^i\}_{i=1}^{N_f}$ is generated using a Latin hypercube sampling technique [Ste87]. The corresponding loss MSE_f from Eq. (4.8) is simply computed as the mean squared error of f_{NN} at all collocation points.

Now the physics-informed neural network can be trained by minimizing the cost function in Eq. (4.5). Contrary to the standard gradient descent methods used in machine learning, the authors applied a quasi-Newton, full-batch gradient descent algorithm, called L-BFGS [LN89]. In an additional step, this optimizer approximates the diagonal of the Hessian to

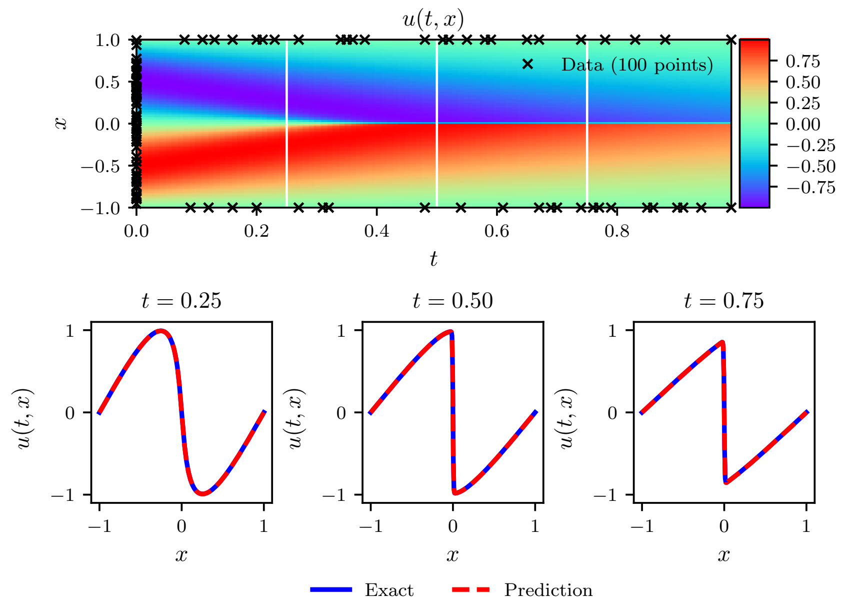


Figure 4.4: Continuous-time inference of Burgers' equation. Top: The predicted solution $u(t,x)$ for the whole spatio-temporal domain is displayed along with the location of the data points used for training. Bottom: Exact and predicted solution at the temporal snapshots indicated by white lines in the upper panel. This figure was generated with the code from Raissi et al. [Rai20].

determine a suitable step-size. For larger data-sets, it was recommended to switch to a computationally more efficient mini-batch gradient descent algorithm.

Even though the proposed approach is not guaranteed to converge to a global minimum, thus an accurate solution $u(t,x)$, the authors showed empirically, that their method achieves accurate results for different problems and architectures. Assuming that the partial differential equation has a unique solution and is well-posed, the physics-informed neural network is able to predict the underlying solution. Further requirements are a network architecture with adequate representational power and a sufficient number of collocation points N_f.

To study the robustness of their proposed method the authors conducted multiple sensitivity analyses. First, they kept the network architecture fixed and changed the number of training examples $N_0 + N_b$ as well as the number of collocation points N_f. In a second run, both training and collocation points remained fixed and the number of layers and neurons was varied. Both studies showed a trend of increased prediction accuracy when more data are available or the network's capacity is increased.

It is remarkable, that the approach was capable of accurate prediction, even for relatively small amounts of labeled training data. For instance, given $N_0 + N_b = 100$ randomly distributed training examples and $N_f = 10\,000$ collocation points, a physics-informed neural network with 8-hidden layers containing 20 neurons each could predict the solution to the one dimensional Burgers' equation from Eq. (4.3) with an error of 6.7×10^{-4} in the relative $\mathcal{L}_2$-norm (cf. Fig. 4.4). In the discussion of their results, the authors claimed, that the design of the cost function prevents the common habit of machine learning algorithms to overfit (cf. Section 2.6). The combination of multiple terms in the cost function and especially the loss MSE_f seems to have a regularizing effect. They further emphasized that the proposed method is mesh-free. Nevertheless, there are limits to this approach since the number of collocation points increases exponentially for higher-dimensional problems. To keep a comparable density in three spatial dimensions, not 100^2, but 100^4 collocation points are needed. Propagating such a vast amount of examples through the network at each iteration results in an extremely slow algorithm. In these cases, a mini-batch gradient descent algorithm might promise better performance. Apart from the Burgers' equation, the authors tested the continuous-time inference on the Schrödinger equation and the method yielded comparably accurate results on this more complex example.

Discrete-time Model

To circumvent the need for collocation points Raissi et al. proposed an alternative solution-inference approach based on a Runge-Kutta time stepping scheme. Contrary to a continuous prediction in time, the solution is only predicted at certain time steps. Assuming the solution at time step t^n is known, then the proposed method is able to predict the solution at the next time step $t^{n+1} = t^n + \Delta t$, where Δt denotes step size.

To solve the following problem

$$u_t = g[u], \tag{4.9}$$

the general form of the Runga-Kutta method with q stages is written as [Ise08]

$$\begin{aligned} u^{n+c_i} &= u^n + \Delta t \sum_{j=1}^{q} a_{ij} g\left[u^{n+c_j}\right], \quad i = 1, \dots, q, \\ u^{n+1} &= u^n + \Delta t \sum_{j=1}^{q} b_j g\left[u^{n+c_j}\right], \end{aligned} \tag{4.10}$$

where

$$u^{n+c_j}(x) = u(t^n + c_j \Delta t, x) \quad j = 1, \dots, q. \tag{4.11}$$

Writing the Runge-Kutta method in this general form allows the usage of both explicit and implicit time-stepping schemes. The advantage of explicit methods is that they are fast and easy to implement. After the solution at the first stage is obtained, it is substituted into the equation at the second stage and so on. As the name suggests, implicit methods can not simply be solved by substitution. They form a system of dependent equations, that requires the use of an iterative solution process. However, implicit methods exhibit excellent stability properties, which make them especially suitable for stiff systems. The type of method depends on the choice of parameters a_{ij}, b_j, and c_j that are organized in a so-called Butcher table [Ise08].

Assuming a neural network is able to predict the solution u^{n+1} at time t^{n+1} and the intermediate solutions u^{n+c_i} at all stages $i = 1, \dots, q$ from an input x, then its output can be written as

$$\left[u^{n+c_1}(x), \dots, u^{n+c_q}(x), u^{n+1}(x)\right]. \tag{4.12}$$

In particular, the neural network predicts the left-hand side of Eq. (4.10). Rearranging

Eq. (4.10) yields

$$\begin{aligned} u^n &= u^{n+c_i} - \Delta t \sum_{j=1}^{q} a_{ij} g\left[u^{n+c_j}\right], \quad i = 1, \ldots, q, \\ u^n &= u^{n+1} - \Delta t \sum_{j=1}^{q} b_j g\left[u^{n+c_j}\right]. \end{aligned} \tag{4.13}$$

Now all the terms dependent on the prediction of the neural network stand on the right-hand side and the solution at time t^n is found on left. In other words, Eq. (4.13) is a backward time-stepping scheme. Knowing the solution at time t^{n+1} as well as all solutions at the intermediate stages, the solution at time t^n can be predicted. To assign a unique identifier to each equation in Eq. (4.13) the following nomenclature is introduced

$$\begin{aligned} u^n &= u_i^n, \quad i = 1, \ldots, q, \\ u^n &= u_{q+1}^n, \end{aligned} \tag{4.14}$$

where

$$\begin{aligned} u_i^n &= u^{n+c_i} - \Delta t \sum_{j=1}^{q} a_{ij} g\left[u^{n+c_j}\right], \quad i = 1, \ldots, q \\ u_{q+1}^n &= u^{n+1} - \Delta t \sum_{j=1}^{q} b_j g\left[u^{n+c_j}\right]. \end{aligned} \tag{4.15}$$

According to Eq. (4.15) the output of the physics-informed neural network for an input x can be defined as

$$\left[u_1^n(x), \ldots, u_q^n(x), u_{q+1}^n(x)\right]. \tag{4.16}$$

Like in the continuous-time model, the physics informed neural network consists of two parts. The first part is a deep neural network with multiple outputs as shown in (4.12). The second part transforms the output of the first network according to Eq. (4.15) and returns the quantities in (4.16) in order to compare them with the known solution at time t^n.

The discrete-time model for data-driven inference was again demonstrated on the Burgers' equation (cf. Eq. (4.3)). So $g\left[u^{n+c_j}\right]$ takes on the following form

$$g\left[u^{n+c_j}\right] = -\mathcal{N}\left[u^{n+c_j}\right] = -u^{n+c_j} u_x^{n+c_j} + (0.01/\pi) u_{xx}^{n+c_j}, \tag{4.17}$$

where $\mathcal{N}$ denotes the non-linear operator introduced in Eq. (4.2).

Given the initial data $\left\{x^{n,i}, u^{n,i}\right\}_{i=1}^{N_n}$ at time t^n and the solution on the boundaries $x = -1$ and $x = 1$ at time t^{n+1}, the network can be trained with the following cost function

$$SSE = SSE_n + SSE_b. \tag{4.18}$$

Here, SSE_n denotes the sum of squared errors over the solutions at time t^n and is defined as

$$SSE_n = \sum_{j=1}^{q+1} \sum_{i=1}^{N_n} \left(u_j^n\left(x^{n,i}\right) - u^{n,i}\right)^2. \tag{4.19}$$

The second term SSE_b enforcing the boundary conditions at time t^{n+1} and all intermediate steps t^{n+c_i} for $i = 1, \dots, q$, is given by

$$SSE_b = \sum_{i=1}^{q} \left(\left(u^{n+c_i}(-1)\right)^2 + \left(u^{n+c_i}(1)\right)^2 \right) + \left(u^{n+1}(-1)\right)^2 + \left(u^{n+1}(1)\right)^2 . \tag{4.20}$$

Thus, the network learns to predict the solution at time t^{n+1} based on the known solution at time t^n and the boundary conditions in the time interval $[t^n, t^{n+1}]$ by minimizing Eq. (4.18). This step can be repeated to predict the solution at the following time steps $u(t^{n+2}, x)$, $u(t^{n+3}, x)$, and so on. When explicit methods are used, the step size Δt is chosen to be small in order to prevent stability issues. Implicit schemes are stable even for larger time steps, but this simultaneously leads to a costly increase in the number of required stages q. What makes the proposed method distinct from the classic Runge-Kutta time stepping-schemes, is the fact that the number of stages q can be increased without a significant increase in computational effort. Adding another stage to the model simply extends the output layer of the neural network by an extra neuron and the corresponding parameters. Overall the parameters only increase linearly with the total number of stages.

Due to the better stability properties, the authors decided to test their method using an implicit time stepping scheme. The network architecture consisted of four hidden layers with 50 neurons each. In order to exploit the computational advantage the number of stages was chosen to be $q = 500$ which allows the selection of a large time step size. Given $N_n = 250$ labeled training points at initial time $t^n = 0.1$, the goal was to predict the solution at $t^{n+1} = 0.9$. As shown in Fig. 4.5 the physics-informed neural network was able to predict the almost discontinuous solution at time $t = 0.9$ from smooth initial data at time $t = 0.1$ in a single time-step of size $\Delta t = 0.8$. The relative $\mathcal{L}_2$-error for this examples was measured at 8.2×10^{-4}. To investigate the robustness of the discrete time inference model, the authors conducted a series of small sensitivity studies. Next to different network architectures, they varied the number of Runge-Kutta stages and the time-step size Δt. As expected there was a tendency toward higher prediction accuracy for more expressive networks. Similarly, the accuracy increased when more Runge-Kutta stages q were used or the time step size Δt was decreased. The authors completed the demonstration of their method with another example of a non-linear partial differential equation, more precisely the Allen-Cahn equation.

4.2.2 Data-driven Identification

System identification from sparse data is another class of problems often encountered in physics or engineering applications. In the second part of their article, Raissi et al. tackled the problem of data-driven discovery with the help of the previously introduced physics-informed neural networks. To phrase the task of system identification in other words: find the parameters λ, that describe the observed data best.

Continuous-time Model

Like in the previous section (cf. Section 4.2.1), the authors began with the definition of the continuous-time model. To recall the kind of problem to be solved, Eq. (4.1) is written again

$$u_t + \mathcal{N}[u; \lambda] = 0, \quad x \in \Omega, \quad t \in [0, T]. \tag{4.21}$$

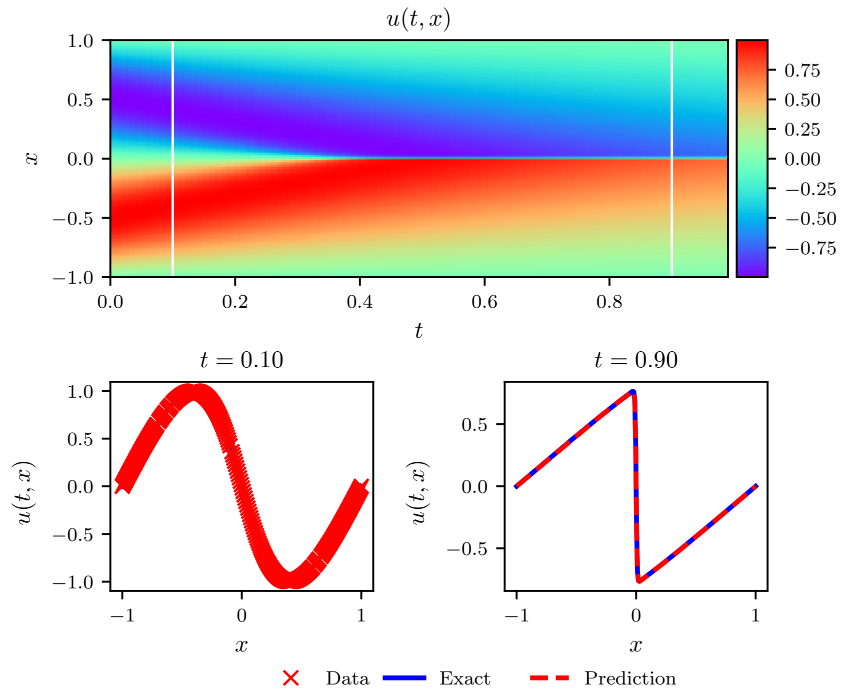


Figure 4.5: Discrete time inference of Burgers' equation. Top: The analytical solution $u(t, x)$ for the whole spatio-temporal domain is displayed along with the location of the training snapshot at $t = 0.1$ and the predicted snapshot at $t = 0.9$. Bottom: The training data and the prediction at the corresponding snapshots are shown. This figure was generated with the code from Raissi et al. [Rai20].

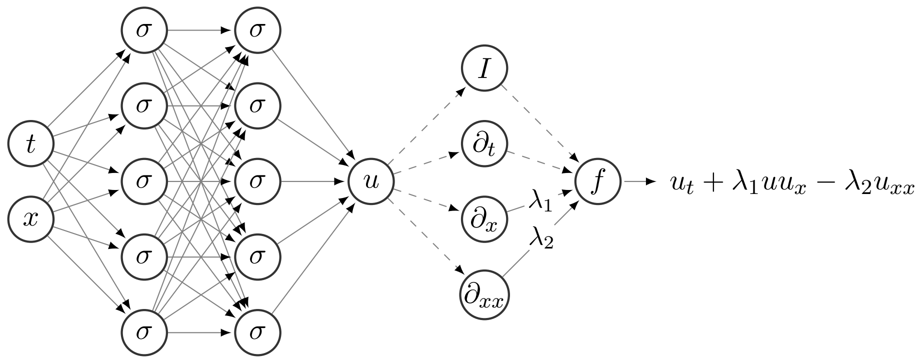


Figure 4.6: Conceptual physics-informed neural network for the identification of the Burgers' equation. The left part shows the feed-forward neural network and the right part represents the physics-informed neural network. The dashed lines denote non-trainable weights and the additional parameters λ_1 and λ_2 can be interpreted as weights of the physics-informed network part.

