Lehrstuhl für Computergestützte Modellierung und Simulation
TUM School of Engineering and Design
Technical University of Munich

Deep Learning-Based Surrogate Models for Linear Elasticity

Scientific work to obtain the degree

Master of Science (M.Sc.)

at the TUM School of Engineering and Design
of the Technical University of Munich.

Supervised by PD Dr.-Ing. habil. Stefan Kollmannsberger
Leon Herrmann, M.Sc.
Lehrstuhl für Computergestützte Modellierung und Simulation

Dr.-Ing. Davide D’Angella
Hyperganic Group

Submitted by Torsten Kai Schmid [redacted]
[redacted]
[redacted]
[redacted]

Submitted on 31. March 2023

Abstract

Design optimization poses significant challenges due to the substantial expensive and time-consuming characteristic of simulations. To counteract this issue, deep learning-based surrogate models have recently emerged as an effective solution. However, current research has primarily focused on applying these models to Computational Fluid Dynamics, with limited studies in the area of Linear Elasticity.
Unlike previous work, the investigated structures are 3D with notable variations in the simulation domains, resulting in significant changes to the learning domain of the network. The proposed Convolutional Neural Network is based on the U-Net architecture and is trained in a supervised manner to learn the displacement and stress results. The problem is treated as an image-to-image learning task, with the simulation domain encoded through a binary mask. The input images are associated with the spatial distribution of material parameters, the applied boundary conditions, and the force density distribution.
The accuracy of the model has been evaluated on datasets encompassing up to 800 hip implant simulations. The finding showed that the trained models exhibit inadequate generalization capabilities, indicating that more data is necessary for the investigated problem formulation. To improve the performance for the given task, alternative encoding strategies or learning approaches may be necessary.

Contents

Acronyms

AI	Artificial Intelligence
BC	Boundary Condition
BN	Batch Normalization
CFD	Computational Fluid Dynamics
CNN	Convolutional Neural Network
DBC	Dirichlet Boundary Condition
DoF	Degree of Freedom
FCM	Finite Cell Method
GPU	Graphics Processing Unit
ML	Machine Learning
NBC	Neumann Boundary Condition
NN	Neural Network
PCA	Principal Component Analysis
PDE	Partial Differential Equation
PINN	Physics-informed Neural Network
STL	Stereolithography

List of Symbols

α	Indicator function in the FCM
β_1	Exponential decay parameter of $\boldsymbol{m}$
β_2	Exponential decay parameter of $\boldsymbol{v}$
$\boldsymbol{\beta}^{\text{BN}}$	Learned shift of BN layer
$\boldsymbol{\epsilon}$	Strain tensor
$\boldsymbol{\gamma}^{\text{BN}}$	Learned scaling factor of BN layer
$\boldsymbol{\mu}^{X}$	Channel-wise mean vector of X_{ijk}^{masked}
$\boldsymbol{\mu}^{Y}$	Channel-wise mean vector of Y_{ijk}^{masked}
$\boldsymbol{\mu}^{\mu^{\tilde{Y}}}$	Channel-wise mean vector of $\boldsymbol{\mu}^{\tilde{Y}}$
$\boldsymbol{\mu}^{\sigma^{\tilde{Y}}}$	Channel-wise mean vector of $\boldsymbol{\sigma}^{\tilde{Y}}$
$\boldsymbol{\mu}^{\text{BN,b}}$	Channel-wise mean vector of batch for BN
$\boldsymbol{\mu}^{\text{BN,mov}}$	Moving average of channel-wise mean vector of BN
$\boldsymbol{\mu}^{\tilde{Y}}$	Simulation and channel-wise mean matrix of $\tilde{Y}_{ijk}^{\text{masked}}$
$\boldsymbol{\sigma}$	Stress tensor
$\boldsymbol{\sigma}^{X}$	Channel-wise standard deviaton vector of X_{ijk}^{masked}
$\boldsymbol{\sigma}^{Y}$	Channel-wise standard deviaton vector of Y_{ijk}^{masked}
$\boldsymbol{\sigma}^{\mu^{\tilde{Y}}}$	Channel-wise standard deviation vector of $\boldsymbol{\mu}^{\tilde{Y}}$
$\boldsymbol{\sigma}^{\sigma^{\tilde{Y}}}$	Channel-wise standard deviation vector of $\boldsymbol{\sigma}^{\tilde{Y}}$
$\boldsymbol{\sigma}^{\text{BN,b}}$	Channel-wise standard deviation vector of batch for BN
$\boldsymbol{\sigma}^{\text{BN,mov}}$	Moving average of channel-wise standard deviation vector of BN
$\boldsymbol{\sigma}^{\tilde{Y}}$	Channel-wise standard deviaton matrix of $\tilde{Y}_{ijk}^{\text{masked}}$
$\boldsymbol{\theta}$	Model parameters
$\boldsymbol{B}^{\text{conv}}$	Bias of convolution operation
$\boldsymbol{f}$	Force density
$\boldsymbol{m}$	Exponential moving average of the gradient
$\boldsymbol{r}$	Spatial position vector
$\boldsymbol{u}$	Displacement
$\boldsymbol{v}$	Exponential moving average of the squared gradient
$\boldsymbol{w}$	Simulation and channel dependent weights for $L_{2,\text{w}}$
$\boldsymbol{W}^{\text{conv}}$	Weights or filters of convolution operation

Continued on next page

$\boldsymbol{X}$	Input dataset
$\boldsymbol{x}$	NN input
$\boldsymbol{x}^{\text{BN,b}}$	Input batch to BN
$\boldsymbol{x}^{\text{BN}}$	Input to BN in evaluation mode
$\boldsymbol{x}^{\text{conv}}$	Input to convolution operation
$\boldsymbol{X}^{\text{train}}$	Input of training dataset
$\boldsymbol{X}^{\text{val}}$	Input of validation dataset
$\boldsymbol{x}_i$	Input data sample
$\boldsymbol{y}$	NN target
$\boldsymbol{Y}$	Target dataset
$\boldsymbol{y}^{\text{BN,b}}$	Output batch after BN
$\boldsymbol{y}^{\text{BN}}$	Output of BN in evaluation mode
$\boldsymbol{y}^{\text{conv}}$	Output of convolution operation
$\boldsymbol{Y}^{\text{train}}$	Target of training dataset
$\boldsymbol{Y}^{\text{val}}$	Target of validation dataset
$\boldsymbol{y}_i$	Target data sample
ϵ	Term to improve numerical stability of Adam
ϵ^{BN}	Term to improve numerical stability of BN
γ	Learning rate of the optimizer
Γ_{D}	Dirichlet boundary
Γ_{N}	Neumann boundary
$\hat{\boldsymbol{\sigma}}$	Predicted stress tensor
$\hat{\boldsymbol{u}}$	Predicted displacement
$\hat{\boldsymbol{y}}$	NN prediction of the target $\boldsymbol{y}$
$\hat{\boldsymbol{y}}_i$	Prediction for the target data sample $\boldsymbol{y}_i$
$\hat{\tilde{\boldsymbol{\mu}}}^{\tilde{Y}}$	Prediction of $\tilde{\boldsymbol{\mu}}^{\tilde{Y}}$
$\hat{\tilde{\boldsymbol{\sigma}}}^{\tilde{Y}}$	Prediction of $\tilde{\boldsymbol{\sigma}}^{\tilde{Y}}$
$\hat{\tilde{\boldsymbol{y}}}_i$	Model prediction for $\tilde{\boldsymbol{y}}_i$
$\hat{\tilde{\tilde{Y}}}_{ijklm}$	Prediction of $\tilde{\tilde{Y}}_{ijklm}$
$\hat{\tilde{Y}}^{\text{masked}}_{ijk}$	Prediction of $\tilde{Y}^{\text{masked}}_{ijk}$
λ	Weight decay parameter
μ^z	Mean of generic distribution z
ν	Poisson's ratio

Continued on next page

Ω_{e}	Extended simulation domain
Ω_{phy}	Physical simulation domain
$\overline{\boldsymbol{F}}$	Prescribed total force constraint
$\overline{\boldsymbol{t}}_{\Gamma_{\text{N}}}$	Prescribed traction on Neumann boundary
$\overline{\boldsymbol{u}}$	Prescribed volumetric displacement constraint
$\overline{\boldsymbol{u}}_{\Gamma_{\text{D}}}$	Prescribed displacement on Dirichlet boundary
σ^{z}	Standard deviation of generic distribution z
τ	Shift of the kernel function
DE	Displacement error as defined in Equation (6.1)
SEMAXVM	Error of the maximum von mises stress as defined in Equation (6.7)
$\tilde{\boldsymbol{\mu}}^{\tilde{Y}}$	Standardized version of $\boldsymbol{\mu}^{\tilde{Y}}$
$\tilde{\boldsymbol{\sigma}}^{\tilde{Y}}$	Standardized version of $\boldsymbol{\sigma}^{\tilde{Y}}$
$\tilde{\boldsymbol{x}}_i$	Scaled input data sample
$\tilde{\boldsymbol{y}}_i$	Scaled target data sample
$\tilde{\tilde{Y}}_{ijklm}$	Simulation and channel-wise standardized form of $\tilde{Y}_{ijklm}$
$\tilde{\tilde{Y}}^{\text{masked}}_{ijk}$	Simulation and channel-wise standardized form of $\tilde{Y}^{\text{masked}}_{ijk}$
$\tilde{X}_{ijklm}$	Channel-wise standardized form of X_{ijklm}
$\tilde{X}^{\text{masked}}_{ijk}$	Channel-wise standardized form of X^{masked}_{ijk}
$\tilde{Y}_{ijklm}$	Channel-wise standardized form of Y_{ijklm}
$\tilde{Y}^{\text{masked}}_{ijk}$	Channel-wise standardized form of Y^{masked}_{ijk}
$\tilde{z}$	Standardized generic distribution
$\widehat{\boldsymbol{m}}$	Bias-corrected version of $\boldsymbol{m}$
$\widehat{\boldsymbol{v}}$	Bias-corrected version of $\boldsymbol{v}$
a	Activation function
b	Batch size
C	Cost function
C_{base}	Number of channels after first convolution
C_{b}	Cost associated to batch
C_{in}	Number of input channels
$C^{\text{BN}}_{\text{in}}$	Number of input channels to BN layer
$C^{\text{conv}}_{\text{in}}$	Number of input channels to convolution operation
C_{mul}	Channel multiplier
C_{out}	Number of output channels

Continued on next page

$C_{\text{out}}^{\text{conv}}$	Number of output channels of convolution operation
C_{train}	Cost of training dataset
D	Image depth
d	Current model depth
$D_{\text{in}}^{\text{conv}}$	Input image depth of the convolution operation
D_{m}	Model depth
$D_{\text{out}}^{\text{conv}}$	Output image depth of the convolution operation
E	Young's modulus
H	Image height
h	Input to convolution
$H_{\text{in}}^{\text{conv}}$	Input image height of the convolution operation
$H_{\text{out}}^{\text{conv}}$	Output image height of the convolution operation
K	Kernel size
k	Convolution kernel
L	Loss function
L_1	Absolute loss function
L_2	Square loss function
$L_{2,\text{w}}$	Weighted square loss function
m^{BN}	Momentum of BN layer
n	Number of batches
N	Dataset size
N^{set}	Size of a subset of the dataset
n_{feat}	Number of features within one model prediction
n_{invox}	Number of voxels inside the physical domain Ω_{phy}
$N_{\text{in}}^{\text{conv}}$	Number of input data samples to convolution operation
n_{Layers}	Number of layers of the layered stochastic lattice
P	Padding
s	Output of convolution
S	Stride
t	Parameter of the functions involved in the convolution
W	Image width
$W_{\text{in}}^{\text{conv}}$	Input image width of the convolution operation
$W_{\text{out}}^{\text{conv}}$	Output image width of the convolution operation

Continued on next page

X_{ijklm}	Input dataset tensor
x^{A}_{ijklm}	Input to activation function
X^{masked}_{ijk}	Masked form of X_{ijklm}
Y_{ijklm}	Target dataset tensor
y^{A}_{ijklm}	Output of activation function
Y^{masked}_{ijk}	Masked form of Y_{ijklm}
z	Generic distribution
z^{score}	Z-score or value after standardization
ADE	Average displacement error as defined in Equation (6.2)
DEMAXU	Error of the maximum displacement as defined in Equation (6.3)

Chapter 1

Introduction

1.1 Motivation

Surrogate models are a mathematical tool used to replace expensive and time-consuming simulations or experiments that assess desired outcomes, such as an object's displacement field. Surrogate modelling is particularly useful for design optimization and sensitivity analysis, where a multitude of objective evaluations for various parameter combinations are required, which may be impractical for real-world cases. Surrogate models are a promising technique to alleviate the computational burden by using a data-driven approach to construct a replacement model that closely mimics the behavior of the simulation model while being computationally cheaper to evaluate.

Deep learning-based surrogate models have recently emerged as a result of the increasing influence of Artificial Intelligence (AI) and the development of more powerful deep learning methods and their broad accessibility through modern deep learning frameworks. Recent studies have reported promising results in using surrogate models based on Convolutional Neural Networks (CNNs) for predicting incompressible (THUEREY et al., 2020) and compressible flows (L. CHEN and THUEREY (2022); DURU et al. (2022)). Moreover, encouraging results have been obtained for the prediction of temperature fields for layout design of heat source components in systems engineering (X. CHEN et al., 2021) and heat conduction (PENG et al., 2020).