Since the example is already familiar, the Burgers' equation is revisited

$$\begin{aligned} &u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0, \quad x \in [-1,1], \quad t \in [0,1], \\ &u(0,x) = -\sin(\pi x), \\ &u(t,-1) = u(t,1) = 0, \end{aligned} \tag{4.22}$$

with the addition of the scalar parameters λ_1 and λ_2 for the differential operator.

Except for the additional parameters, the architecture of the physics-informed neural network as introduced in Section 4.2.1 remains unchanged (cf. Fig. 4.6). The hidden solution $u_{NN}(t,x)$ is approximated by a deep neural network. The output is then fed into the physics-informed part of the network to compute the right-hand side of Eq. (4.22)

$$f := u_t + \lambda_1 u u_x - \lambda_2 u_x x. \tag{4.23}$$

It should be noted, that λ_1 and λ_2 simply behave like trainable parameters similar to the weights and biases of the network.

In order to learn the optimal weights and biases along with the parameters λ_1 and λ_2 for the predictive network the following cost function is minimized

$$C = MSE_u + MSE_f \tag{4.24}$$

where

$$MSE_u = \frac{1}{N}\sum_{i=1}^{N}\left(u\left(t_u^i, x_u^i\right) - u^i\right)^2, \tag{4.25}$$

and

$$MSE_f = \frac{1}{N}\sum_{i=1}^{N}\left(f\left(t_u^i, x_u^i\right)\right)^2. \tag{4.26}$$

Here, MSE_u denotes the mean squared error over the training data $\{t_u^i, x_u^i, u^i\}_{i=1}^N$ and MSE_f enforces Eq. (4.22) on the same set of data points.

The authors chose again the L-BFGS optimizer to train their model on a training set of

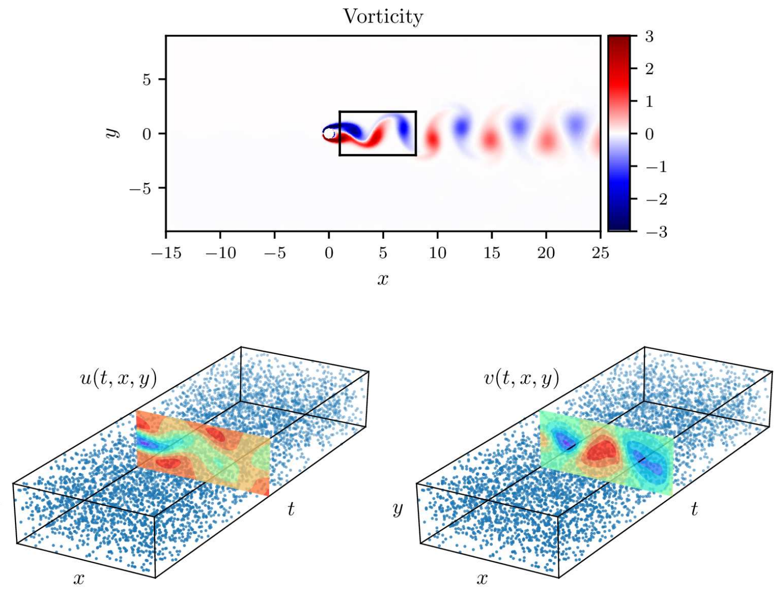


Figure 4.7: Training domain and data used for identification of the Navier-Stokes equation. Top: Incompressible flow around a cylinder with dynamic vortex shedding. Bottom: Locations of training points and representative snapshots of velocity components. Figure and text from Raissi et al. [RPK19].

$N = 2000$ randomly generated points distributed over the whole spatio-temporal domain. The samples corresponded to approximately 8% of the available data and were taken from the exact solution $u(t, x)$ where $\lambda_1 = 1.0$ and $\lambda_2 = 0.01/\pi$. A nine-layer deep neural network with 20 neurons per hidden layer was able to predict both the entire solution and the parameters λ_1, λ_2 accurately, even when the training data was altered by some uncorrelated noise. The results were validated through another systematic study investigating the influence of available training data and different noise levels. In the same way, the architecture of the network was altered to check if larger networks lead to more accurate results. All in all the results confirmed that assumption. Interestingly, the method proved to be quite robust to noise levels of up to 10%.

To engage in a more realistic example, Raissi et al. tried the proposed method on an incompressible fluid flow past a cylinder in two dimensions. The flow was modeled by the Navier-Stokes equation and a high-fidelity data set was generated with a corresponding solver. A snapshot of the regime with dynamic behavior right behind the cylinder was used for the training of the physics-informed neural network (cf. Fig. 4.7). Both the velocity and the pressure field were approximated with a deep neural network. Then, the physics-informed extension computed the Navier-Stokes equation for the two spatial dimensions. As in the previous example, the parameters of the differential operator were unknown and learned by the network. A nine-layer model, where every hidden layer accommodated 20 neurons, was trained with $N = 5000$ labeled data points corresponding to only 1% of the available snapshot data. The results showed that the physics-informed neural network was able to identify the two parameters with satisfactory accuracy even for the case when the training data were corrupted with one percent random noise.

Another remarkable observation was that the network can predict continuous quantities of interest from secondary data. In the presented example, the network could predict the pressure field even though the training data did not contain any information regarding the pressure. According to the authors, this showed the potential of physics-informed neural networks to aid the process of solving inverse problems.

Discrete Time Model

The final approach introduced by Raissi et al. is a discrete-time model for the identification of the governing partial differential equation. As described in Section 4.2.1 the aim is to solve the following problem

$$u_t = g[u; \lambda], \tag{4.27}$$

where λ represents any additional parameters. First, the general form of the Runge-Kutta method with q stages is written as

$$\begin{aligned} u^{n+c_i} &= u^n + \Delta t \sum_{j=1}^{q} a_{ij} g\left[u^{n+c_j}; \lambda\right], \quad i = 1, \dots, q, \\ u^{n+1} &= u^n + \Delta t \sum_{j=1}^{q} b_j g\left[u^{n+c_j}; \lambda\right], \end{aligned} \tag{4.28}$$

where

$$u^{n+c_j}(x) = u(t^n + c_j \Delta t, x) \quad j = 1, \dots, q, \tag{4.29}$$

The choice of parameters a_{ij}, b_j and c_j decides if an explicit or implicit time-stepping scheme is applied. Rearranging Eq. (4.28) yields

$$\begin{aligned} u^n &= u^{n+c_i} - \Delta t \sum_{j=1}^{q} a_{ij} g\left[u^{n+c_j}; \lambda\right], \quad i = 1, \dots, q, \\ u^{n+1} &= u^{n+c_i} - \Delta t \sum_{j=1}^{q} (a_{ij} - b_j) g\left[u^{n+c_j}; \lambda\right], \quad i = 1, \dots, q. \end{aligned} \tag{4.30}$$

Now, a neural network predicts the intermediate solutions for all stages q

$$\left[u^{n+c_1}(x), \dots, u^{n+c_q}(x)\right]. \tag{4.31}$$

Using the output of this network and splitting Eq. (4.30) according to the following scheme

$$\begin{aligned} u^n &= u_i^n, \quad i = 1, \dots, q \\ u^{n+1} &= u_i^{n+1}, \quad i = 1, \dots, q, \end{aligned} \tag{4.32}$$

two physics-informed neural networks are constructed, namely

$$\left[u_1^n(x), \dots, u_q^n(x), u_{q+1}^n(x)\right] \tag{4.33}$$

and

$$\left[u_1^{n+1}(x), \dots, u_q^{n+1}(x), u_{q+1}^{n+1}(x)\right]. \tag{4.34}$$

Since the three networks share their parameters by construction, the weights and biases along with the unknown parameters λ can be learned by minimizing the following sum of squared errors

$$C = SSE_n + SSE_{n+1} \tag{4.35}$$

where

$$SSE_n := \sum_{j=1}^{q} \sum_{i=1}^{N_n} \left(u_j^n\left(x^{n,i}\right) - u^{n,i}\right)^2 \tag{4.36}$$

and

$$SSE_{n+1} := \sum_{j=1}^{q} \sum_{i=1}^{N_{n+1}} \left(u_j^{n+1}\left(x^{n+1,i}\right) - u^{n+1,i}\right)^2. \tag{4.37}$$

Two distinct measurements at times t^n and t^{n+1} serve as the training data for the model. The first snapshot at time t^n is defined by the labeled data points $\left\{x^{n,i}, u^{n,i}\right\}_{i=1}^{N_n}$ and the second snapshot at time t^{n+1} consists of labeled examples $\left\{x^{n+1,i}, u^{n+1,i}\right\}_{i=1}^{N_{n+1}}$.

The example of the parameterized Burgers' equation was revisited as the basis for a systematic analysis of the proposed method

$$u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0. \tag{4.38}$$

For this specific example, the nonlinear operator with parameters λ_1 and λ_2 is defined as

$$g\left[u^{n+c_j}; \lambda\right] = -\mathcal{N}\left[u^{n+c_j}; \lambda\right] = -\lambda_1 u^{n+c_j} u_x^{n+c_j} + \lambda_2 u_{xx}^{n+c_j}. \tag{4.39}$$

Choosing a set of randomly sampled points $N_n = 199$ and $N_{n+1} = 201$ of the exact solution at the corresponding times $t^n = 0.1$ and $t^{n+1} = 0.9$ a physics informed neural network was trained to find the unknown parameters λ_1 and λ_2 by minimizing Eq. (4.35). The network architecture consisted of four hidden layers with 50 neurons each. The number of Runge-Kutta stages q for time step size $\Delta t = 0.8$ was determined by the following rule

$$q = 0.5 \log \epsilon / \log(\Delta t). \tag{4.40}$$

This empirical rule was introduced to keep the accumulated error of the time stepping scheme around the same order of magnitude as the machine precision ϵ. The method showed remarkable accuracy for varying time step size Δt, while being quite robust to random noise in the training data. Eventually, these results were validated by repeating the experiment with the Korteweg-de Vries equation [KV95].

4.2.3 Review Conclusion

Raissi et al. concluded that their findings demonstrate the potential and usefulness of the proposed methods in a great variety of applications. Whether it is the prediction of a continuous solution or the identification of hidden parameters, physics-informed neural networks are able to solve problems governed by partial differential equations in scarce data regimes. However, the authors also emphasized that the proposed approach is not aimed to replace classical methods. Instead, they rather see a coexistence of physics-informed neural networks and existing methods as demonstrated with the discrete time stepping models in Section 4.2.1 and

Section 4.2.2. Further, the authors acknowledged that their method poses many unanswered questions. Some of these questions concern the minimum requirements for the network architecture or what kind of activation functions and weight initialization techniques should be used. Another great issue that has not been tackled yet, is the quantification of uncertainty associated with neural network predictions.

4.3 Related Work

Raissi, Perdikaris, and Karniadakis, the authors of the article reviewed in the previous section, have been working on physics-enriched learning machines before [RPK17a, RPK17b]. In their earlier work, they used Gaussian processes for the inference and identification of differential equations. Certain limitations imposed by the nature of Gaussian processes [RK18, RPK17c] led to the development of neural-network-based approaches [RPK19].

The introduction of physics-informed neural networks inspired fellow researchers to adapt and extend the proposed method. For instance, Pang et al. introduced a hybrid approach, that combines numerical discretization and physics-informed neural networks to solve space-time fractional advection-diffusion equations [PLK18]. Another extension is tailored to solve three-dimensional fluid flow problems by encapsulating the governing physics of the Navier-Stokes equation [RYK18]. Following classical numerical approximations from the family of Galerkin methods, Kharazmi et al. proposed a physics-enriched network, that is trained by minimizing the variational formulation of the underlying partial differential equation [KZK19]. Choosing a variational residual as the cost function reduces the order of the differential operator and thus promises to simplify the optimization problem during training. Nevertheless, the use of a variational description does not come without limitations, since the choice of integration points and the enforcement of Dirichlet boundaries require special treatment.

Using the variational formulation of a problem is the typical approach for the numerical solution of partial differential equations. Thus, it is not surprising to find this idea in other related publications. For example, Samaniego et al. followed this paradigm and translate the variational energy formulation of mechanical systems into the cost function of a deep learning model [SAG+19]. Next to a description of their physics-informed neural network implementation, the publication entailed several example applications from the field of computational solid mechanics, like phase-field modeling of fracture or bending of a Kirchhoff plate.

Motivated by the successful application of machine learning algorithms to the solution of high-order non-linear partial differential equations [BEJ19], E and Yu employed a deep residual neural network for solving variational problems [EY17]. Residual neural networks form an extension of feed-forward neural networks (cf. Section 3.1), that introduces additional connections between non-adjacent layers. In this way, some outputs can skip intermediate layers, which reduces the difficulty of training extremely deep networks [HZRS15]. Nabian and Meidani et al. also made use of this network architecture in order to build a surrogate model for high-dimensional random partial differential equations [NM19]. On the academic examples of diffusion and heat conduction they demonstrated, that their implementation is able to deal with both strong and variational formulation of the problem. The challenge of solving high-dimensional partial differential equations also motivated Sirignano and Spiliopoulos to investigate the applicability of deep learning in this context [SS18]. Their method can solve free-boundary partial differential equations in up to 200 dimensions. Apart from a special treatment of the free boundaries, they proposed a Monte-Carlo based method for the computation of second derivatives in higher dimensions.

The aforementioned papers dealt only with the inference of solutions to partial differential equations. An alternative approach for data-driven discovery of partial differential equations was presented by Rudy et al. [RBPK17]. They introduced a technique for discovering governing equations and physical laws by observing time-series measurements in the spatial domain. Their method allows the use of either an Eulerian or a Lagrangian frame of reference.

Chapter 5

Physics-informed Neural Network for the Non-linear Heat Equation

Joseph Fourier developed the heat equation in 1822 to describe the temperature distribution in a solid medium over time. Except for the formalization of the heat flow, the heat equation is often used as a canonical example for the class of parabolic partial differential equations. More generally, it is used for the description of diffusion processes in various scientific fields and, thus, also known as the diffusion equation [AP03].

Beyond the broad applicability, the choice of the problem is motivated by the importance of thermal analysis to additive manufacturing processes. A common method for printing complex metal components is laser bed powder fusion. Layer-by-layer, a laser melts and fuses a metallic powder into solid three-dimensional geometries. The effective numerical modeling of such processes is an ongoing research challenge since the simulation with conventional mesh-based methods is computationally expensive. The region of laser impact is usually highly localized compared to the surrounding spatial domain. Thus, the use of fine meshes is required to achieve accurate results in the area of interest. On top of that, extensive runtimes of the manufacturing process demand a large temporal domain with adequate discretization. To reduce the computational effort current research suggests the use of adaptive mesh-refinement techniques [KÖC$^+$18] or the derivation of effective physical models [KCRA19]. The apparent challenges show that the exploration of alternative methods is imperative.

As stated in Chapter 4, the development of physics-based learning models cause rising research interest and equally motivates this thesis to investigate their application in the context of thermal analysis. To pursue this line of study, a physics-informed neural network is used to predict the solution of a one-dimensional heat transfer example.

5.1 Heat Equation

Before looking into the application of physics-informed neural networks to concrete examples of heat transfer, a general description of the heat equation is stated. In particular, the evolution of temperature u as a function of space and time is described by a transient heat equation and boundary conditions of the following form

$$\begin{aligned} c\dot{u} - \nabla \cdot (\kappa \nabla u) &= s && \text{on } \mathcal{T} \times \Omega \\ n \cdot \kappa \nabla u &= h && \text{on } \mathcal{T} \times \Gamma_N \\ u &= g && \text{on } \mathcal{T} \times \Gamma_D \\ u(x,0) &= u_0 && \text{on } \Omega. \end{aligned} \tag{5.1}$$

Here, $\Omega \in \mathbb{R}^D$ represents the spatial domain in D dimensions whereas the temporal domain is denoted by $\mathcal{T}$. Together they form the tempo-spatial domain $\mathcal{T} \times \Omega$ with Neumann and Dirichlet boundaries represented by Γ_N and Γ_D, respectively. If the heat capacity $c(u)$ or the thermal conductivity $\kappa(u)$ are chosen to be temperature-dependent, a non-linear heat equation is obtained.

5.2 Time-continuous Solution of a Non-linear Heat Problem

5.2.1 Problem Statement

As a preliminary step toward the future analysis of more complex thermal problems, a one-dimensional heat transfer example in the tempo-spatial domain $\mathcal{T} \times \Omega = [0, 0.5] \times [0, 1]$ is investigated. The governing partial differential equation of the problem can be written as

$$c\frac{\partial u}{\partial t} - \frac{\partial}{\partial x}(\kappa \frac{\partial u}{\partial x}) = s \quad t \in [0, 0.5],\ x \in [0, 1]. \tag{5.2}$$

Here, s denotes an inhomogeneous source term, while $c(u)$ represents the heat capacity and $\kappa(u)$ the thermal conductivity, respectively. Further, the problem is subject to homogeneous Neumann boundary conditions

$$-\kappa\frac{\partial u(t,0)}{\partial x} = -\kappa\frac{\partial u(t,1)}{\partial x} = 0, \tag{5.3}$$

and the initial condition

$$u(0,x) = u_0. \tag{5.4}$$

Both the heat capacity $c(u)$ and the thermal conductivity $\kappa(u)$ are temperature-dependent and defined as

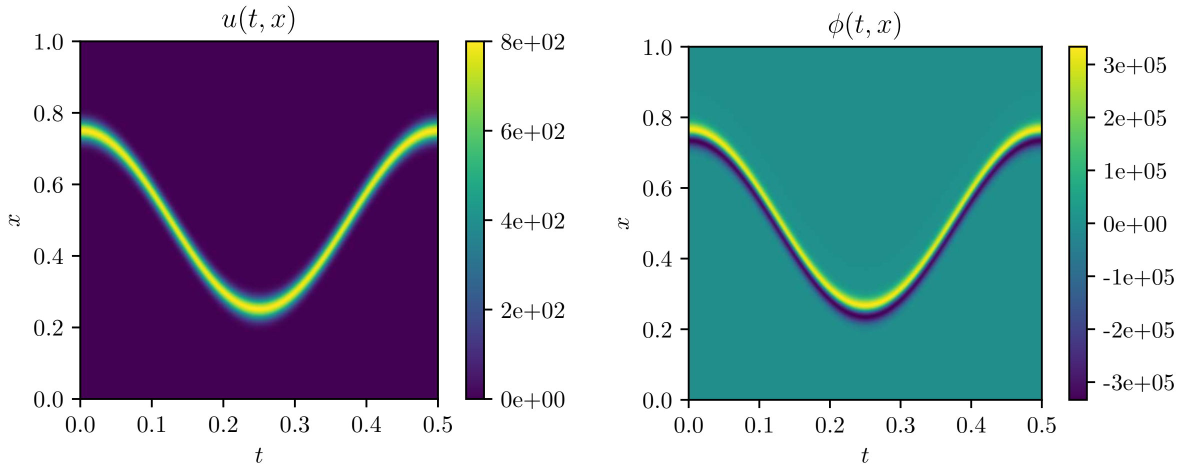
$$c(u) = 4.55 \times 10^{-4}u^2 - 5.78 \times 10^{-3}u + 5.849 \times 10^2, \tag{5.5}$$

$$\kappa(u) = 1.29 \times 10^{-2}u + 6.856. \tag{5.6}$$

In order to validate the network predictions, a manufactured solution of the following form is proposed

$$u = u_{\max} \exp\left(-\frac{(x-p)^2}{2\sigma^2}\right), \tag{5.7}$$

where $u_{\max} = 800$ and $\sigma = 0.02$. Normally, such Gaussian bell formulations are used to



(a) Manufactured solution u. **(b)** Manufactured heat flux ϕ.

Figure 5.1: Manufactured solution u and corresponding flux ϕ for the one-dimensional heat transfer problem. The solution was generated using MATLAB with a resolution of $t \times x = 201 \times 256$ points.

model the the laser-induced heat source s [KCRA19] which is traveling along a path p

$$p(t) = \frac{1}{4}\cos\left(\frac{2\pi t}{t_{\max}}\right) + \frac{1}{2}. \tag{5.8}$$

Here, the Gaussian bell term is chosen to represent the manufactured solution to the problem instead. Plugging Eq. (5.7) into Eq. (5.2) yields

$$s = \frac{\kappa u}{\sigma^2} + u\frac{x-p}{\sigma^2}\left[c\frac{\partial p}{\partial t} - \frac{x-p}{\sigma^2}\left(\kappa + u\frac{\partial \kappa}{\partial u}\right)\right]. \tag{5.9}$$

The manufactured solution u and the corresponding heat flux ϕ were generated with MATLAB at $t \times x = 201 \times 256$ discrete points (cf. Fig. 5.1).

The method of manufactured solutions is commonly used to validate numerical solvers on sufficiently complex examples [Roa02]. Nonetheless, the manufactured solution from Eq. (5.7) should by no means be interpreted as an realistic example. For instance, the source term s partially exhibits negative values which would in the example of laser powder bed fusion correspond to a laser extracting energy from the domain. However, the manufactured solution is a fast and direct way to benchmark the predictions of the physics-informed neural network and it may help to replicate certain characteristics of the problem found in real-world applications.

5.2.2 Implementation

The implementation for the results at hand is based on the code accompanying the paper by Raissi et al. [Rai20, RPK19]. In addition to adapting the code to the specific problem, several amendments are proposed in the following sections.

To begin with, the architecture of the physics-informed neural network is determined. In accordance with Section 4.2.1, a feed-forward neural network $u_{NN}(t,x;\boldsymbol{\Theta})$ is used to predict the hidden solution of the problem described in Eqs. (5.2) to (5.4). The inputs of the network are the temporal variable t and the spatial variable x. The network output $\hat{u}$ approximates the corresponding temperature distribution $u(t,x)$. To define the output of the physics-informed neural network $f_{NN}(t,x;\boldsymbol{\Theta})$ (cf. Fig. 4.3), the left-hand side of Eq. (5.2) is recalled as

$$f := c\frac{\partial u}{\partial t} - \frac{\partial}{\partial x}(\kappa\frac{\partial u}{\partial x}) = c\frac{\partial u}{\partial t} - \frac{\partial \kappa}{\partial x}\frac{\partial u}{\partial x} - \kappa\frac{\partial^2 u}{\partial x^2}. \tag{5.10}$$

According to Eqs. (5.5), (5.6), and (5.10), the code is adapted to consider for the left-hand side of the given heat equation:

```
def f(t,x):
u, u_x = self.net_u(x, t)
u_t = tf.gradients(u,t)[0]
u_xx = tf.gradients(u_x,x)[0]
k = 1.29 * 10 ** -2 * u + 6.856
k_u = 1.29 * 10 ** -2
k_x = k_u * u_x
c = 4.55 * 10 ** -4 * u ** 2 - 5.78 * 10 ** -3 * u + 5.849 * 10 ** 2
f = c * u_t - k_x * u_x - k * u_xx
return f
```

Here, `self.net_u(t,x)` denotes the neural network $u_{NN}(t,x;\boldsymbol{\Theta})$ that returns the predicted temperature $\hat{u}$ and the corresponding gradient $\frac{\partial \hat{u}}{\partial x}$. The latter is used to compute the heat flux and to enforce the Neumann boundary conditions. Correspondingly, the source term from Eq. (5.9) equates to the right-hand side of Eq. (5.2) and it is implemented as:

```
def s(t,x)
t_max = 0.5
sigma = 0.02
u_max = 800
p = 0.25 * tf.cos(2 * np.pi * t / t_max) + 0.5
p_t = - 0.5 * np.pi / t_max * tf.sin(2 * np.pi * t / t_max)
u_sol = u_max * tf.exp(-(x - p) ** 2 / (2 * sigma ** 2))
k_sol = 1.29 * 10 ** -2 * u_sol + 6.856
k_u_sol = 1.29 * 10 ** -2
c_sol = 4.55 * 10 ** -4 * u_sol ** 2 - 5.78 * 10 ** -3 * u_sol + 5.849 * 10 ** 2
fac_sigma = 1/(sigma ** 2)
s = fac_sigma * k_sol * u_sol + u_sol * (x - p) * fac_sigma * (
c_sol * p_t - (x - p) * fac_sigma * (k_sol + u_sol * k_u_sol))
return s
```

In order to train the network, the following cost function is defined

$$C = MSE_0 + MSE_b + MSE_f, \tag{5.11}$$

where

$$MSE_0 = \frac{1}{N_0} \sum_{i=1}^{N_0} \left(u_{NN} \left(0, x_0^i\right) - u_0^i \right)^2, \tag{5.12}$$

$$MSE_b = \frac{1}{N_b} \sum_{i=1}^{N_b} \left(\frac{\partial u_{NN}}{\partial x} \left(t_b^i, 0\right) \right)^2 + \frac{1}{N_b} \sum_{i=1}^{N_b} \left(\frac{\partial u_{NN}}{\partial x} \left(t_b^i, 1\right) \right)^2, \tag{5.13}$$

and

$$MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} \left(f_{NN} \left(t_f^i, x_f^i\right) - s \right)^2. \tag{5.14}$$

The loss function MSE_0 penalizes the error between the network prediction and N_0 random sample points $\{0, x_0^i, u_0^i\}_{i=1}^{N_0}$ at initial time $t = 0$ (cf. Eq. (5.4)). The term MSE_b enforces the Neumann boundary condition in Eq. (5.3) with N_b random samples $\{t_b, x_b^i, u_b^i\}_{i=1}^{N_b}$ on each boundary. Lastly, adding the error of the residual MSE_f ensures that the solution satisfies the governing Eq. (5.2). As proposed by Raissi et al. [RPK19], the N_f collocation points $\{t_f, x_f^i\}_{i=1}^{N_f}$ are generated by a Latin-hypercube sampling technique [Ste87]. Subsequently, a full-batch gradient-based optimization procedure searches for the optimal network parameters $\boldsymbol{\Theta}^*$ by minimizing the cost function in Eq. (5.11). In addition to the L-BFGS method, a preceding minimization with the Adam optimizer is employed [KB17]. This combination was adapted from the existing code [Rai20] and has empirically proven to be the most robust approach throughout this study.

To evaluate the performance of the model, the manufactured solution is used to generate a test set of N discrete points covering the whole domain $t \times x = [0, 0.5] \times [0, 1]$. The euclidean relative error measure for the test set is computed as follows

$$\frac{||\hat{\boldsymbol{u}} - \boldsymbol{u}||_2}{||\boldsymbol{u}||_2} = \frac{(\sum_{i=1}^{N} |\hat{u}_i - u_i|^2)^{1/2}}{(\sum_{i=1}^{N} |u_i|^2)^{1/2}}, \tag{5.15}$$

where $\hat{u}_i$ denotes the network prediction and u_i corresponds to the manufactured solution. A relative error allows to compare different approaches, which will be introduced later in this section.

Algorithm 4 summarizes the previously introduced training procedure employed for a continuous prediction of the temperature distribution $u(t, x)$. After the algorithm terminated, the network $u_{NN}(t, x; \boldsymbol{\Theta})$ with trained parameters $\boldsymbol{\Theta}$ is used to predict the continuous temperature distribution over the whole domain $t \times x = [0, 0.5] \times [0, 1]$.

If not indicated differently, the results in the coming sections were conducted with the choice of parameters presented in Table 5.1 Additional parameters of the Adam and L-BFGS optimizers default to the values chosen by Raissi et al. [Rai20]. The current implementation is based on Tensorflow version 1.15 and Python version 3.7. The computations were executed on a laptop computer with an Intel Core i7-7700HQ @ 2.80GHz CPU and a NVIDIA GeForce GTX 1050 graphics card.

Algorithm 4 Training a physics-informed neural network for the continuous solution of the problem described in Eq. (5.2).

```
Require: training data for initial condition $\{0, x_0^i, u_0^i\}_{i=1}^{N_0}$
Require: training data for boundary condition $\{t_b, x_b^i, u_b^i\}_{i=1}^{N_b}$
  generate $N_f$ collocation points with Latin-hypercube sampling $\{t_f, x_f^i\}_{i=1}^{N_f}$
  define network architecture (input, output, hidden layers, hidden neurons)
  initialize network parameters $\boldsymbol{\Theta}$: weights $\{\boldsymbol{W}^l\}_{l=1}^{L}$ and biases $\{\boldsymbol{b}^l\}_{l=1}^{L}$ for all layers $L$
  set hyperparameters for Adam optimizer (epochs, learning rate $\alpha$, ...)
  set hyperparameters for L-BFGS optimizer (convergence criterion, max iterations, ...)
  loop
      $\hat{\boldsymbol{u}}_0 \leftarrow \boldsymbol{u}_{NN}(\boldsymbol{0}, \boldsymbol{x}_0; \boldsymbol{\Theta})$
      $\frac{\partial \hat{\boldsymbol{u}}_b}{\partial x} \leftarrow \frac{\partial \boldsymbol{u}_{NN}}{\partial x}(\boldsymbol{t}_b, \boldsymbol{x}_b; \boldsymbol{\Theta})$
      $\boldsymbol{f} \leftarrow \boldsymbol{f}_{NN}(\boldsymbol{t}_f, \boldsymbol{x}_f; \boldsymbol{\Theta})$
      compute $MSE_0$, $MSE_b$, $MSE_f$                          ▷ cf. Eqs. (5.12) to (5.14)
      computer cost function: $C \leftarrow MSE_0 + MSE_b + MSE_f$
      update parameters: $\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \alpha \frac{\partial C}{\partial \boldsymbol{\Theta}}$       ▷ Adam or L-BFGS
  end loop
  for all epochs do
      run loop with Adam optimizer
  end for
  repeat
      run loop with L-BFGS optimizer
  until convergence or max. iterations
```

no. of hidden layers:	3
no. hidden neurons:	20 (per layer)
hidden activation function:	Tanh
output activation function:	linear
no. of initial data points N_0:	100
no. of boundary data points N_b:	50 (on each boundary)
no. of collocation points N_f:	20 000
no. of *epochs*:	10 000 (Adam optimizer)
learning rate α:	0.001 (Adam optimizer)
max iterations:	50 000 (L-BFGS optimizer)
tolerance for convergence:	machine precision ϵ (L-BFGS optimizer)

Table 5.1: Parameters for the continuous-time inference.

.

5.2.3 First Results

The first attempt to run the algorithm, solely adapting the previously problem-specific functions, led to no satisfactory result. Even for an extended number of gradient descent steps and different learning rates, the algorithm did not converge to an acceptable solution. The L-BFGS optimizer usually terminated after very few iterations.

To determine a cause for this behavior, the individual loss terms were plotted against the number of iterations. Figure 5.2 shows that the loss regarding the collocation points MSE_f

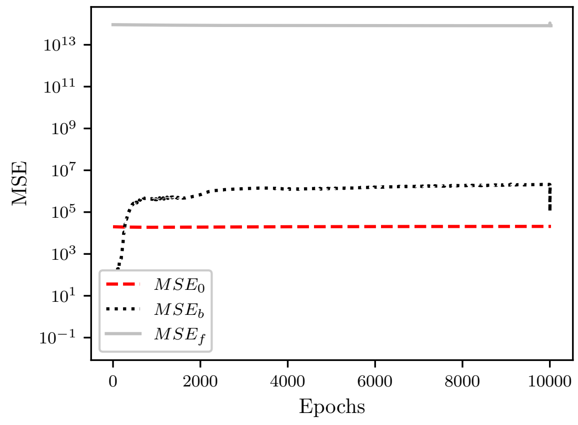


Figure 5.2: Loss term history of the first attempt.