Modern deep learning frameworks provide an opportunity to leverage gradient-based optimization methods using the fully differentiable constructed surrogate models. Recent studies have shown the effectiveness of such methods for shape optimization of airfoils (L.-W. CHEN et al. (2020); MALLIK et al. (2022)). Furthermore, the fast evaluation of surrogate models also enables the use of population-based optimization algorithms (MESSNER, 2020). Overall, surrogate modeling, especially deep learning-based surrogate models, are powerful tools that can significantly reduce the computational burden of complex simulations and enable more efficient designs by accelerating optimization.

1.2 Objectives

The primary objective of this thesis is to investigate the implementation of advanced deep learning architectures for the development of surrogate models in the context of 3D Linear Elasticity. The majority of existing literature on deep learning-based surrogate models focuses on Computational Fluid Dynamics (CFD) or thermal problems, with a limited number of studies addressing the application of deep learning for Linear Elasticity. Linear elastic simulations are indispensable for the design of physical objects; however, they can be computationally expensive during the initial stages of design optimization when exploring an extensive range of designs. The proposed surrogate model has the potential to efficiently optimize geometries within a constrained design subspace, characterized by a geometric recipe based on select parameters. The presented geometries originate from a geometric recipe specifically tailored for hip implants, illustrating the potential use case for rapid design optimization of patient-specific hip implants. Furthermore, the surrogate model can function as an initial approximation for the solution of iterative solvers, such as the conjugate gradient, as examined in UM et al. (2020), resulting in accelerated convergence of the solver.

Unlike previous studies, this thesis focuses on the application of CNNs to 3D Linear Elasticity with a significant part of the bounding box-based images extending beyond the simulation domain. The geometric variations result in substantial changes of the simulation domain and hence to the network's learning domain. Additionally, the integration of variable Dirichlet Boundary Conditions (DBCs) as images increases the challenge, as they exert considerable influence on the simulation outcomes despite their limited spatial extent.

1.3 Outline

The content of the upcoming chapters can be summarized as follows:

- Chapter 2: Embeds deep learning into the broader field of AI and covers the properties and training of Neural Networks (NNs).
- Chapter 3: Describes the different components of CNNs and presents the basic structure of a state-of-the-art architecture for image-to-image tasks.
- Chapter 4: Outlines the strategy for generating data for the learning task, based on a parametrized geometry and subsequent simulation of the structure.
- Chapter 5: Showcases the preparation of data for the learning task and introduces different learning methodologies.
- Chapter 6: Demonstrates the obtained results, with a specific emphasis on error metrics.
- Chapter 7: Summarizes the main results and outlines possible future research directions.

Chapter 2

Basics of Neural Networks

This chapter gives a concise overview of the fundamental principles of NNs.

2.1 Neural Networks in the Context of Artificial Intelligence

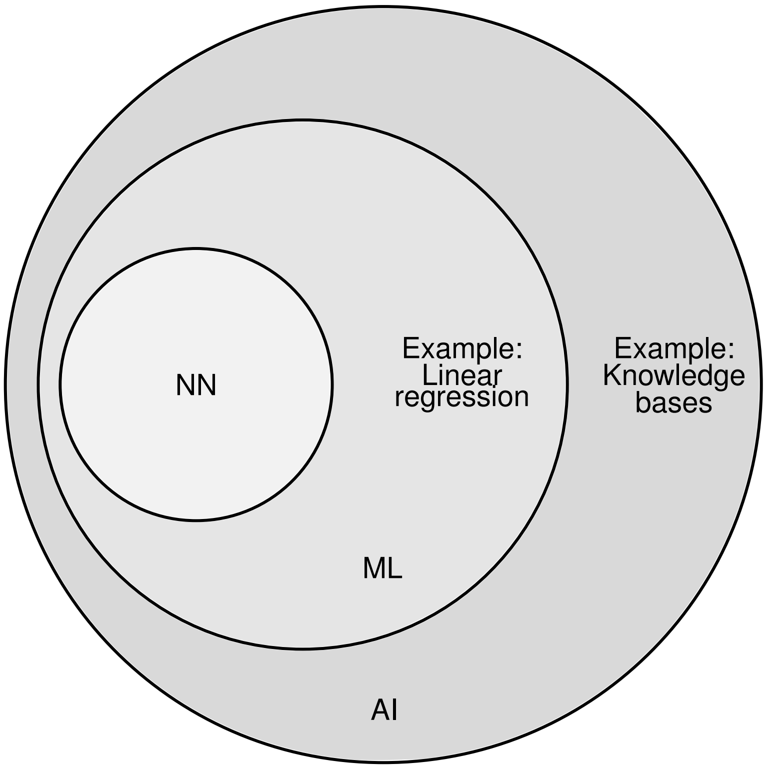


Figure 2.1: AI vs ML vs NN

This overview closely follows the work of GOODFELLOW et al. (2016) and is depicted in Figure 2.1. AI refers to the ability of machines, particularly computers, to display intelligence. Knowledge bases are one example of AI, which employ a set of rules to make informed decisions based on a certain input. While following a predefined set of rules to make decisions may be challenging for humans, it is a straightforward task for computers. However, tasks such as speech and image recognition are difficult to define using a set of rules. Therefore, Machine Learning (ML) methods that can automatically learn and identify underlying patterns in data without being explicitly programmed are essential. Linear regression is a basic example of ML, where the goal is to find a linear relationship between the model parameters and the output variables (KOLLMANNSBERGER et al., 2021). On the other hand, NNs are a quite sophisticated family of algorithms being capable of extracting complicated features by composing them out of many simpler parts.

Learning algorithms are used to solve specific problems, such as Classification and Regression. In Classification, the algorithm is trained to produce a function $f : \mathbb{R}^n \rightarrow \{1, \ldots, k\}$ (GOODFELLOW et al., 2016). This function assigns a category to an input described by a vector $\boldsymbol{x}$, e.g., it assigns a number to a handwritten digit. Semantic Segmentation takes this further by assigning a class label to each pixel within an image. In Regression problems, the outputs are numerical values, instead of a label.

To learn the underlying patterns in data, different types of learning techniques, such as Supervised Learning, Unsupervised Learning, and Reinforcement Learning exist. In Supervised Learning, the dataset is labeled, meaning that for some input data points $\boldsymbol{x}_i$ the true outputs $\boldsymbol{y}_i$ are known. In Unsupervised Learning, the dataset is not labeled, so the true output for a given input is unknown. Reinforcement Learning involves training an algorithm to make decisions based on interactions with its environment in order to maximize the average reward provided by its environment.

The problems studied in this thesis belong to the category of Regression and will be trained in a supervised manner using labeled datasets.

2.2 Properties of Neural Networks

The goal of NNs is to approximate a function $\boldsymbol{y} = f(\boldsymbol{x})$, where $\boldsymbol{x}$ represents the input and $\boldsymbol{y}$ the output (GOODFELLOW et al., 2016). The NN is a learned mapping that approximates this function as:

$$\hat{\boldsymbol{y}} = f_{\text{NN}}(\boldsymbol{x}; \boldsymbol{\theta}) \tag{2.1}$$

where $\boldsymbol{\theta}$ denotes the parameters of the NN and $\hat{\boldsymbol{y}}$ its prediction. According to the universal approximation theorem for functions, NNs are universal approximators, meaning that they can theoretically approximate any continuous function to an arbitrary degree of accuracy given enough parameters (HORNIK et al., 1989).

Typically, NNs are nested functions since they compose complex features out of a sequence of simpler functions. In the case of three functions, the NN can be represented as $f_{\text{NN}}(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. Various model architectures can be used, and one special family of models will be discussed in Chapter 3.

Usually, the function $\boldsymbol{y} = f(\boldsymbol{x})$ is not entirely known, but only some specific inputs $\boldsymbol{x}_i$ and their corresponding outputs $\boldsymbol{y}_i$ are available. These known inputs and outputs are collected in an input dataset $\boldsymbol{X}$ and a target dataset $\boldsymbol{Y}$, respectively, both of the size N. The objective is to construct the function f_{NN} that predicts the output $\hat{\boldsymbol{y}}$ as a good approximation of an unknown $\boldsymbol{y}$ for a new input $\boldsymbol{x}$.

The parameters $\boldsymbol{\theta}$ are optimized or learned based on a specific objective function with the procedure outlined in Chapter 2.3. Although NNs can theoretically approximate any continuous function, the learning procedure may not find the optimal parameters and introduce an error.

2.3 Optimization/Learning

The goal of learning is to minimize the expected error of novel data arising from an underlying data-generating process. Given only a dataset consisting of N samples, the expected error can only be indirectly minimized. To accomplish this, the dataset, consisting of $\boldsymbol{X}$ and $\boldsymbol{Y}$, is partitioned into two sets: the training set with $\boldsymbol{X}^{\text{train}}$ and $\boldsymbol{Y}^{\text{train}}$, and the validation set with $\boldsymbol{X}^{\text{val}}$ and $\boldsymbol{Y}^{\text{val}}$. In this thesis 80% of the dataset are used for training and 20% for validation, which is a common choice. The training set is employed to optimize the model parameters, while the validation set is used to estimate the expected error of new data and is thus the crucial quantity to quantify the model's generalization ability. (GOODFELLOW et al., 2016)

2.3.1 Cost Function

To optimize the model parameters, the cost function C, is used as the objective function. Typically, the cost function is represented as the mean value over a given dataset:

$$C = \frac{1}{N^{\text{set}}} \sum_{i=1}^{N^{\text{set}}} L(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i) \tag{2.2}$$

where N^{set} denotes the size of the dataset and L denotes the loss function measuring the error of a single data sample, by comparing the target values $\boldsymbol{y}_i$ with the model's prediction $\hat{\boldsymbol{y}}_i$. During the learning process, the cost of the training set is utilized to adjust the model parameters, while the cost of the validation set is employed to estimate the expected error on novel data.

Choosing the appropriate loss function is critical as it should possess favorable characteristics for optimization and result in good generalization performance on novel data. However, in certain cases, the optimization process may necessitate a different loss function than the one used for measuring performance on new data, resulting in an even more indirect optimization. (GOODFELLOW et al., 2016).

In regression problems, two popular loss functions are the square loss and the absolute loss, as illustrated in Figure 2.2 The square loss L_2 computes the squared difference between the true value $\boldsymbol{y}_i$ and the predicted value $\hat{\boldsymbol{y}}_i$. The model's individual prediction consists of n_{feat} values, which are compared element-wise with their respective target values.

$$L_2(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i) = \frac{1}{n_{\text{feat}}} \sum_{j=1}^{n_{\text{feat}}} (\boldsymbol{y}_{ij} - \hat{\boldsymbol{y}}_{ij})^2 \tag{2.3}$$

The square loss exhibits a linearly dependent gradient magnitude on the error, which results in a larger gradient magnitude for larger losses, and a gradient of zero for $L = 0$. This variable gradient length can be advantageous for optimization. However, the square loss is sensitive to outliers since it amplifies significant errors, leading to an increased emphasis on outliers.

In contrast, the absolute loss L_1 computes the absolute difference between the true value $\boldsymbol{y}_i$ and the predicted value $\hat{\boldsymbol{y}}_i$.

$$L_1(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i) = \frac{1}{n_{\text{feat}}} \sum_{j=1}^{n_{\text{feat}}} \left|\boldsymbol{y}_{ij} - \hat{\boldsymbol{y}}_{ij}\right| \tag{2.4}$$

Unlike the square loss, the absolute loss is more robust to outliers because it does not amplify large errors. Nonetheless, the gradient of the absolute error is constant and not smooth when the error approaches zero, which can negatively impact optimization.

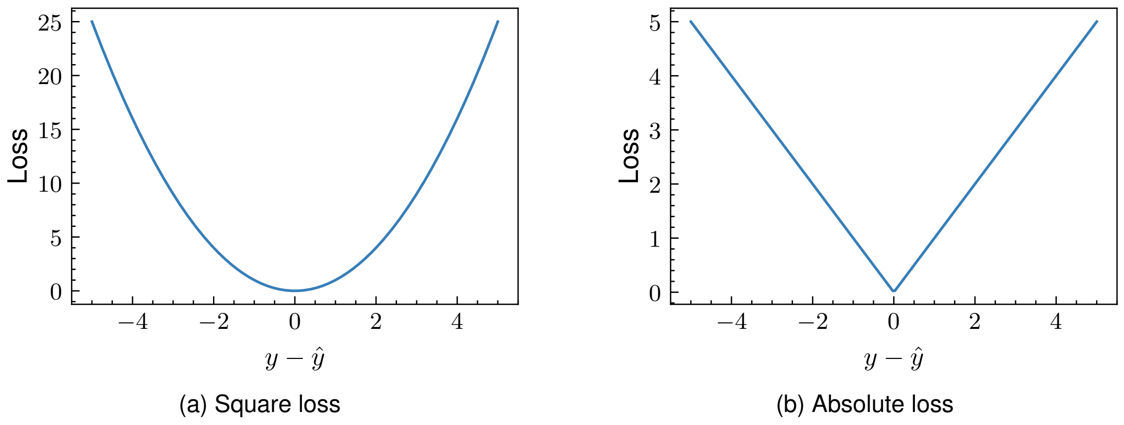


Figure 2.2: Common loss functions for regression problems: square loss L_2 and absolute loss L_1

Other commonly used loss functions for regression tasks include the Huber loss, the Log-cosh loss and the Quantile loss. Further details about loss functions in ML can be found in WANG et al. (2020).