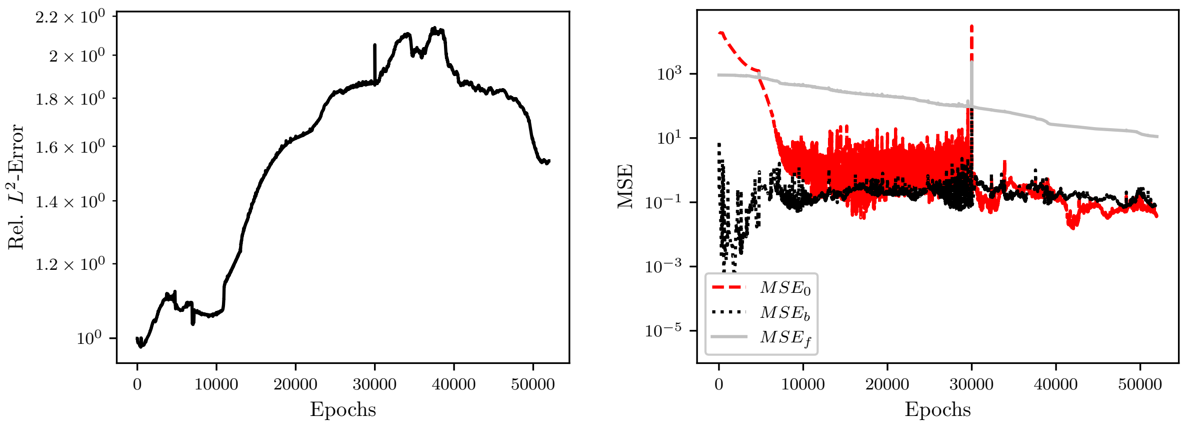
was several magnitudes larger compared to the other losses and dominated the cost function. As a result, the network mainly learned to satisfy the partial differential equation at the collocation points disregarding the boundary and initial conditions. Nabian and Meidani addressed this issue by introducing weights adjusting the relative importance of each term in the cost function [NM19]. They further highlighted the similarity between those weights and the formulation for boundary conditions used in the finite element collocation method [BG09]. Adding the weight factors to Eq. (5.11) yields a new cost function

$$C = \gamma_0 \, MSE_0 + \gamma_b \, MSE_b + \gamma_f \, MSE_f. \tag{5.16}$$

To continue the study the weight factor for MSE_f was empirically determined as $\gamma_f = 1 \times 10^{-11}$ while γ_0 and γ_b were set to 1. After updating the cost function, the physics-informed neural network was trained again to infer the solution of the problem stated in Eqs. (5.2) to (5.4). Even with extensive hyperparameter tuning, the network learned very slowly as the plot of the training losses in Fig. 5.3b suggests. In particular the learning rate was set to $\alpha = 0.01$ and 30 000 Adam iterations were scheduled. Afterward, the L-BFGS optimization continued the parameter search and terminated after additional 20 000 iterations. The corresponding predictions of the temperature distribution $u(t, x)$ and the heat flux $\phi(t, x)$ are depicted in Fig. 5.4. It can be seen that the network began to recognize certain features of the underlying solution, but still failed to produce a satisfactory result as the error plot in Fig. 5.3a confirms.

5.2.4 Nondimensionalization

A closer look at the original paper by Raissi et al. and other publications about physics-informed networks revealed another interesting insight [RPK19, NM19, SAG+19]. It was noticed that for most investigated systems the prediction targets laid in a range between -1 and 1. As known from the manufactured solution, the temperature in the present heat transfer example ranges from 0 to 800. In order to scale down the problem without changing its physical characteristics, a partial nondimensionalization was applied. Nondimensionalization is a technique often used in simulations of mechanical problems [LP16]. For instance, it allows the execution of fluid flow experiments in different length and time scales or helps to improve the solver stability in complex numerical simulations. The crucial part is to determine suitable



(a) Relative L^2-Error over the number of training iterations.

(b) Weighted loss terms over the number of training iterations.

Figure 5.3: Error and weighted loss history of the modified example. Hyperparameters: *epochs* = 30 000, learning rate $\alpha = 0.01$ and weight factors $\gamma_0 = \gamma_b = 1, \gamma_f = 1 \times 10^{-11}$.

scaling factors for the underlying problem. Typically, a dimensionless variable q is defined as

$$\bar{q} = \frac{q - q_0}{q_c},$$

where q_0 is a reference value, that often equates to zero, and q_c denotes a characteristic value of q determining the scaling [LP16]. Since q_c has the same units as q, the dimensionless variable $\bar{q}$ is obtained. The goal is to scale the temperature u into the range of unity. Neglecting the reference temperature and choosing the characteristic temperature as $u_c = u_{\max}$, the dimensionless temperature variable is defined as

$$\bar{u} = \frac{u}{u_c} = \frac{u}{u_{\max}}. \tag{5.17}$$

Inserting Eq. (5.17) into the problem defined in Eq. (5.2) yields

$$c\frac{\partial \bar{u} u_c}{\partial t} - \frac{\partial}{\partial x}(\kappa \frac{\partial \bar{u} u_c}{\partial x}) = s, \tag{5.18}$$

which is equivalent to

$$c\frac{\partial \bar{u}}{\partial t} - \frac{\partial}{\partial x}(\kappa \frac{\partial \bar{u}}{\partial x}) = \frac{s}{u_c}. \tag{5.19}$$

This introduces a scaled source term $\tilde{s}$ defined as

$$\tilde{s} = \frac{s}{u_{\max}} = \frac{\kappa \tilde{u}}{\sigma^2} + \tilde{u}\frac{x-p}{\sigma^2}\left[c\frac{\partial p}{\partial t} - \frac{x-p}{\sigma^2}\left(\kappa + u\frac{\partial \kappa}{\partial u}\right)\right], \tag{5.20}$$

where

$$\tilde{u} = \exp\left(-\frac{(x-p)^2}{2\sigma^2}\right). \tag{5.21}$$

Based on the scaled problem, a new solution was generated using MATLAB. Plots of the dimensionless solution $\bar{u}$ and the corresponding flux $\bar{\phi}$ are shown in Fig. 5.5.

As a next step, the physics-informed network was adapted to consider the proposed changes.

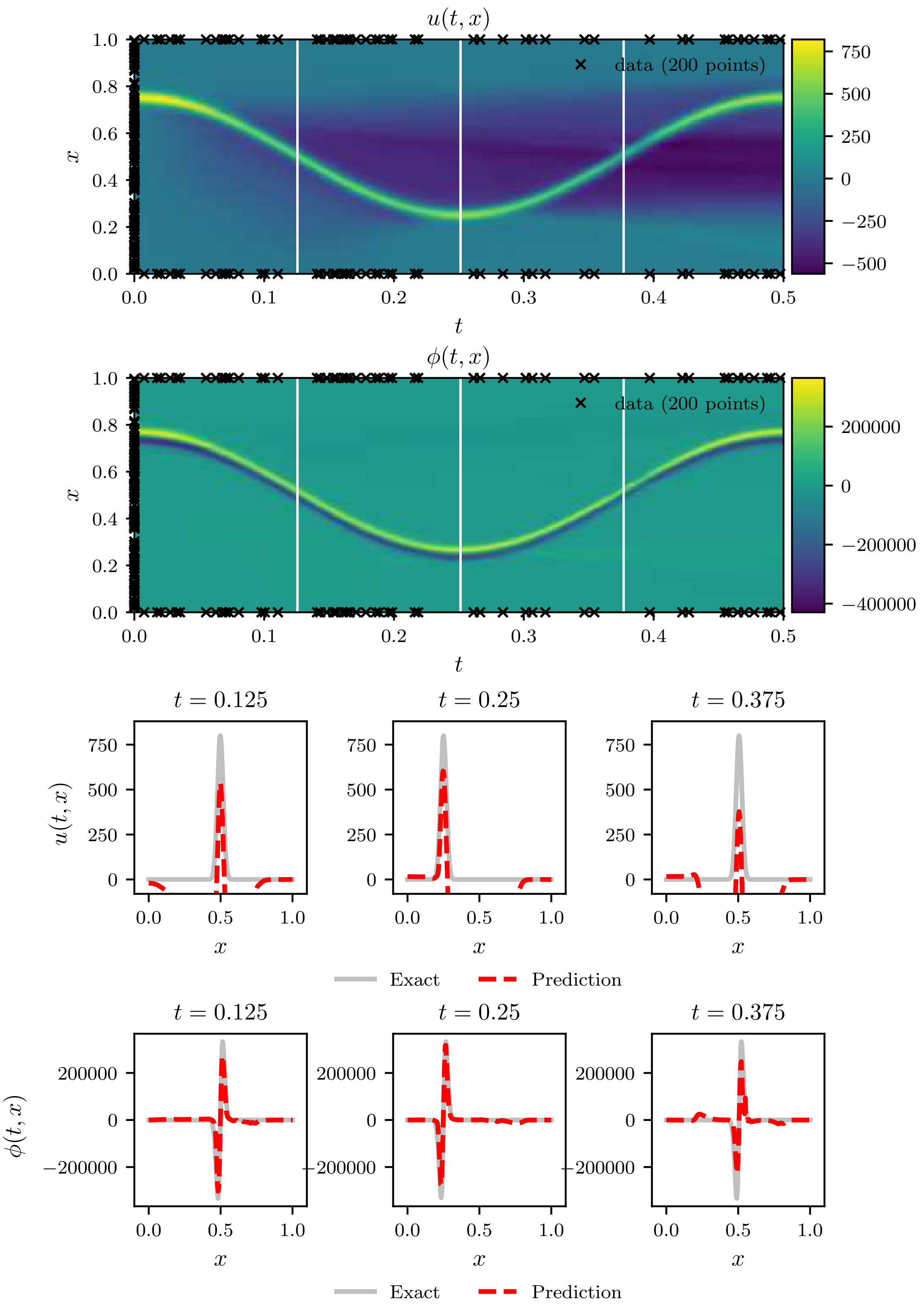
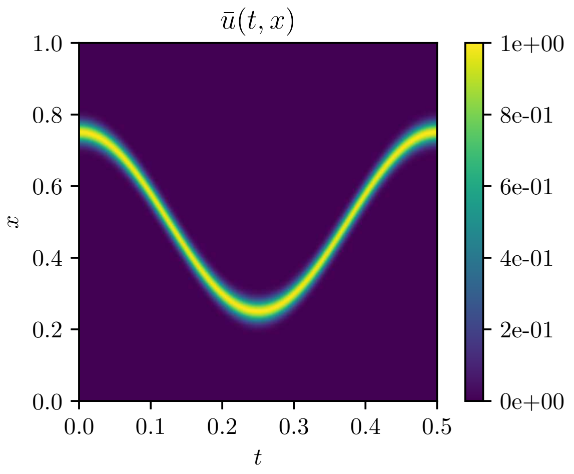
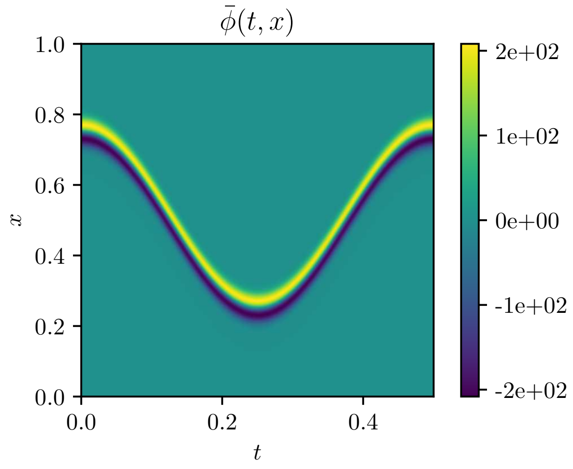


Figure 5.4: Prediction of temperature distribution u and the heat flux ϕ for the modified example. Hyperparameters: *epochs* = 30 000, learning rate $\alpha = 0.01$ and weight factors $\gamma_0 = \gamma_b = 1, \gamma_f = 1 \times 10^{-11}$. Top: Approximated solution and location of time snapshots (white lines). Bottom: Comparison of predicted and exact solution at distinct snapshots.

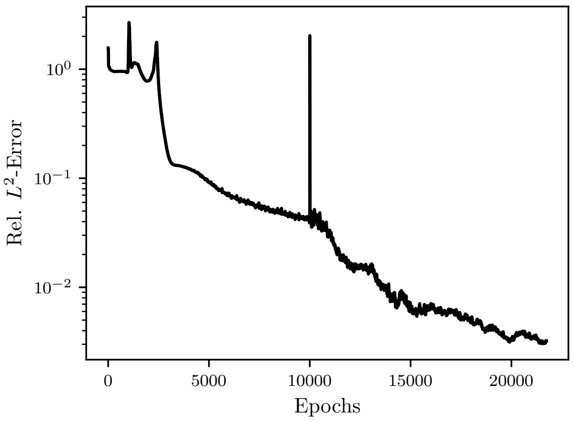


(a) Manufactured and scaled temperature distribution $\bar{u}$.

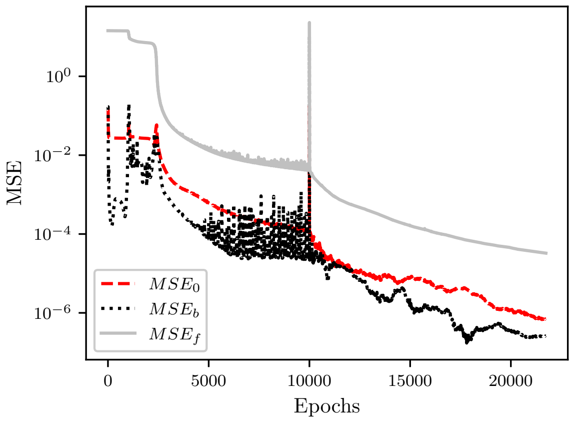


(b) Manufactured and scaled heat flux $\bar{\phi}$.

Figure 5.5: Manufactured solution of the nondimensionalized problem.



(a) Relative L^2-Error over the number of training iterations.



(b) Weighted loss terms over the number of training iterations.

Figure 5.6: Error and weighted loss history of the nondimensionalized example. Hyperparameters: *epochs* $= 10\,000$, learning rate $\alpha = 0.001$ and weight factors $\gamma_0 = \gamma_b = 1, \gamma_f = 1 \times 10^{-7}$.

The results for the prediction of the scaled temperature distribution $\bar{u}$ and the corresponding heat flux $\bar{\phi}$ are depicted in Fig. 5.7. Figure 5.6 presents the related error and loss history. In combination with an appropriately chosen weight for the loss MSE_f ($\gamma_f = 1 \times 10^{-7}$), the network was able to predict an accurate solution as the L-BFGS optimizer converged. More precisely, the relative prediction error for $\bar{u}$ was measured at $3.234\,353 \times 10^{-3}$ and the relative error of $\bar{\phi}$ yielded $2.390\,841 \times 10^{-3}$.

5.2.5 Adaptive Cost Function

Finding the right weight factor for a problem involves great effort. Since introducing another hyperparameter is not desirable, an alternative approach for weight balancing is proposed. The idea is to scale the network outputs individually for each loss term. In particular, the predictions are divided by the absolute maximum of the target values. For example, the loss MSE_f depends on the network output f_{NN} and the source term s. Since the source term s is known at every point in the domain, the scaling factor $\max|s|$ can be easily determined. Similarly, the remaining factors are set. The loss for the initial condition is divided by $\max|u_0|$, where u_0 represents the known solution at time $t = 0$. In the same way, the loss on the boundary could be adapted. Since the present case imposes homogeneous Neumann boundary conditions, scaling the boundary loss MSE_b was neglected to avoid a division by zero. Including the proposed scaling scheme in the losses yields a new cost function

$$\bar{C} = \overline{MSE}_0 + \overline{MSE}_b + \overline{MSE}_f, \tag{5.22}$$

with

$$\overline{MSE}_0 = \frac{1}{N_0} \sum_{i=1}^{N_0} \left(\frac{u_{NN}\left(0, x_0^i\right) - u_0^i}{\max|u_0|} \right)^2, \tag{5.23}$$

$$\overline{MSE}_b = MSE_b \tag{5.24}$$

and

$$\overline{MSE}_f = \frac{1}{N_f} \sum_{i=1}^{N_f} \left(\frac{f_{NN}\left(t_f^i, x_f^i\right) - s}{\max|s|} \right)^2. \tag{5.25}$$

Employing the introduced cost function, the solution inference for the nondimensionalized problem from Section 5.2.4 was repeated. The plots of the error and losses in Fig. 5.8 suggest that the optimization procedure converged. The prediction accuracy of $\bar{u}$ was measured at $1.510\,847 \times 10^{-2}$ and the error of $\bar{\phi}$ at $1.437\,716 \times 10^{-2}$ according to Eq. (5.15). Figure 5.9 presents the predictions of the temperature distribution $\bar{u}$ and the corresponding heat flux $\bar{\phi}$. The results imply that the adaptive cost function yields comparable accuracy to a manual fitting of the cost function.