2.3.2 Optimization Problem

The described indirect minimization of the expected error can be formulated as the minimization of the training cost C_{train} with respect to the model parameters $\boldsymbol{\theta}$.

$$\min_{\boldsymbol{\theta}} C_{\text{train}} \tag{2.5}$$

This is an unconstrained optimization problem. Since the loss function associates the individual predictions $\hat{\boldsymbol{y}}_i$ consisting of n_{feat} real values with a real number $L : \mathbb{R}^{n_{\text{feat}}} \to \mathbb{R}$ and the cost of the training set is just the mean of the individual losses, the optimization problem is a single-objective optimization problem. However, this problem is usually not solvable analytically, and gradient-based methods are used instead.

To compute the gradient of the cost function, the full cost function is typically too expensive to calculate the gradient. Instead, minibatch methods are used, which involve using subsets of the training dataset to approximate the gradient. Since the total cost of a dataset is the mean of the individual losses, the mean can be estimated on smaller subsets called batches. This is further motivated by the large dataset size, which features redundant information regarding the gradient. The batches are shuffled during the training process to mitigate the effect of correlation within the training set. By approximating the gradient using batches, the algorithm can update the parameters more frequently, leading to faster convergence of the optimizer. The choice of the batch size b is often influenced by hardware considerations, as larger batch sizes can offer better gradient approximation without additional computational cost if they can fit into GPU memory. Typically, the batch size that maximizes throughput, measured as examples processed per second, leads to the fastest training (Godbole et al., 2023).

Algorithm 2.1: Mini-batch gradient descent

```
Require: Training data inputs $X^{\text{train}}$ and targets $Y^{\text{train}}$, validation data inputs $X^{\text{val}}$ and
         targets $Y^{\text{val}}$, network architecture (see Chapter 3), optimizer (see
         Chapter 2.3.3), initialize model parameters $\boldsymbol{\theta}$
forall epochs do
    shuffle training dataset
    divide training dataset into n batches of size b
    forall batches do
        for $i < b$ do
            $\hat{\boldsymbol{y}}_i = f_{\text{NN}}(x_i; \theta)$                    // Get model prediction
            $L_i = L(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i)$   // Compute loss of sample with chosen loss function
        end for
        $C_{\text{b}} = \frac{1}{b} \sum_{i=1}^{b} L_i$
        optimizer.updateModelParameters()                // see Chapter 2.3.3
    end forall
    check validation cost and stopping criteria   // No update of model parameters
    if shouldStop then
        stop training early
    end if
end forall
```

The mini-batch gradient descent algorithm, outlined in Algorithm 2.1, starts by setting up the network architecture including the initialization of its parameters, the datasets, and the optimizer. The initialization of the model parameters specifies the starting point for the optimization and can strongly influence the convergence of the optimizer and the generalization abilities of the model. Usually the parameters are initialized in some random fashion. For further details the reader is referred to Goodfellow et al. (2016).

The entire dataset is processed by the optimizer a number of epochs. Before each epoch, the training dataset is shuffled and divided into n batches of the size b. The cost associated to this batch C_{b} is calculated and utilized by the optimizer to update the model parameters $\boldsymbol{\theta}$. Afterwards, the cost of the validation set is checked, and the stopping criteria is evaluated, which may result in the training being stopped early. Although, the training cost usually continues to decrease during training, the validation cost, which measures the performance on unseen data, may plateau or even increase, necessitating early termination. The training time is determined by the available number of epochs due to hardware constraints or the early stopping procedure.

The specifics of how the optimizer updates the model parameters have not been described and will be introduced in Chapter 2.3.3.

2.3.3 Optimizer

Several optimizers have been suggested for the training of NNs throughout the years. These optimizers vary in the type of information they employ to update the model parameters. While some optimizers utilize gradients, others also utilize second-order information. Moreover, some heuristic methods are often utilized to enhance the convergence behavior. (GOODFELLOW et al., 2016)

The Adam optimizer, which was introduced in KINGMA and BA (2014), is one of the most popular optimizers for NNs. As Adam only necessitates the computation of gradients, it is computationally less expensive per update step compared to optimizers also using second-order information.

Given that NNs are composed of nested functions and that the cost is a scalar, the gradient is determined via backpropagation using automatic differentiation. This approach differs from classical methods of differentiation, such as symbolic differentiation and numerical differentiation. Symbolic differentiation requires the explicit specification of derivatives with respect to certain parameters, while numerical differentiation introduces approximation errors. In contrast, automatic differentiation utilizes the chain rule, and backward mode is typically employed due to the scalar nature of the cost function, which increases computational efficiency. Further informations about automatic differentiation can be found in GRIEWANK and WALTHER (2008).

During the prediction step of a NN, a computational graph is generated by recording the applied operations and their corresponding derivatives. This approach leverages the fact that any complex computer program can be decomposed into simple steps with known derivatives. The gradient is then calculated through the repeated application of the chain rule starting from the cost. When considering a parameter being part of $f^{(2)}$ in the case of three functions as given in Chapter 2.2 this can be written as:

$$\frac{\partial C_{\text{train}}}{\partial \boldsymbol{\theta}} = \frac{\partial C_{\text{train}}}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{\theta}} = \left(\frac{\partial C_{\text{train}}}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial f^{(3)}} \right) \frac{\partial f^{(3)}}{\partial \boldsymbol{\theta}} \tag{2.6}$$

For further details regarding backprogagtion the reader is referred to NIELSEN (2015).

Adam, in addition to using the gradient information, incorporates momentum and adaptive learning rates to accelerate convergence. It calculates exponential moving averages of the gradient, represented by the first moment $\boldsymbol{m}$, and the squared gradient, represented by the second moment $\boldsymbol{v}$. The decay of these averages is controlled by the parameters β_1 and β_2. Since the estimates of $\boldsymbol{m}$ and $\boldsymbol{v}$ are biased due to their initialization as 0's, they are corrected, resulting in the bias-corrected versions, $\widehat{\boldsymbol{m}}$ and $\widehat{\boldsymbol{v}}$. The momentum parameter effectively increases the update step of the parameters with consistent gradient directions, while reducing the parameter update steps for parameters with inconsistent gradients. This helps to stabilize the learning process and smoothens out stochastic variations in the gradient updates. The learning rate adapts based on the square of the magnitudes of the recent gradients controlled by $\boldsymbol{v}$. Specifically, it normalizes the gradient by dividing it by the square root of the second moment. This normalization helps to prevent the learning rate from becoming too large and unstable, especially for parameters with large variances. Additionally, ϵ is added to the parameter update step to improve numerical stability. More information on the Adam optimizer can be found in GOODFELLOW et al. (2016) and KINGMA and BA (2014). The update step for Adam is summarized in Algorithm 2.2.

Algorithm 2.2: Adam optimizer: update model parameters

```
Require          : Adam parameters $\gamma$, $\beta_1$, $\beta_2$, $\epsilon$, weight decay $\lambda$
On training start: $\boldsymbol{m}= 0$ (first moment), $\boldsymbol{v}= 0$ (second moment)
$\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} C_{\mathrm{b}}$                                  // Application of backpropagation
if $\lambda \neq 0$ then
    $\boldsymbol{g} = \boldsymbol{g} + \lambda\boldsymbol{\theta}$                          // Add weight decay for regularization
end if
$\boldsymbol{m} = \beta_1\boldsymbol{m} + (1-\beta_1)\boldsymbol{g}$                        // Update first moment for momentum
$\boldsymbol{v} = \beta_2\boldsymbol{v} + (1-\beta_2)\boldsymbol{g}^2$       // Update second moment for adaptive learning rate
$\widehat{\boldsymbol{m}} = \frac{\boldsymbol{m}}{1-\beta_1}$                                             // Bias-correction
$\widehat{\boldsymbol{v}} = \frac{\boldsymbol{v}}{1-\beta_2}$                                             // Bias-correction
$\boldsymbol{\theta} = \boldsymbol{\theta} - \gamma\frac{\widehat{\boldsymbol{m}}}{\sqrt{\widehat{\boldsymbol{v}}}+\epsilon}$                                  // Update model parameters
```

The required parameters of the optimizer γ, β_1, β_2 and ϵ, the weight decay parameter λ and the parameters specifying the model architecture defined in Chapter 3, are all hyperparameters. These are configuration parameters defined before training, and they remain unchanged during training. They determine the model and training specifications and must be tuned separately from the model parameters $\boldsymbol{\theta}$. For a detailed explanation of the tuning process for these parameters, please refer to GODBOLE et al. (2023).

Weight decay, represented by λ, is a regularization method commonly used in NNs. It penalizes large model parameters, thereby encouraging the model to learn simpler patterns. The Universal approximation theorem, as discussed in Chapter 2.2, states that NNs can approximate any continuous function to a desired level of precision if given enough parameters. If a model fails to learn the patterns in the training dataset, this typically implies a lack of model parameters, also known as Underfitting.

However, in deep learning, a common issue is that the training and validation costs exhibit a large discrepancy when enough parameters are provided, which may indicate that the model is overemphasizing the training data and poorly learning the underlying patterns, known as Overfitting. Weight decay, also known as L2 regularization, addresses this issue by penalizing large model parameters associated with this behavior. Further regularization methods of NNs can be found in GOODFELLOW et al. (2016) and SRIVASTAVA et al. (2014).

Chapter 3

Convolutional Neural Networks

CNNs are a specialized type of NNs that are particularly well-suited for processing grid-like structures like images and time-series data. The name "convolutional" implies that CNNs use the mathematical operation convolution, which is a special linear operation. The convolution operation processes the input h depending on a parameter t with a convolution kernel k, which is shifted by τ, to produce the output or feature map s.

$$s(t) = h(t) * k(t) := \int_{-\infty}^{\infty} h(\tau)k(t-\tau)d\tau \tag{3.1}$$

In ML applications, both the input and the kernel are usually multidimensional arrays or tensors, such as images used in this thesis. In these cases, the integral in Equation (3.1) can be converted to a finite sum.

Unlike traditional NNs, which connect every input to every output in each layer, CNNs take advantage of the spatial or temporal structure of the data. This means that datapoints that are far apart are treated differently from those that are close together. By using small kernels to detect simple features like edges, more complex features can be assembled from these simpler ones. CNNs leverage sparse interactions, meaning that not every input has an impact on every output, by using a smaller kernel than the input and weight sharing by employing the same kernel across all parts of the image. This results in reduced memory requirements and faster computations. Further details on the ideas of CNNs can be found in GOODFELLOW et al. (2016) and NIELSEN (2015).

3.1 Convolution

Initially, this chapter presents a visual explanation of the 2D convolution process, focusing on one input and one output image. Afterwards, the convolution procedure is extended to involve multiple 2D input images and subsequently to multiple input and output 3D voxel images.

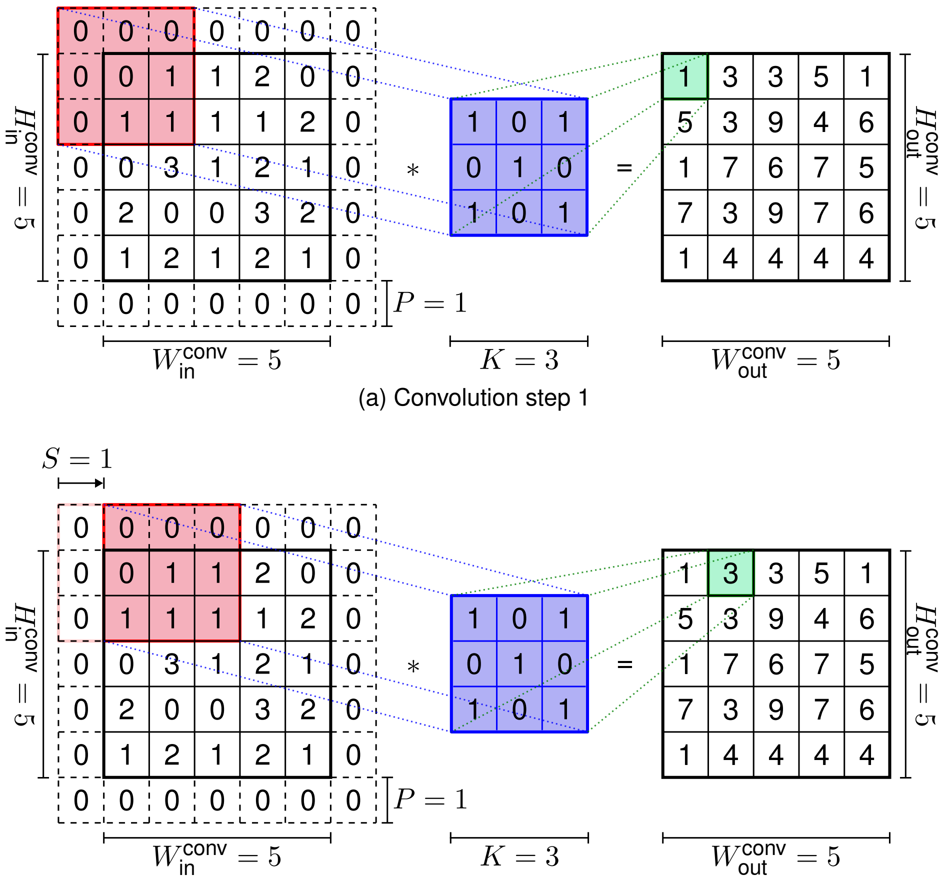
The convolution operation uses a small matrix of weights also known as a kernel that slides over the input image. During this process, the kernel performs elementwise multiplication with a specific input area, referred to as the receptive field, and then sums up the results to get the output pixel value. The output image's height and width, represented by $H_{\text{out}}^{\text{conv}}$ and $W_{\text{out}}^{\text{conv}}$, respectively, are calculated as follows:

$$H_{\text{out}}^{\text{conv}} = \frac{H_{\text{in}}^{\text{conv}} + 2P - K}{S} + 1 \tag{3.2}$$

$$W_{\text{out}}^{\text{conv}} = \frac{W_{\text{in}}^{\text{conv}} + 2P - K}{S} + 1 \tag{3.3}$$

$H_{\text{in}}^{\text{conv}}$ and $W_{\text{in}}^{\text{conv}}$ are the input image height and width, respectively. The equations are the same since the padding size P, the kernel size K, and the stride S are assumed to be isotropic. However, in more general cases, these values may differ in each direction. Padding adds fake pixels to the edges of the image, usually these pixels have the value of zero, which is known as zero padding.