5.2.6 Alternative Output Function

The previous investigations revealed that scaling the temperature to a range from 0 to 1 led to faster training of the network and more accurate results. However, a general problem of nondimensionalizing the target variable is to find an appropriate scaling factor. Apart from the boundaries, no information about the hidden solution is given. Whenever, for instance,

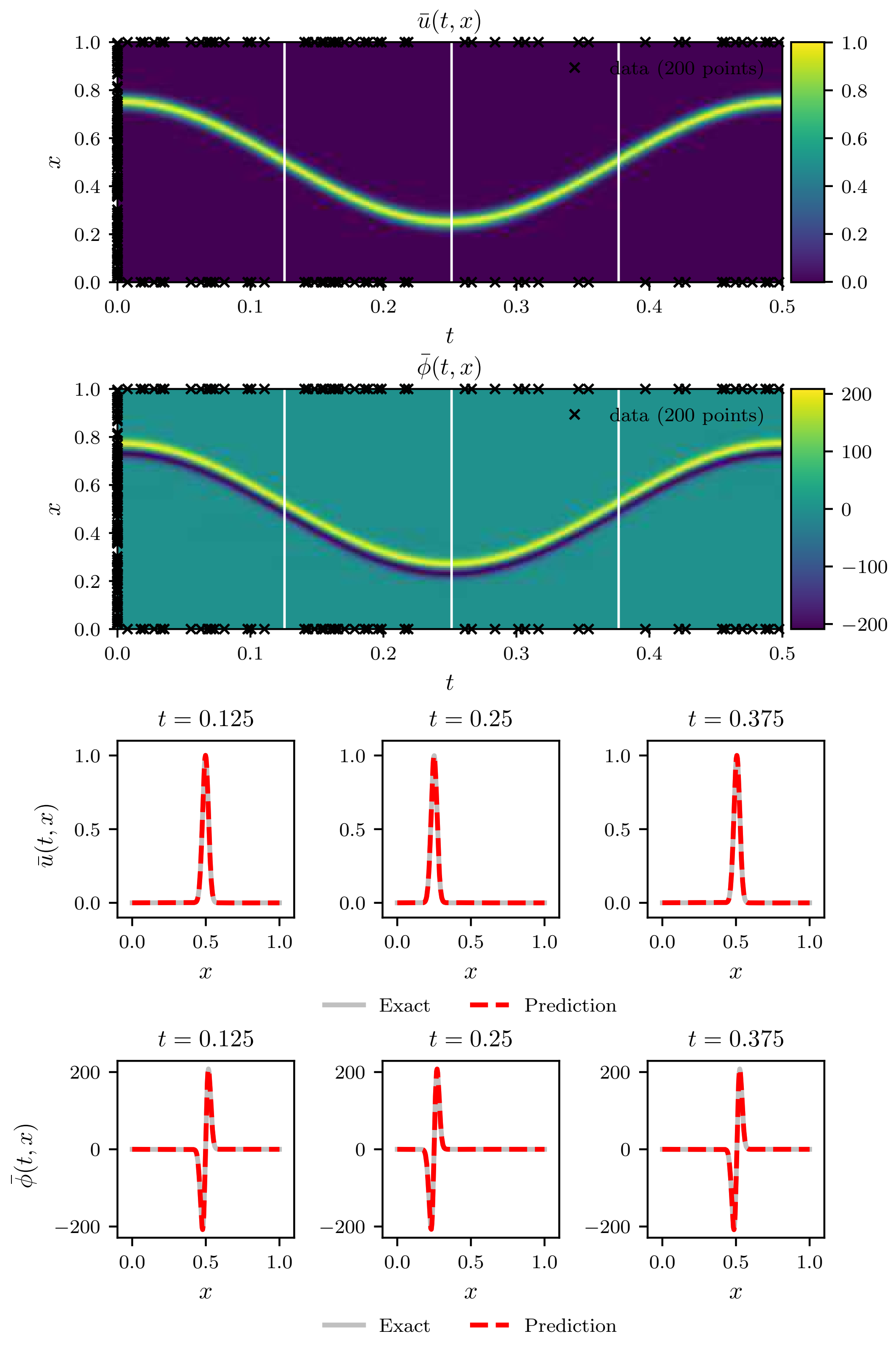
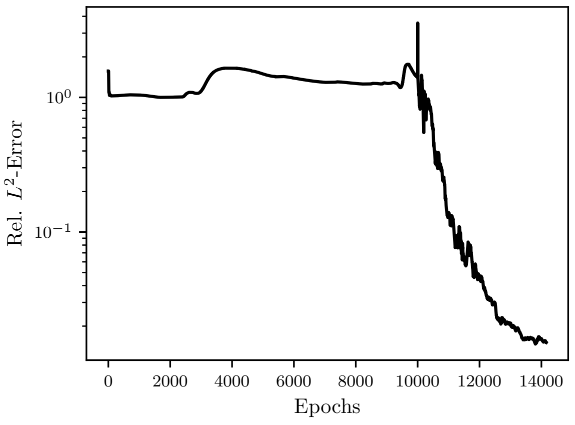
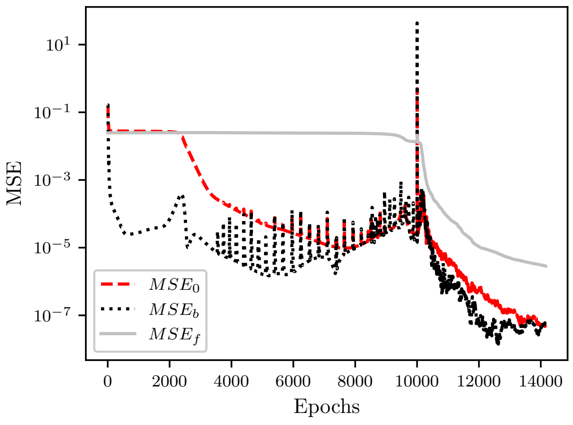


Figure 5.7: Predictions of the temperature distribution $\bar{u}$ and the heat flux $\bar{\phi}$ for the nondimensionalized example. Hyperparameters: *epochs* = 10 000, learning rate $\alpha = 0.001$ and weight factors $\gamma_0 = \gamma_b = 1, \gamma_f = 1 \times 10^{-7}$. Top: Approximated solution and location of time snapshots (white lines). Bottom: Comparison of predicted and exact solution at distinct snapshots.



(a) Relative L^2-Error over the number of training iterations.



(b) Weighted loss terms over the number of training iterations.

Figure 5.8: Error and weighted loss history of the nondimensionalized example with adaptive cost function. Hyperparameters: $epochs = 10\,000$ and learning rate $\alpha = 0.001$.

the initial condition u_0 is small compared to the hidden solution, a scaling by $\max|u_0|$ might be insufficient or could even lead to an up-scaling if $\max|u_0| < 1$. To summarize, the applicability of nondimensionalization is problem-dependent and not always reasonable.

From the limited capabilities to scale down the predictions, the idea arose to increase the expressive power of the network outputs. In other words, introducing an alternative activation function, should enable the output neuron to learn large values faster. Normally, the literature proposes a linear output function for regression tasks [GBC16]. In contrast, the hyperbolic sine is introduced as the activation function for the output layer (cf.Fig. 5.10). To test the proposed idea, the algorithm was adapted correspondingly. The implementation was straightforward, since only the activation for the output neuron had to be changed. Neglecting the nondimensionalization, the original problem as described in Eqs. (5.2) to (5.4) was revisited. In combination with the adaptive cost function from Eq. (5.22), the physics-informed neural network was trained to predict the latent temperature distribution. Figure 5.11 depicts the error and loss history, while the approximations of the temperature distribution u and the heat flux ϕ are presented in Fig. 5.12. The relative error of u was measured at $3.232\,114 \times 10^{-2}$ and the error of ϕ was measured at $1.465\,904 \times 10^{-2}$, respectively.

It is important to note that the loss term MSE_b had to be scaled down by a factor of 2×10^4 to obtain a satisfactory solution for the chosen hyperparameters. The successful predictions conclude this section on the continuous solution inference of a non-linear heat equation. The results are discussed further in Section 5.5.

5.3 Discrete Solution of a Non-linear Heat Problem

5.3.1 Problem Statement

For the study of the discrete solution method the problem from Eqs. (5.2) to (5.4) is recalled. The governing partial differential equation is observed in the tempo-spatial domain $\mathcal{T} \times \Omega =$

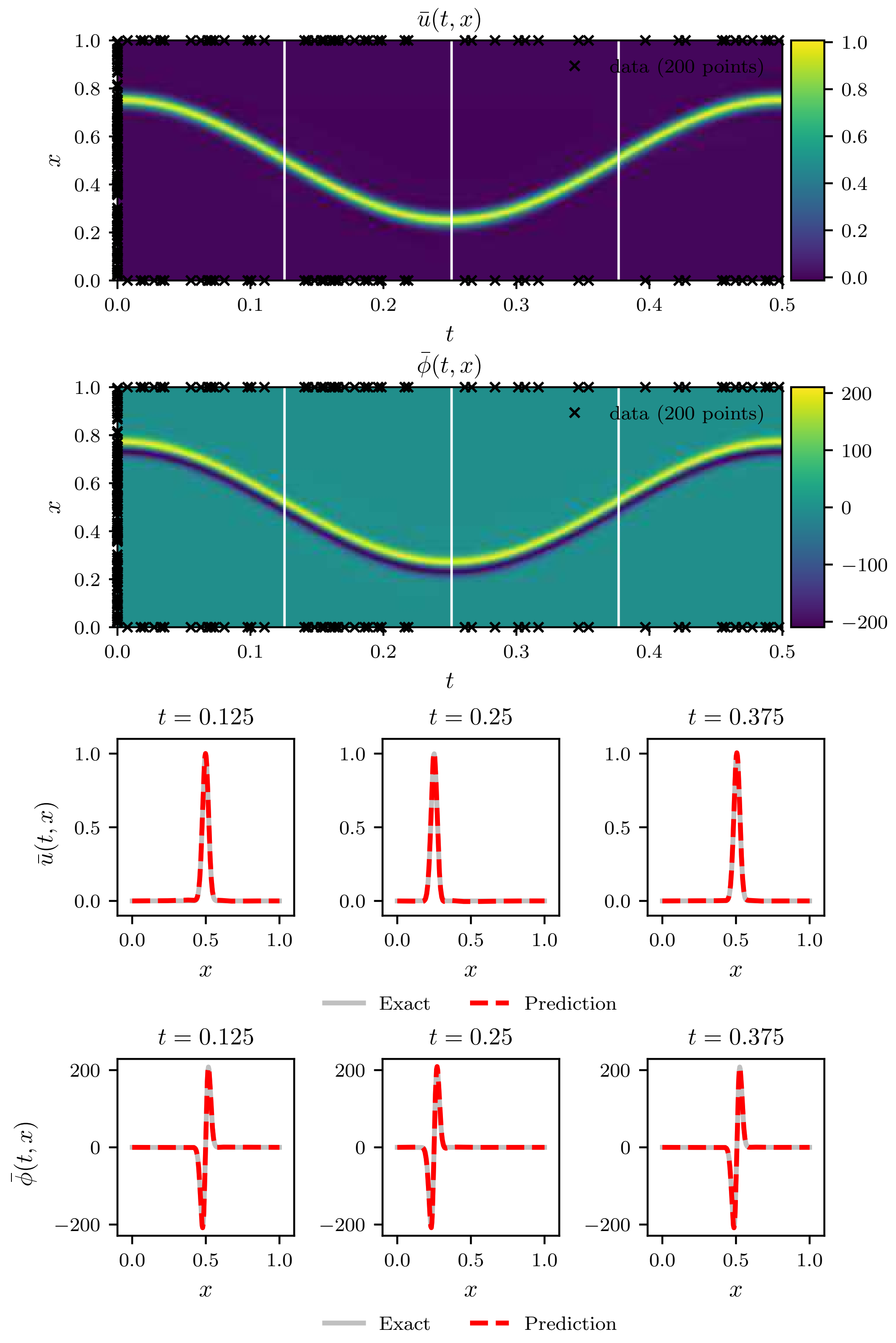


Figure 5.9: Prediction of the temperature distribution $\bar{u}$ and the heat flux $\bar{\phi}$ for the nondimensionalized example with adaptive cost function. Hyperparameters: *epochs* = 10 000 and learning rate $\alpha = 0.001$. Top: Approximated solution and location of time snapshots (white lines). Bottom: Comparison of predicted and exact solution at distinct snapshots.

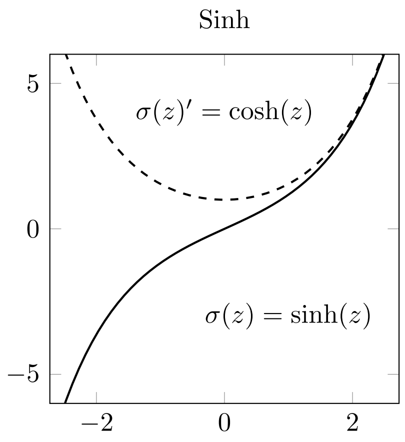
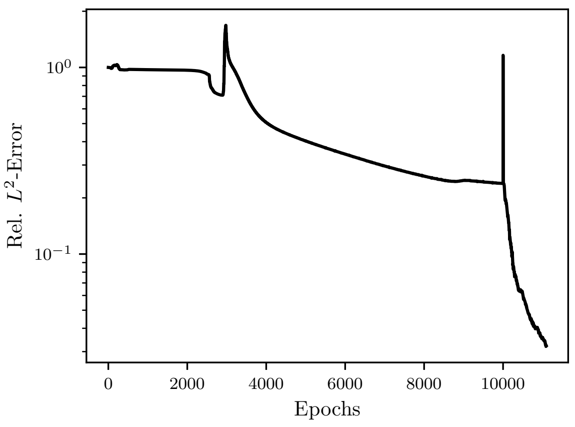
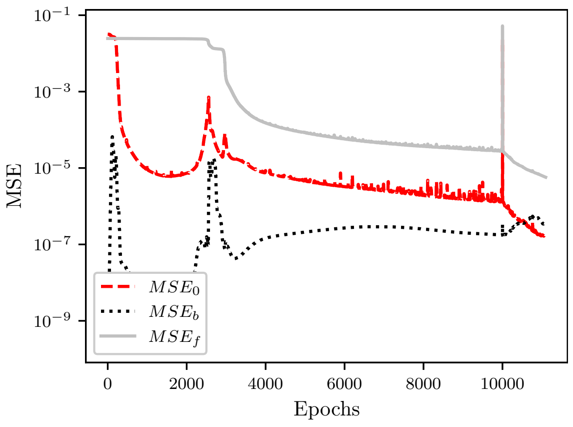


Figure 5.10: Hyperbolic sine activation $\sigma(z) = \sinh(z)$ and its derivative $\sigma(z)' = \sinh(z)' = \cosh(z)$.



(a) Relative L^2-Error over the number of training iterations.



(b) Weighted loss terms over the number of training iterations.

Figure 5.11: Error and weighted loss history of the modified example with adaptive cost function and Sinh output activation. Hyperparameters: *epochs* = 10 000 and learning rate $\alpha = 0.001$.