Figure 3.1 demonstrates the sliding of the kernel over the image, displaying the first and second elementwise multiplication and summation.


$H_{\text{in}}^{\text{conv}} = 5$
$P = 1$
$W_{\text{in}}^{\text{conv}} = 5$
$K = 3$
$W_{\text{out}}^{\text{conv}} = 5$
$H_{\text{out}}^{\text{conv}} = 5$
$S = 1$
$*$
=

(a) Convolution step 1

(b) Convolution step 2: indicating the stride $S = 1$

Figure 3.1: Convolution of one 2D image

The process of extending the convolution to multiple input images and one output image is depicted in Figure 3.2. In this scenario, each input image undergoes convolution with its own kernel like depicted in Figure 3.1, and the resulting contributions are added up on a per-pixel basis. Moreover, a constant bias can be incorporated to generate the final output. To generate multiple output images, the aforementioned procedure is repeated for each output image.

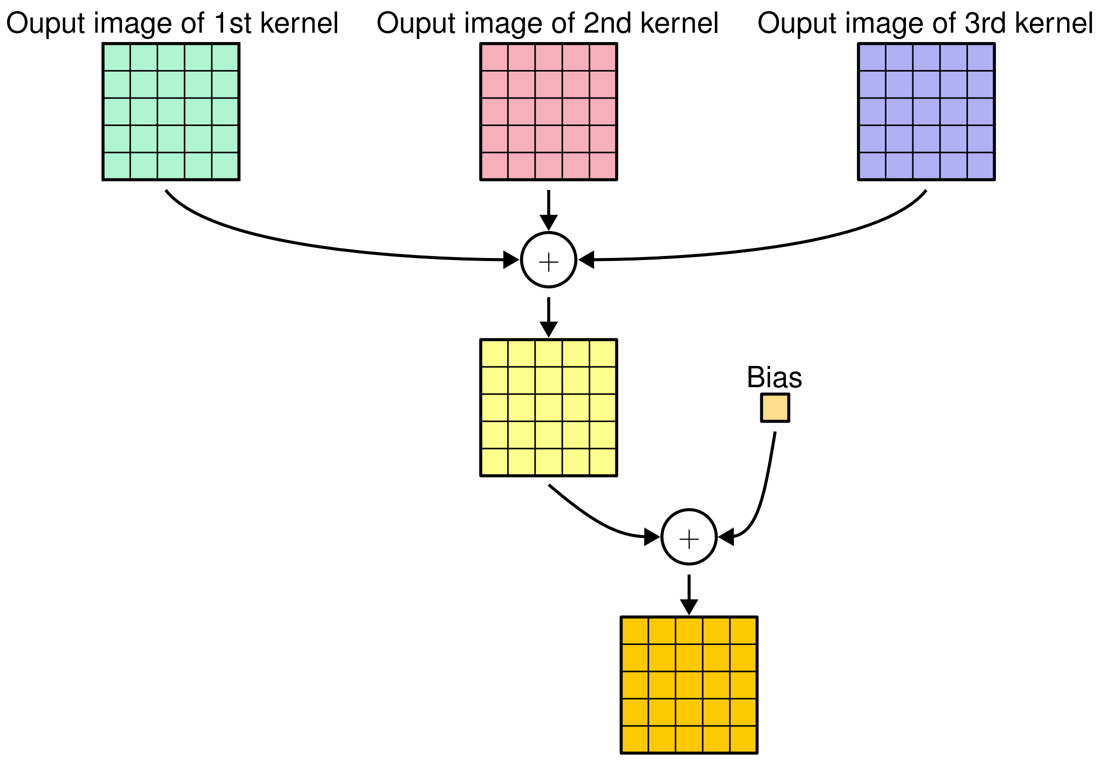

Ouput image of 1st kernel
Ouput image of 2nd kernel
Ouput image of 3rd kernel
+
Bias
+

Figure 3.2: Convolution of multiple 2D input images to one output image

The general case of multiple input and output 3D voxel images is given as:

$$\boldsymbol{y}_{ij}^{\text{conv}} = B_j^{\text{conv}} + \sum_{k=1}^{C_{\text{in}}^{\text{conv}}} \boldsymbol{W}_{jk}^{\text{conv}} * \boldsymbol{x}_{ik}^{\text{conv}} \tag{3.4}$$

The 3D convolution operation takes the input $\boldsymbol{x}^{\text{conv}}$ with dimensions ($N_{\text{in}}^{\text{conv}}$, $C_{\text{in}}^{\text{conv}}$, $D_{\text{in}}^{\text{conv}}$, $H_{\text{in}}^{\text{conv}}$, $W_{\text{in}}^{\text{conv}}$), and produces the output $\boldsymbol{y}^{\text{conv}}$ with dimensions ($N_{\text{in}}^{\text{conv}}$, $C_{\text{out}}^{\text{conv}}$, $D_{\text{out}}^{\text{conv}}$, $H_{\text{out}}^{\text{conv}}$, $W_{\text{out}}^{\text{conv}}$). The bias vector $\boldsymbol{B}^{\text{conv}} \in \mathbb{R}^{C_{\text{out}}^{\text{conv}}}$ contains the constants added to the output channels. The weight tensor $\boldsymbol{W}^{\text{conv}} \in \mathbb{R}^{C_{\text{out}}^{\text{conv}} \times C_{\text{in}}^{\text{conv}} \times K \times K \times K}$ contains different kernels of size $K \times K \times K$. The additional image dimension D adds a depth dimension to Figure 3.1, resulting in a 3D kernel that also operates on the input voxel image in the D direction.

3.2 Normalization

Normalization layers are a crucial component of most modern deep learning architectures as they address the problem of internal covariate shift that arises during training, which will be discussed subsequently. Various normalization layers have been introduced in recent years, including Batch Normalization (BN) (Ioffe & Szegedy, 2015), Layer Normalization (Ba et al., 2016), and Instance Normalization (Ulyanov et al., 2016). In this work, BN is used as the normalization layer.