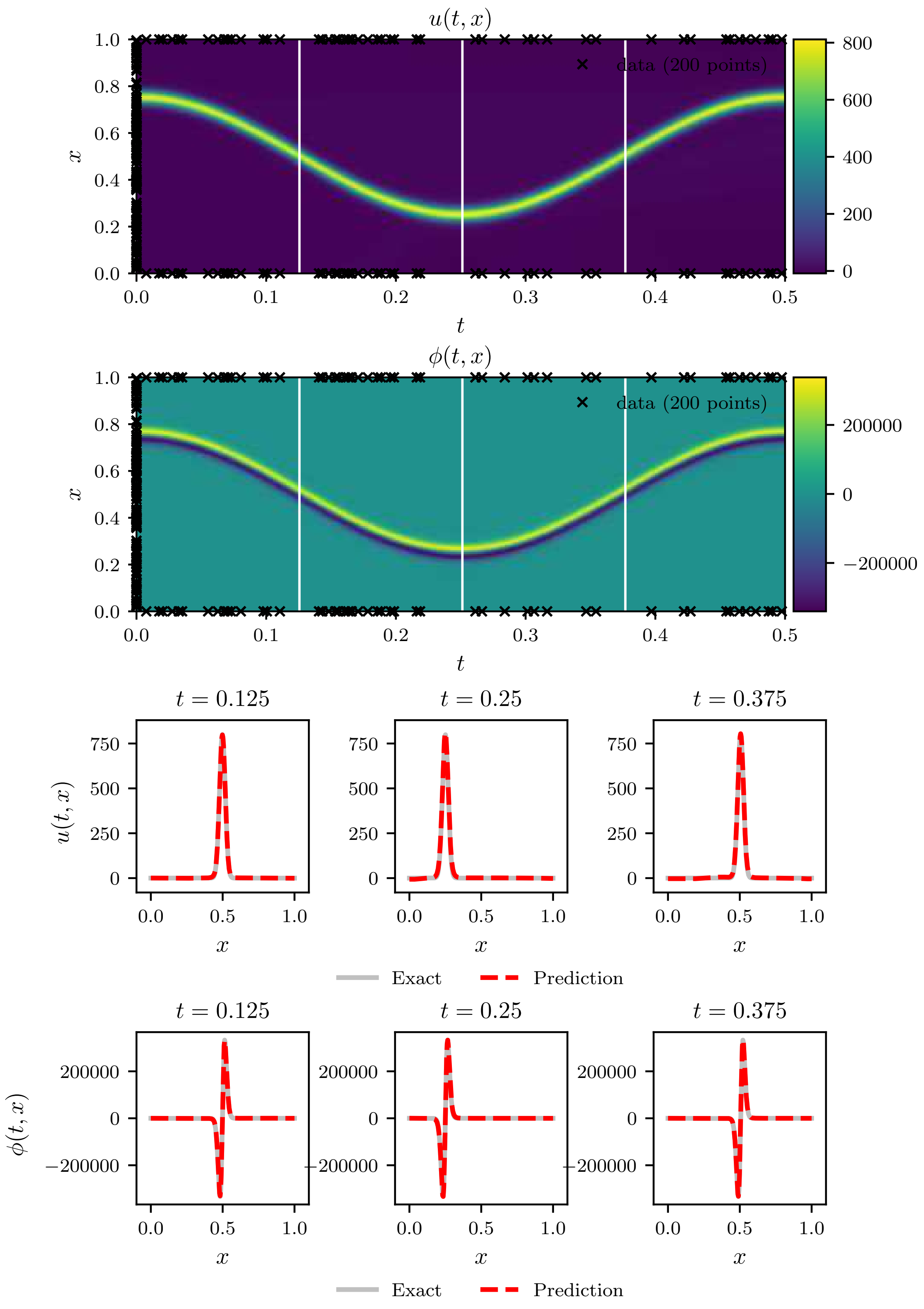


Figure 5.12: Prediction of the temperature distribution u and corresponding heat flux ϕ of the modified example with adaptive cost function and Sinh output activation. Hyperparameters: $epochs = 10\,000$ and learning rate $\alpha = 0.001$.Top: Approximated solution and location of time snapshots (white lines). Bottom: Comparison of predicted and exact solution at distinct snapshots.

$[0, 0.5] \times [0, 1]$ and is written as

$$c\frac{\partial u}{\partial t} - \frac{\partial}{\partial x}(\kappa \frac{\partial u}{\partial x}) = s \quad t \in [0, 0.5],\ x \in [0, 1]. \tag{5.26}$$

The temperature-dependent coefficients, namely, the heat capacity $c(u)$ and thermal conductivity $\kappa(u)$ are defined as

$$c(u) = 4.55 \times 10^{-4}u^2 - 5.78 \times 10^{-3}u + 5.849 \cdot 10^2, \tag{5.27}$$

$$\kappa(u) = 1.29 \times 10^{-2}u + 6.856. \tag{5.28}$$

In contrast to the previous case, homogeneous Dirichlet boundary conditions are assumed

$$u(0, t) = u(1, t) \approx 0. \tag{5.29}$$

Further, the problem is subject to the following initial condition

$$u(0, x) = u_0. \tag{5.30}$$

Again, the Gaussian bell formulation represents the manufactured solution to the problem

$$u = u_{\max} \exp\left(-\frac{(x-p)^2}{2\sigma^2}\right), \tag{5.31}$$

where $u_{\max} = 800$ and $\sigma = 0.02$. The path p takes on the form

$$p(t) = \frac{1}{4}\cos\left(\frac{2\pi t}{t_{\max}}\right) + \frac{1}{2}. \tag{5.32}$$

Inserting Eq. (5.31) into Eq. (5.26) yields the source term

$$s = \frac{\kappa u}{\sigma^2} + u\frac{x-p}{\sigma^2}\left[c\frac{\partial p}{\partial t} - \frac{x-p}{\sigma^2}\left(\kappa + u\frac{\partial \kappa}{\partial u}\right)\right]. \tag{5.33}$$

Since the manufactured solution u and the corresponding heat flux ϕ remain unchanged, the previously generated data set is used (cf. Fig. 5.1).

5.3.2 Implementation

The implementation of the discrete method is based on the code by Raissi and follows the scheme introduced in Section 4.2.1 [Rai20].

First, the architecture of the physics-informed surrogate model for a desired time step $t^{n+1} = t^n + \Delta t$ and q stages is specified. Since an implicit time discretization is applied, the feed-forward neural network $\boldsymbol{u}_{NN}(x; \boldsymbol{\Theta})$ only takes the spatial variable x as an input. Next to the desired solution $\hat{u}^{n+1}(x)$ at time t^{n+1}, the network predicts the intermediate solutions $\hat{u}^{n+c_i}$ for all stages $i = 1, \ldots, q$. Here, the parameters a_{ij}, b_j, and c_j for the implicit Runge-Kutta time stepping scheme are selected from a Butcher table corresponding to the number of selected stages [Ise08]. To apply the temporal discretization, the problem described in

Eq. (5.26) is rearranged

$$\frac{\partial u}{\partial t} = \left(\frac{\partial \kappa}{\partial x} \frac{\partial u}{\partial x} + \kappa \frac{\partial^2 u}{\partial x^2} + s \right) \bigg/ c. \tag{5.34}$$

Following the modified Runge-Kutta time stepping scheme from Eq. (4.15) and using Eq. (5.34) the physics-informed neural network $\boldsymbol{f}_{NN}(x; \boldsymbol{\Theta})$ is defined. The corresponding output $\hat{u}_i^n(x)$ for $i = 1, \ldots, q+1$ consists of the solution at time t^n as well as the intermediate results defined by Eq. (4.15).

So far, the initial and boundary conditions have been enforced weakly in the cost functions. However, it is also possible to apply constraints in a strong sense. To do so, the solution u is modified to satisfy the boundary conditions for any given input. Following the generalized approach of Lagaris et al. [LLF98], the network output is adapted to consider for the homogeneous Dirichlet boundary conditions as defined in Eq. (5.29). Since the discrete solution is only dependent on the spatial variable x, the following formulation is obtained

$$\left[\hat{u}^{n+c_1}, \ldots, \hat{u}^{n+c_q}, \hat{u}^{n+1}(x)\right] = (1-x)x\, \boldsymbol{u}_{NN}(x; \boldsymbol{\Theta}) \quad i = 1, \ldots, q. \tag{5.35}$$

To demonstrate the simplicity of the approach using a near-mathematical notation, a code snippet from the physics-informed neural network is presented:

```
net_U0(self, x):
U1 = (x-1)*x*self.neural_net(x, self.weights, self.biases)
...
F = (k_x * U_x + k * U_xx + s) / c
U0 = U1 - self.dt * tf.matmul(F, self.IRK_weights.T)
return U0
```

Here, `neural_net` denotes the neural network $\boldsymbol{u}_{NN}(x; \boldsymbol{\Theta})$ predicting the solution at time t^{n+1} and the intermediate steps, while `IRK_weights` stores the time-stepping parameters a_{ij} and b_j. The differential operators `U_x` and `U_xx` are computed with automatic differentiation and the material coefficients are defined according to Eqs. (5.27) and (5.28).

As a result, the cost function for training the network simplifies to a single loss term, which omits the need of identifying suitable weighting factors. With the output of the physics-informed neural network $\hat{u}_i^n(x)$ for $i = 1, \ldots, q+1$, the resulting cost function is written as

$$C = MSE_n, \tag{5.36}$$

where

$$MSE_n = \sum_{j=1}^{q+1} \sum_{i=1}^{N_n} \left(\hat{u}_j^n \left(x^{n,i} \right) - u^{n,i} \right)^2. \tag{5.37}$$

In contrast to the squared error sum proposed by Raissi et al. [RPK19], a mean squared error loss is chosen. The term MSE_n computes the prediction error of the physics-informed neural network at N_n randomly sampled points $\{x^{n,i}, u^{n,i}\}_{i=1}^{N_n}$ of the solution at initial time t^n. To learn the shared set of optimal parameters $\boldsymbol{\Theta}^*$, the combination of Adam optimizer and L-BFGS method is employed to minimize the cost function Eq. (5.36). In accordance with the previous example from Section 5.2, a test set of N discrete points describing $x = [0, 1]$ at time t^n is taken from the previously generated manufactured solution. The relative error measure for the test set computes as follows

$$\frac{||\hat{\boldsymbol{u}}^{n+1} - \boldsymbol{u}^{n+1}||_2}{||\boldsymbol{u}^{n+1}||_2} = \frac{(\sum_{i=1}^{N} |\hat{u}_i^{n+1} - u_i^{n+1}|^2)^{1/2}}{(\sum_{i=1}^{N} |u_i^{n+1}|^2)^{1/2}}, \tag{5.38}$$

where $\hat{u}_i^{n+1}$ denotes the network prediction and u_i^{n+1} represents the manufactured solution at time t^{n+1}.

To summarize the discrete time method for solution inference, Algorithm 5 describes the previously elaborated steps of the training procedure. After the algorithm terminated, the

Algorithm 5 Training a discrete physics-informed neural network for a single time step $t^{n+1} = t^n + \Delta t$ and q stages.

```
Require: training data for initial condition $\{x^{n,i}, u^{n,i}\}_{i=1}^{N_n}$
  define initial time step $t^n$ and time step size $\Delta t$
  define number of stages $q$
  define network architecture (input, hidden layers, hidden neurons)
  initialize output layer with $q + 1$ neurons
  initialize network parameters $\boldsymbol{\Theta}$: weights $\{\boldsymbol{W}^l\}_{l=1}^{L}$ and biases $\{\boldsymbol{b}^l\}_{l=1}^{L}$ for all layers $L$
  set hyperparameters for Adam optimizer (epochs, learning rate $\alpha$, ...)
  set hyperparameters for L-BFGS optimizer (convergence criterion, max iterations, ...)
  loop
     $\boldsymbol{U}_n \leftarrow \boldsymbol{F}_{NN}(\boldsymbol{x}_n; \boldsymbol{\Theta})$
     compute $MSE_n$                                        ▷ cf. Eq. (5.37)
     $C \leftarrow MSE_n$
     update parameters: $\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \alpha \frac{\partial C}{\partial \boldsymbol{\Theta}}$            ▷ Adam or L-BFGS
  end loop
  for all epochs do
     run loop with Adam optimizer
  end for
  repeat
     run loop with L-BFGS optimizer
  until convergence or max. iterations
```

network $\boldsymbol{u}_{NN}(x; \boldsymbol{\Theta})$ with the trained parameters $\boldsymbol{\Theta}$ is used to predict the temperature at time t^{n+1} for a given input x.

The results in this study were generated with the set of parameters denoted in Table 5.2. Additional parameters of the Adam and L-BFGS optimizers default to the values chosen by Raissi [Rai20]. The current implementation is based on Tensorflow version 1.15 and Python version 3.7. The computations were executed on a laptop computer with an Intel Core i7-7700HQ @ 2.80GHz CPU and a NVIDIA GeForce GTX 1050 graphics card.

5.3.3 Results

Outgoing from the solution at time $t^n = 0.05$, the aim was to predict the temperature distribution at time $t^{n+1} = 0.3$ for the problem outlined in Section 5.3.1. With the resulting time step size $\Delta t = 0.25$ the solution is resolved over half of the total domain $t = [0, 0.5]$. The

no. of stages q:	500
no. of hidden layers:	4
no. hidden neurons:	50 (per layer)
hidden activation function:	Tanh
output activation function:	Sinh
no. of initial data points N_n:	200
no. of *epochs*:	10 000 (Adam optimizer)
learning rate α:	0.001 (Adam optimizer)
max iterations:	50 000 (L-BFGS optimizer)
tolerance for convergence:	machine precision ϵ (L-BFGS optimizer)

Table 5.2: Parameters for discrete-time inference.

first solution attempt using the algorithm explained in Section 5.3.2 is depicted in Fig. 5.13. The snapshot at time $t = 0.3$ reveals that the network by construction satisfied the boundary conditions, but failed to predict the underlying solution. Further, the L-BFGS optimization stopped shortly after the 10 000 epochs of the Adam optimizer, indicating that no desirable minimum was found.

In a second step, the non-linear output activation proposed in Section 5.2.6 was implemented and tested. The repeated training resulted in a satisfactory solution that is presented in Fig. 5.14. The network was able to accurately predict the temperature distribution at time $t^{n+1} = 0.3$ solely based on 200 randomly distributed training points representing the known solution at time $t = 0.05$ and the prescribed boundary conditions. The error according to the relative error norm from Eq. (5.38) was measured at $5.498\,873 \times 10^{-3}$.

5.4 Further Studies

5.4.1 Weight Initialization

This short study aimed to expose the influence of the randomized weight initialization on the training outcome. Precisely, the two popular weight initialization schemes proposed by Glorot and Bengio are compared [GB10]. The physics-informed neural network was trained to predict the solution of the non-dimensionalized example from Section 5.2.4. In addition, the adaptive loss function from Section 5.2.5 was employed. During the repeated execution of the algorithm, the network architecture and the hyperparameters according to Table 5.1 remained unchanged. For each run the internal random generator generates a new set of random numbers for the weight initialization. It should be noted that the selection of training points is also randomized, and thus, adds further uncertainty to the results.

Having that in mind, Table 5.3 presents the relative errors of u and ϕ for 10 runs with a weight initialization based on a Glorot normal distribution [GB10]. The results for 10 repetitions with a Glorot uniform weight initialization are shown in Table 5.4 [GB10]. The obtained results indicate that both schemes achieve a comparable performance since no large difference between the respective errors of u and ϕ occurred.

The computations were executed on Google colab that automatically assigns a virtual ma-

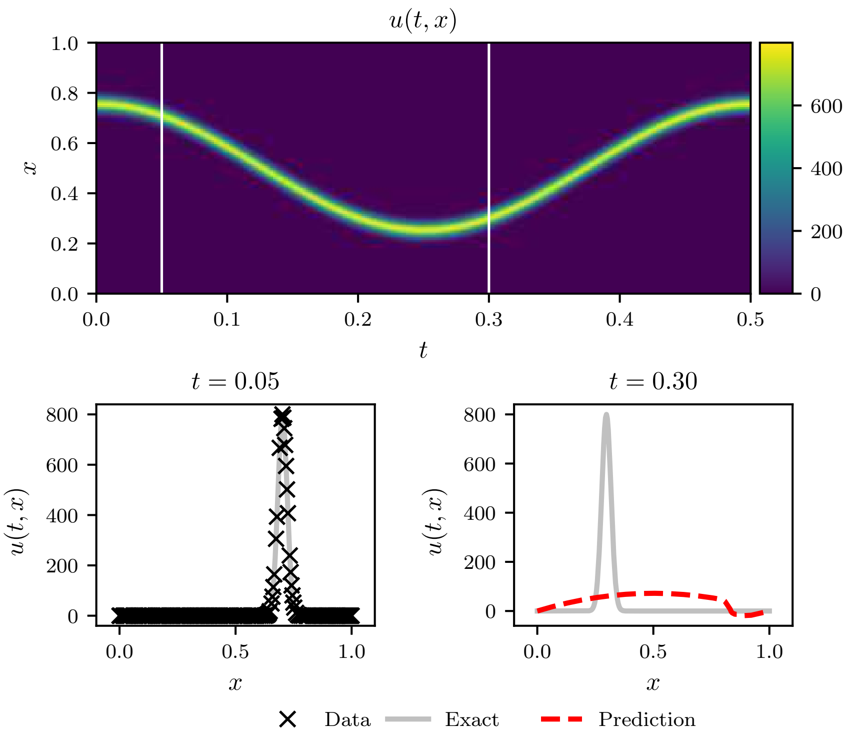


Figure 5.13: Discrete solution without modifications. Top: Manufactured solution u and time snapshots t^n and t^{n+1} (white lines). Bottom: Training data at t^n and prediction at t^{n+1}.

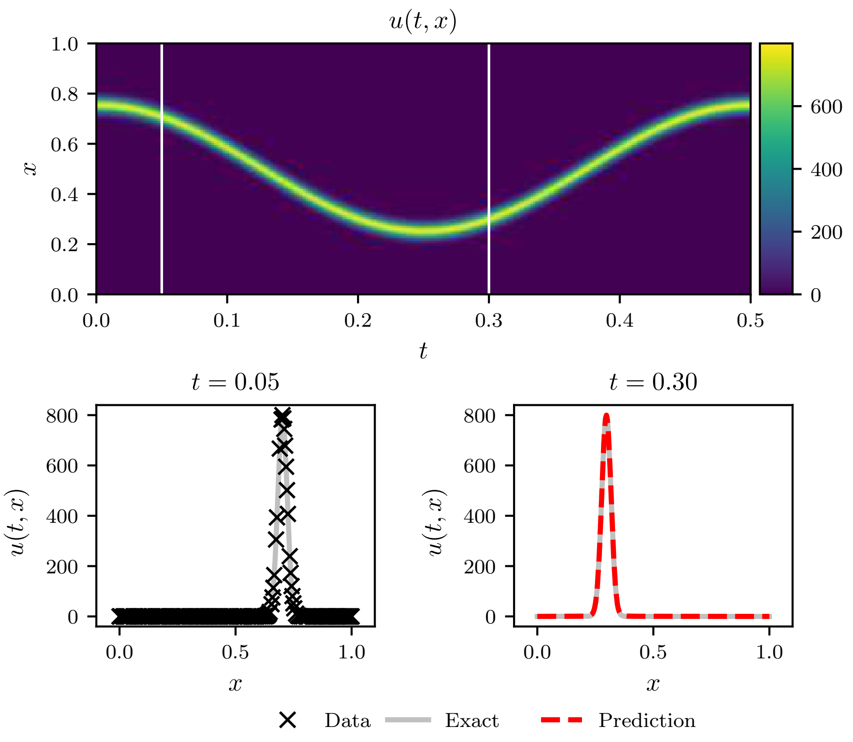


Figure 5.14: Discrete solution with Sinh output activation. Top: Manufactured solution and time snapshots t^n and t^{n+1} (white lines). Bottom: Training data at t^n and prediction at t^{n+1}.

Iterations	Time $[s]$	Error u	Error ϕ
1.53e+04	4.24e+02	2.78e-02	1.52e-02
1.32e+04	3.58e+02	1.60e-02	1.62e-02
1.29e+04	3.52e+02	2.36e-02	1.41e-02
1.31e+04	3.64e+02	1.67e-02	1.18e-02
1.33e+04	3.78e+02	1.74e-02	1.45e-02
1.31e+04	3.71e+02	2.99e-02	1.37e-02
1.18e+04	3.31e+02	2.89e-02	2.43e-02
1.33e+04	3.88e+02	1.76e-02	1.58e-02
1.40e+04	4.24e+02	2.02e-02	1.63e-02
1.34e+04	3.96e+02	1.97e-02	1.29e-02

Table 5.3: Glorot nornmal initialization with Tanh activation.

Iterations	Time $[s]$	Error u	Error ϕ
1.30e+04	4.04e+02	1.59e-02	1.24e-02
1.34e+04	4.32e+02	1.76e-02	1.47e-02
1.30e+04	4.10e+02	1.89e-02	1.87e-02
1.31e+04	4.20e+02	2.12e-02	1.17e-02
1.29e+04	4.19e+02	2.76e-02	1.98e-02
1.42e+04	4.98e+02	3.32e-02	1.90e-02
1.41e+04	4.92e+02	2.98e-02	1.53e-02
1.24e+04	4.04e+02	1.75e-02	1.23e-02
1.48e+04	5.53e+02	2.59e-02	1.71e-02
1.31e+04	4.53e+02	1.25e-02	1.14e-02

Table 5.4: Glorot uniform initialization with Tanh activation.

chine to the user [Goo20]. Depending on available resources, the computations are executed on different hardware, and therefore, the execution times are subject to variations. This experiment was conducted on a NVIDIA Tesla K80 GPU.

5.4.2 Alternative Activation Function

Similarly, a study investigating the performance of an alternative activation function was conducted. Instead of a regular Tanh activation, the sine was chosen as the activation function for the hidden units. Apart from that, the experiment was conducted under the same conditions as the previous study.

The results achieved with sine activations and Glorot uniform weight initialization are presented Table 5.5. Comparing the results in Table 5.5 with the ones in Table 5.3 from the previous study shows, that the sine generally performed worse than the Tanh activation. In particular, the errors on u and ϕ are higher and the algorithm needs additional iterations to converge.

Iterations	Time $[s]$	Error u	Error ϕ
3.60e+04	1.97e+03	1.93e-01	1.06e-01
3.01e+04	1.59e+03	1.39e-01	6.44e-02
4.15e+04	2.41e+03	1.87e-01	1.14e-01
3.53e+04	2.02e+03	3.81e-02	1.77e-02
2.54e+04	1.35e+03	9.50e-03	6.54e-03
2.80e+04	1.57e+03	3.20e-02	1.55e-02
2.68e+04	1.48e+03	1.19e-02	1.21e-02
1.00e+04	2.85e+02	1.46e+00	1.17e+00
3.35e+04	1.98e+03	1.11e-01	5.34e-02
2.38e+04	1.30e+03	2.03e-02	1.12e-02

Table 5.5: Glorot nornmal initialization with sine activation.

5.5 Discussion

The previous sections dealt with the implementation of physics-informed neural networks predicting continuous and discrete solutions to a one-dimensional heat transfer example. The implementation process revealed several hurdles that are, most probably, not unique to the domain of thermal analysis. Concurrently, tailored solutions for the encountered problems were proposed. Their general relevance and impact on further investigations is discussed in the following paragraphs.

The first solution attempts disclosed that without further modification, the algorithm was unable to produce any viable results for the investigated example. It was found, that an imbalanced cost function prevented convergence of the optimization problem. Introducing weighting factors allowed to account for the influence of the individual loss terms. To avoid manual tuning of the new hyperparameters, an alternative cost function was proposed in Section 5.2.5. The idea is to scale the loss terms relative to the absolute maximum of the expected target values. Even though the scaling scheme yielded satisfactory results, it leaves room for further improvement. This should be considered as a starting point for future investigations of adaptive cost functions. An interesting alternative could be an approach that automatically learns the weighting factors [KGC18].

For deep-learning-based regression tasks, it is sometimes recommended to scale the target values to ensure stability and to speed up the training [Bro19]. In the present case, this is somewhat similar to the partial nondimensionalization applied in Section 5.2.4. Here, the scaling of the problem resulted in a hidden solution with target values around unity. In combination with the adaptive cost function, the first accurate results were obtained. However, as stated in Section 5.2.6, it is not always possible to determine an appropriate scaling factor since any knowledge about the hidden solution is solely based on sparse boundary data. To circumvent the need for a manual scaling of the underlying problem, a simple yet effective enhancement was proposed. Replacing the linear activation function of the output neuron with the hyperbolic sine enabled faster and more accurate predictions for the un-scaled example. The effect can be traced back to the increased expressive power of the output layer compared to a conventional linear activation. As a matter of fact, the network prediction is a linear combination of the outputs from the previous layers. Furthermore, the Tanh activation function of the hidden neurons only returns values between -1 and 1 (cf. Fig. 3.3). In return, larger weights in the final layer are necessary to predict a greater output value. The magnitude of the initial weights depends on the initialization procedure, but often ranges around 0 [GB10]. On top of that, the weight update depends only on the chosen learning rate, since the derivative of a linear function is constant (cf. Section 3.3). Thus, faster training of the output layer would require a large learning rate that often causes unstable training [Cho18]. The Tanh activation seems to overcome this issue due to its exponential growth and non-constant derivative. Since the hyperbolic sine is almost linear in the interval between -1 and 1, it can be equally suitable for predictions in smaller ranges. A rudimentary test on the Burgers' equation example from the paper by Raissi et al. exhibited comparable results. Nevertheless, more detailed investigations are necessary to confirm the effectiveness of this newly proposed output activation in the context of physics-informed neural networks.

In general, activation function constitute another potential research subject. A recent publication provided examples, where a periodic activation was beneficial to problems governed by partial differential equations [SMB$^+$20]. This inspired a small study to compare the performance of the sine and the original hyperbolic tangent as activation functions. It was noted that the sine performed slightly worse for the investigated example in terms of prediction accuracy and the number of iterations (cf. Section 5.4.2). Nonetheless, more fundamental investigations are necessary and left for future research. Another paper related to physics-informed neural networks suggested the use of a squared ReLU activation [GACR19]. Generally, every nonlinear function could act as an activation function. However, to account for most differential operators found in physical problems, it should have a non-zero second derivative. For this reason, simple ReLU activations are not considered in the context of physics-informed surrogate models.

The investigation of the discrete-time method also demonstrated the advantage of the modified output neuron. In combination with a strong enforcement of the boundary condition, the physics-informed neural network was able to predict an accurate solution for the given example. The idea to strongly enforce the boundary conditions of the partial differential equation dates back to the early paper of Lagaris et al. [LLF98]. They provided a formalized description for various types of boundary conditions. One big advantage of strong enforcement is the simplified cost function. As claimed in several other publications, this promises a better-posed optimization problem [SAG$^+$19, GACR19, NM19]. However, in cases of irregular boundaries, the derivation of a suitable solution function is often infeasible. Thereby, the general applicability of physics-informed neural networks gets affected.

Furthermore, this study unveiled that the algorithm's convergence is subject to various factors. Next to the choice of cost function or hyperparameters, the randomized weight initialization and training point selection had a notable impact on the prediction outcome. The initial weights determine were the optimization algorithm begins its search for a minimum. Varying this starting point and the training data adds a noticeable degree of uncertainty to the solution as the study from Section 5.4.1 indicates. A second factor limiting the reproducibility of results is owed to the training on GPUs. Usually, determinism is only achieved for a specific hardware-software constellation [Ria20]. One way to address these issues is to repeatedly execute the algorithm and to average over the obtained results. However, this would increase the computational effort even further. Without an explicit comparison, it is safe to say that the training process of a physics-informed neural network is significantly slower than computing a solution with a conventional method. All in all, this implies that the results conducted in this work underlie uncertainties and therefore should be treated with caution.

Even though a variety of aspects were studied, many areas of interest remain untouched. First of all, Raissi et al. claimed that the physics-informed neural networks are not prone to overfitting since the physical constraints impose a regularizing effect [RPK19]. Within the scope of this work, no contrary indications were found. However, a proof for this hypothesis is still pending. Further, the scaling properties of the continuous method are yet to be determined [RPK19]. The number of required collocation points grows exponentially in higher dimensions. Due to increased memory demand, this allows only the application of mini-batch gradient descent algorithms. To prepare further studies, the extended code accompanying this treatise entails an implementation of a mini-batch optimization approach. Apart from that, implementations of physics-informed neural networks based on a variational formulation appeared in multiple recent publications (cf. Section 4.3). By adequately integrating the partial differential equation, the order of the differential operator can be reduced, which supposedly simplifies the optimization problem. Therefore, a more detailed investigation of these approaches should be considered.

Chapter 6

Conclusion and Outlook

6.1 Conclusion

The preceding chapters explored the interface between data-driven algorithms and computational mechanics. This was accomplished by introducing the basic concepts of machine learning and reviewing current literature on learning-based solutions to physics and engineering problems. Some of the presented works demonstrated that machine learning algorithms could be used to enhance existing numerical methods [OY17]. However, the improvements usually failed to justify the increased computational effort [BSHHB19]. Other publications have replaced expensive computations with neural networks as surrogates for mechanical systems [LLMS18, WBT19]. To ensure accurate predictions, the networks required vast amounts of data generated by conventional methods. An alternative approach incorporating domain knowledge into the surrogate model circumvented the necessity of costly data acquisition [RPK19]. Promising results inspired the work presented here to pursue this line of study. In particular, the operating principles of physics-informed neural networks were elaborated and investigated with a transient heat transfer problem. Building upon an existing code base [Rai20], the physics-enriched learning algorithm was adapted to continuously predict the latent temperature distribution. The obstacles that occurred during the implementation process were documented and possible workarounds were proposed. For example, the different magnitudes of the loss terms required the introduction of penalty factors. To avoid an empirical hyperparameter search, a scheme was proposed accounting for the scales of the individual cost function terms. Another observation revealed that training of the network worked best when the prediction targets stayed close to unity. However, re-scaling the output is only possible when additional knowledge about the hidden solution is available. With the idea to extend the representational power of the network, the linear activation in the output layer was replaced by the hyperbolic sine function. This enabled faster learning in case the network needed to predict values beyond unity. However, it remains subject to further investigations in order to validate the effectiveness of the proposed idea. Subsequently, another solution strategy based on physics-informed neural networks was introduced. The applied method combines the physics-enriched prediction of the network with an implicit time-stepping scheme. According to the authors, this approach allows for the accurate resolution of almost the entire domain in a single time-step [RPK19]. Due to computational limitations, implicit time schemes are usually employed with a small number of stages per time step. Here, the network approximated intermediate solutions at relatively low extra cost and thus, allowed larger time steps. For the example of non-linear heat transfer, the described method provided accurate predictions when combined with strong enforcement of boundary conditions and the non-linear output activation.

Even though the methods presented offer advantages such as mesh-free solutions, they are far from being able to compete with conventional methods in terms of computational efficiency and reliability. The implementation process revealed a series of drawbacks mainly due to the very nature of deep learning algorithms. Specifically, the high training costs and the sensitivity toward a multitude of hyperparameters should be taken into consideration. Among others, the weight initialization, the choice of activation function, and the learning rates had a non-negligible impact on the results. Most of these issues are not unique to the applications presented. The problems are rather general to the field of deep learning and are still under intensive investigation. The uncertain convergence properties of gradient-based optimization algorithms are a particularly major concern. Since neural networks impose a highly non-convex optimization landscape, the procedures are not guaranteed to find a satisfying solution.

However, should these drawbacks be eliminated or at least diminished, data-driven approaches could become a valuable addition to the scientific computing toolbox. In specific cases where conventional numerical methods would exceed the computational limitations, physics-informed learning algorithms could provide alternative solutions. As the rising research interest indicates [FDC20, BNK20], deep learning and its interface with other scientific fields is developing at an extremely rapid pace. Thus, the presented findings could be outdated sooner rather than later. Nevertheless, by discussing and identifying the assets and drawbacks of data-driven computational mechanics, this work has established a basic foundation for further research on the subject.

6.2 Outlook

Even though first insights regarding the application of physics-informed neural networks were developed, many relevant questions remain unanswered. Therefore, this outlook suggests interesting research directions. First of all, this work only demonstrated the learning-based algorithm on transient heat problems in one spatial dimension. To get closer to the adoption in real-world applications it is crucial that the learning-based approaches scale up to problems in higher dimensions. Further, it was suggested to use physics-informed neural networks to identify coefficients in the governing equations by observing random data [RPK19]. In cases where the material coefficients are usually determined by experimentation, this approach could offer a worthy alternative. Recent publications introduce data-driven methods to other engineering domains. An example from solid mechanics employed an enhanced physics-informed neural network in combination with a phase field model for crack propagation [GACR19]. To speed up the iterative computations a transfer learning approach was proposed. This idea could be picked up to apply predictive solutions in the layered processes of additive manufacturing or in biomechanical problems dealing with brittle fracture [HKYR20]. Next to transfer learning data sciences provide a wide range of improvements for neural networks. Those include batch normalization, dropout or modified architectures, to name a few. Some of the presented approaches already applied a network variation making use of residual connections between non-adjacent layers. Another archetype could possibly provide improved predictions of transient problems. Long-short term memory neural networks were particularly designed to predict temporal evolution.

Given the fact, that deep learning in general is a young area of research and subject to constant transformations, new discoveries are expected to elevate applications in all scientific disciplines. For instance, the advent of quantum computer could revolutionize the data-driven computing by providing significant speed-ups of the training process. Some go even so far and predict that learning-based algorithms will aid the discovery of fundamental physical concepts.

Even though initial insights regarding the application of physics-informed neural networks were developed, many relevant questions remain unanswered. Therefore, this outlook suggests interesting directions for future research. First of all, this work has only demonstrated the learning-based algorithm on transient heat problems in one spatial dimension. To approach the adoption of real-world applications, the learning-based approaches must scale up to problems in higher dimensions. Further, physics-informed neural networks were used to identify coefficients in the governing equations by observing random data [RPK19]. In cases where the material coefficients are usually determined by experimentation, this approach could offer a worthy alternative. Recent publications have introduced data-driven methods to other engineering domains. An example from solid mechanics employed an enhanced physics-informed neural network in combination with a phase-field model for crack propagation [GACR19]. To speed up iterative computations, a transfer learning approach was proposed. This idea could be adopted to apply predictive solutions in the layered processes of additive manufacturing or biomechanical problems dealing with brittle fracture [HKYR20]. Data sciences provide a wide range of techniques to further improve the performance of neural networks. Those include batch normalization, dropout, or modified architectures, to name a few. Some of the presented approaches already applied a network variation making use of residual connections between non-adjacent layers [EY17, NM19]. Another archetype could provide improved predictions of transient problems as a first application suggests [WBT19]. Long-short term memory neural networks were particularly designed to predict temporal evolution.

Given the fact that deep learning is, in general, a young area of research and subject to constant transformations, discoveries are expected to advance applications across scientific disciplines. For instance, the advent of quantum computers could revolutionize neural network training by providing significant optimization speed-ups [CCC^{+}19]. Some have even gone so far as to predict that learning-based algorithms will aid the discovery of fundamental physical concepts.

Index

Appendix A

Data Stick

The attached data stick contains the following materials:

- Python script for the sine approximation example used to produce the presented results,
- MATLAB code used for generating the manufactured solutions,
- Python scripts for continuous and discrete-time physics-informed neural networks used to produce the presented results,
- the source code of this document in addition with all embedded graphics.

List of Figures

List of Tables

Bibliography

[AAB$^+$15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ABP$^+$19] Yaser Afshar, Saakaar Bhatnagar, Shaowu Pan, Karthik Duraisamy, and Shailendra Kaushik. Prediction of Aerodynamic Flow Fields Using Convolutional Neural Networks. *Comput Mech*, 64(2):525–545, August 2019. arXiv: 1905.13166.

[AJZS18] Jeff Adie, Yang Juntao, Xuemeng Zhang, and Simon See. Deep Learning for Computational Science and Engineering. 2018.

[All07] Grégoire Allaire. *Numerical analysis and optimization: an introduction to mathematical modelling and numerical simulation.* Numerical mathematics and scientific computation. Oxford University Press, Oxford ; New York, 2007. OCLC: ocm82671667.

[AMMIL12] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data.* AMLBook, S.l., 2012.

[AP03] Larry C. Andrews and Ronald L. Phillips. *Mathematical Techniques for Engineers and Scientists.* SPIE Press, 2003. Google-Books-ID: MwrD-fvrQyWYC.

[BEJ19] Christian Beck, Weinan E, and Arnulf Jentzen. Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations. *J Nonlinear Sci*, 29(4):1563–1619, August 2019. arXiv: 1709.05963.

[BG09] Pavel B. Bochev and Max D. Gunzburger. *Least-squares finite element methods.* Number 166 in Applied mathematical sciences. Springer, New York, 2009. OCLC: ocn318869130.

[BNK20] Steven Brunton, Bernd Noack, and Petros Koumoutsakos. Machine Learning for Fluid Mechanics. *Annu. Rev. Fluid Mech.*, 52(1):477–508, January 2020. arXiv: 1905.11075.

[Bro19] Jason Brownlee. How to use Data Scaling Improve Deep Learning Model Stability and Performance. https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/, February 2019. Blog post.

[BSHHB19] Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P. Brenner. Learning data-driven discretizations for partial differential equations. *Proc Natl Acad Sci USA*, 116(31):15344–15349, July 2019.

[Bur19] Andriy Burkov. *The Hundred-Page Machine Learning Book.* Andriy Burkov, January 2019.

[CCC$^+$19] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. Machine learning and the physical sciences. *Rev. Mod. Phys.*, 91(4):045002, December 2019. arXiv: 1903.10563.

[Cho18] François Chollet. *Deep learning with Python.* Manning Publications Co, Shelter Island, New York, 2018. OCLC: ocn982650571.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control Signal Systems*, 2(4):303–314, December 1989.

[DBDJH14] Howard B. Demuth, Mark H. Beale, Orlando De Jess, and Martin T. Hagan. *Neural Network Design.* Martin Hagan, Stillwater, OK, USA, 2nd edition, 2014.

[EY17] Weinan E and Bing Yu. The Deep Ritz method: A deep learning-based numerical algorithm for solving variational problems. *arXiv:1710.00211 [cs, stat]*, September 2017. arXiv: 1710.00211.

[FDC20] Michael Frank, Dimitris Drikakis, and Vassilis Charissis. Machine-Learning Methods for Computational Science and Engineering. *Computation*, 8(1):15, March 2020.

[GACR19] Somdatta Goswami, Cosmin Anitescu, Souvik Chakraborty, and Timon Rabczuk. Transfer learning enhanced physics informed neural network for phase-field modeling of fracture. *arXiv:1907.02531 [cs, stat]*, July 2019. arXiv: 1907.02531.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, March 2010. ISSN: 1938-7228 Section: Machine Learning.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

[GLI16] Xiaoxiao Guo, Wei Li, and Francesco Iorio. Convolutional Neural Networks for Steady Flow Approximation. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 481–490, San Francisco California USA, August 2016. ACM.

[Goo20] Google. Colaboratory. `https://research.google.com/colaboratory/faq.html`, 2020. Software.

[GVR+19] Paul Garnier, Jonathan Viquerat, Jean Rabault, Aurélien Larcher, Alexander Kuhnle, and Elie Hachem. A review on Deep Reinforcement Learning for Fluid Mechanics. *arXiv:1908.04127 [physics]*, August 2019. arXiv: 1908.04127.

[HKKC20] Norbert Huber, Surya R Kalidindi, Benjamin Klusemann, and Christian Johannes Cyron, editors. *Machine Learning and Data Mining in Materials Science*. Frontiers Research Topics. Frontiers Media SA, 2020.

[HKYR20] L. Hug, S. Kollmannsberger, Z. Yosibash, and E. Rank. A 3D benchmark problem for crack propagation in brittle fracture. *Computer Methods in Applied Mechanics and Engineering*, 364:112905, June 2020.

[HMP17] Matthew Hirn, Stéphane Mallat, and Nicolas Poilvert. Wavelet Scattering Regression of Quantum Chemical Energies. *Multiscale Model. Simul.*, 15(2):827–863, January 2017. arXiv: 1605.04654.

[Hoc91] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015. arXiv: 1512.03385.

[Ise08] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge University Press, November 2008. Google-Books-ID: 3acgAwAAQBAJ.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, January 2017. arXiv: 1412.6980.

[KCRA19] Stefan Kollmannsberger, Massimo Carraturo, Alessandro Reali, and Ferdinando Auricchio. Accurate Prediction of Melt Pool Shapes in Laser Powder Bed Fusion by the Non-Linear Temperature Equation Including Phase Changes: Model validity: isotropic versus anisotropic conductivity to capture AM Benchmark Test AMB2018-02. *Integr Mater Manuf Innov*, 8(2):167–177, June 2019.

[KGC18] Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics. *arXiv:1705.07115 [cs]*, April 2018. arXiv: 1705.07115.

[KO16] Trenton Kirchdoerfer and Michael Ortiz. Data-driven computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 304:81–101, June 2016. arXiv: 1510.04232.

[KÖC+18] S. Kollmannsberger, A. Özcan, M. Carraturo, N. Zander, and E. Rank. A hierarchical computational model for moving thermal loads and phase changes with applications to selective laser melting. *Computers & Mathematics with Applications*, 75(5):1483–1497, March 2018.

[Kon18] Risi Kondor. N-body Networks: a Covariant Hierarchical Neural Network Architecture for Learning Atomic Potentials. *arXiv:1803.01588 [cs]*, March 2018. arXiv: 1803.01588.

[KV95] Dr D. J. Korteweg and Dr G. de Vries. XLI. On the change of form of long waves advancing in a rectangular canal, and on a new type of long stationary waves. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 39(240):422–443, May 1895. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/14786449508620739.

[KZK19] E. Kharazmi, Z. Zhang, and G. E. Karniadakis. Variational Physics-Informed Neural Networks For Solving Partial Differential Equations. *arXiv:1912.00873 [physics, stat]*, November 2019. arXiv: 1912.00873.

[LBBH98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. Conference Name: Proceedings of the IEEE.

[LLF98] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Trans. Neural Netw.*, 9(5):987–1000, September 1998.

[LLMS18] Liang Liang, Minliang Liu, Caitlin Martin, and Wei Sun. A deep learning approach to estimate stress distribution: a fast and accurate surrogate of finite-element analysis. *J. R. Soc. Interface*, 15(138):20170844, January 2018.

[LLS08] B. Llanas, S. Lantarón, and F. J. Sáinz. Constructive Approximation of Discontinuous Functions by Neural Networks. *Neural Process Lett*, 27(3):209–226, June 2008.

[LN89] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, August 1989.

[LP16] Hans Petter Langtangen and Geir K. Pedersen. *Scaling of Differential Equations*. Springer International Publishing, Cham, 2016.

[LRVL+12] Philippe Lambin, Emmanuel Rios-Velazquez, Ralph Leijenaar, Sara Carvalho, Ruud G.P.M. van Stiphout, Patrick Granton, Catharina M.L. Zegers, Robert Gillies, Ronald Boellard, André Dekker, and Hugo J.W.L. Aerts. Radiomics: Extracting more information from medical images using advanced feature analysis. *European Journal of Cancer*, 48(4):441–446, March 2012.

[Mal16] Stéphane Mallat. Understanding deep convolutional networks. *Phil. Trans. R. Soc. A*, 374(2065):20150203, April 2016.

[Mar17] Florian Marquardt. Visualize the output of a multilayer network. http://www.thp2.nat.uni-erlangen.de/images/0/0e/MultiLayerNet_SimpleExample.py, 2017. Python code.

[Mar19] Florian Marquardt. Machine Learning for Physicists. Lecture, 2019.

[MBW+19] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G. R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to Machine Learning for physicists. *Physics Reports*, 810:1–124, May 2019. arXiv: 1803.08823.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill series in computer science. McGraw-Hill, New York, 1997.

[MK02] Michele Milano and Petros Koumoutsakos. Neural Network Modeling for Near Wall Turbulent Flow. *Journal of Computational Physics*, 182(1):1–26, October 2002.

[MLP16] Hrushikesh Mhaskar, Qianli Liao, and Tomaso Poggio. Learning Functions: When Is Deep Better Than Shallow. *arXiv:1603.00988 [cs]*, May 2016. arXiv: 1603.00988.

[MMRMMS+17] F. Martínez-Martínez, M.J. Rupérez-Moreno, M. Martínez-Sober, J.A. Solves-Llorens, D. Lorente, A.J. Serrano-López, S. Martínez-Sanchis, C. Monserrat, and J.D. Martín-Guerrero. A finite element-based machine learning approach for modeling the mechanical behavior of the breast tissues under compression in real-time. *Computers in Biology and Medicine*, 90:116–124, November 2017.

[Ng20] Andrew Ng. Machine Learning. https://www.coursera.org/learn/machine-learning?, 2020. Online course.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[NM19] Mohammad Amin Nabian and Hadi Meidani. A Deep Neural Network Surrogate for High-Dimensional Random Partial Differential Equations. *Probabilistic Engineering Mechanics*, 57:14–25, July 2019. arXiv: 1806.02957.

[Ola15] Christopher Olah. Understanding LSTM Networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, August 2015. Blog post.

[OY17] Atsuya Oishi and Genki Yagawa. Computational mechanics enhanced by deep learning. *Computer Methods in Applied Mechanics and Engineering*, 327:327–351, December 2017.

[PLK18] Guofei Pang, Lu Lu, and George Em Karniadakis. fPINNs: Fractional Physics-Informed Neural Networks. *arXiv:1811.08967 [physics]*, November 2018. arXiv: 1811.08967.

[PTA12] Richard H. Pletcher, John C. Tannehill, and Dale Anderson. *Computational Fluid Mechanics and Heat Transfer, Third Edition*. CRC Press, August 2012. Google-Books-ID: Cv4IERczJ4oC.

[PU92] Dimitris C. Psichogios and Lyle H. Ungar. A hybrid neural network-first principles approach to process modeling. *AIChE J.*, 38(10):1499–1511, October 1992.

[Rai20] Maziar Raissi. maziarraissi/PINNs. https://github.com/maziarraissi/PINNs, July 2020. original-date: 2018-01-21T04:04:32Z.

[RBPK17] Samuel H. Rudy, Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Data-driven discovery of partial differential equations. *Sci. Adv.*, 3(4):e1602614, April 2017.

[Ria20] Duncan Riach. NVIDIA/framework-determinism. https://github.com/NVIDIA/framework-determinism, July 2020. original-date: 2019-03-18T20:51:45Z.

[RK18] Maziar Raissi and George Em Karniadakis. Hidden Physics Models: Machine Learning of Nonlinear Partial Differential Equations. *Journal of Computational Physics*, 357:125–141, March 2018. arXiv: 1708.00588.

[RMK93] R. Rico-Martinez and I.G. Kevrekidis. Continuous time modeling of nonlinear systems: a neural network-based approach. In *IEEE International Conference on Neural Networks*, pages 1522–1525, San Francisco, CA, USA, 1993. IEEE.

[RND10] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: a modern approach.* Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, 3rd ed edition, 2010.

[Roa02] Patrick J. Roache. Code Verification by the Method of Manufactured Solutions. *J. Fluids Eng*, 124(1):4–10, March 2002. Publisher: American Society of Mechanical Engineers Digital Collection.

[RPK17a] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Inferring solutions of differential equations using noisy multi-fidelity data. *Journal of Computational Physics*, 335:736–746, April 2017. arXiv: 1607.04805.

[RPK17b] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Machine learning of linear differential equations using Gaussian processes. *Journal of Computational Physics*, 348:683–693, November 2017.

[RPK17c] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Numerical Gaussian Processes for Time-dependent and Non-linear Partial Differential Equations. *arXiv:1703.10230 [cs, math, stat]*, March 2017. arXiv: 1703.10230.

[RPK17d] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. *arXiv:1711.10561 [cs, math, stat]*, November 2017. arXiv: 1711.10561.

[RPK17e] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations. *arXiv:1711.10566 [cs, math, stat]*, November 2017. arXiv: 1711.10566.

[RPK19] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, February 2019.

[RYK18] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. Hidden Fluid Mechanics: A Navier-Stokes Informed Deep Learning Framework for Assimilating Flow Visualization Data. *arXiv:1808.04327 [physics, stat]*, August 2018. arXiv: 1808.04327.

[SAG+19] Esteban Samaniego, Cosmin Anitescu, Somdatta Goswami, Vien Minh Nguyen-Thanh, Hongwei Guo, Khader Hamdia, Timon Rabczuk, and Xiaoying Zhuang. An Energy Approach to the Solution of Partial Differential Equations in Computational Mechanics via Machine Learning: Concepts, Implementation and Applications. *arXiv:1908.10407 [cs, math, stat]*, September 2019. arXiv: 1908.10407.

[SMB+20] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit Neural Representations with Periodic Activation Functions. *arXiv:2006.09661 [cs, eess]*, June 2020. arXiv: 2006.09661.

[SS18] Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, December 2018. arXiv: 1708.07469.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, October 2017.

[Ste87] Michael Stein. Large Sample Properties of Simulations Using Latin Hypercube Sampling. *Technometrics*, 29(2):143–151, May 1987.

[Vel20] Petar Veličković. PetarV-/TikZ. `https://github.com/PetarV-/TikZ`, 2020. Library Catalog: github.com.

[WBT19] Steffen Wiewel, Moritz Becher, and Nils Thuerey. Latent-space Physics: Towards Learning the Temporal Evolution of Fluid Flow. *arXiv:1802.10123 [cs]*, March 2019. arXiv: 1802.10123.

[WM03] D.Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, December 2003.