During the training process the gradient tells how to update each parameter, under the assumption that only these parameters are changed. In practice, all parameters are updated simultaneously, and higher order interactions between layers may have a significant impact. The internal covariate shift problem occurs when layers need to adjust to varying input distributions arising from the parameter updates of the previous layers, which can be mitigated by normalization layers. Normalization layers reparameterize the output of the previous operation to promote uniform input distributions to the layers, allowing for higher learning rates, reducing the sensitivity of model parameter initialization, and serving as a form of regularization to prevent overfitting and improve the model's generalization ability. (IOFFE and SZEGEDY (2015); GOODFELLOW et al. (2016))

The operation of BN varies depending on whether it is used in training mode, as in the training loop in Algorithm 2.1, or evaluation mode, as in validation cost checking or prediction for new data.

During training, the statistical quantities $\boldsymbol{\mu}^{\text{BN,b}}$ and $\boldsymbol{\sigma}^{\text{BN,b}}$ of size $C_{\text{in}}^{\text{BN}}$ are calculated channel-wise for a batch of input data $\boldsymbol{x}^{\text{BN,b}}$ using the other four dimensions and applied to the same batch. To ensure numerical stability, a constant ϵ^{BN} is added to the denominator. This normalization step is followed by the application of learned scaling $\boldsymbol{\gamma}^{\text{BN}}$ and shifting parameters $\boldsymbol{\beta}^{\text{BN}}$, which enhance the network's expressive power and produce the output $\boldsymbol{y}^{\text{BN,b}}$.

$$\boldsymbol{y}_{ij}^{\text{BN,b}} = \frac{\boldsymbol{x}_{ij}^{\text{BN,b}} - \mu_j^{\text{BN,b}}}{\sqrt{\left(\sigma_j^{\text{BN,b}}\right)^2 + \epsilon^{\text{BN}}}} \gamma_j^{\text{BN}} + \beta_j^{\text{BN}} \tag{3.5}$$

In addition to calculating statistical quantities for each batch during training, the BN layer also keeps track of moving averages $\boldsymbol{\mu}^{\text{BN,mov}}$ and $\boldsymbol{\sigma}^{\text{BN,mov}}$ with a momentum of m^{BN}, which serve as an estimate of the statistical quantities for the whole dataset.

$$\mu_j^{\text{BN,mov,new}} = \left(1 - m^{\text{BN}}\right) \mu_j^{\text{BN,mov,old}} + m^{\text{BN}} \mu_j^{\text{BN,b}} \tag{3.6}$$

$$\sigma_j^{\text{BN,mov,new}} = \left(1 - m^{\text{BN}}\right) \sigma_j^{\text{BN,mov,old}} + m^{\text{BN}} \sigma_j^{\text{BN,b}} \tag{3.7}$$

These moving averages are then used in the evaluation mode to apply the normalization to a new input $\boldsymbol{x}^{\text{BN}}$, resulting in the output $\boldsymbol{y}^{\text{BN}}$. In evaluation mode the following linear transformation is applied:

$$\boldsymbol{y}_{ij}^{\text{BN}} = \frac{\boldsymbol{x}_{ij}^{\text{BN}} - \mu_j^{\text{BN,mov}}}{\sqrt{\left(\sigma_j^{\text{BN,mov}}\right)^2 + \epsilon^{\text{BN}}}} \gamma_j^{\text{BN}} + \beta_j^{\text{BN}} \tag{3.8}$$

3.3 Activation Function

Convolution and BN are linear transformations. Therefore, using only these components in a CNN would result in a linear model in total. To introduce non-linearity, a fixed nonlinear function known as an activation function is commonly employed without learnable parameters (GOODFELLOW et al., 2016).

The purpose of activation functions is to add non-linearity to the model while keeping computational complexity at a manageable level. Another crucial aspect is to avoid the vanishing gradient problem that can arise if the input values to the activation function remain in regions with gradients close to zero, leading to the learning of previous layers to stop. (DUBEY et al., 2021)

There have been several proposals for activation functions in recent years, and DUBEY et al. (2021) provides an extensive overview of these functions and their properties.

The activation function a is typically applied element-wise to the input x^{A}_{ijklm} to generate the output y^{A}_{ijklm}:

$$y^{\mathsf{A}}_{ijklm} = a\left(x^{\mathsf{A}}_{ijklm}\right) \tag{3.9}$$

In this thesis, the LeakyReLU activation function is employed, with a negative slope of 0.01. It is defined as follows:

$$LeakyReLU(x_{ijklm}) = \begin{cases} x_{ijklm}, & \text{if } x \geq 0 \\ 0.01 x_{ijklm}, & \text{otherwise} \end{cases} \tag{3.10}$$

To improve visibility, the function is visualized in Figure 3.3 with a negative slope of 0.1.

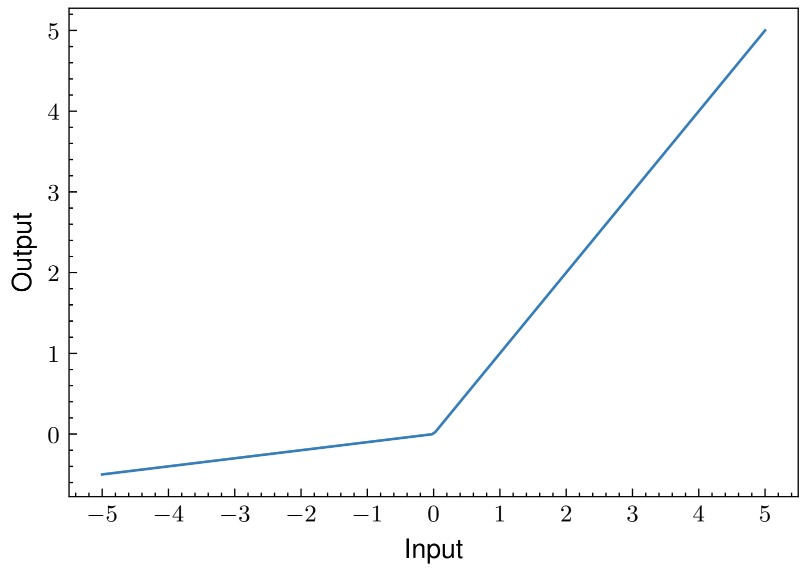


Figure 3.3: LeakyReLU activation function with negative slope of 0.1

3.4 The U-Net Family

The U-Net is a commonly utilized architecture for image-to-image problems, as described in the publication by RONNEBERGER et al. (2015). The architecture is named after its symmetrical U-shape, consisting of an encoder and a decoder connected by skip connections. The encoder captures contextual information and compresses it into a representation, while the decoder reconstructs the output images from this compressed representation. To improve localization capabilities, high resolution features are forwarded from the encoder to the decoder via skip connections. Skip connections help the network recover spatial information lost during downsampling and prevent gradients from becoming too small during backpropagation.

In Figure 3.4, a parameterized U-Net model is depicted for 2D images with a model depth $D_\text{m} = 1$, which can be easily extended to 3D by incorporating additionally the image depth D.

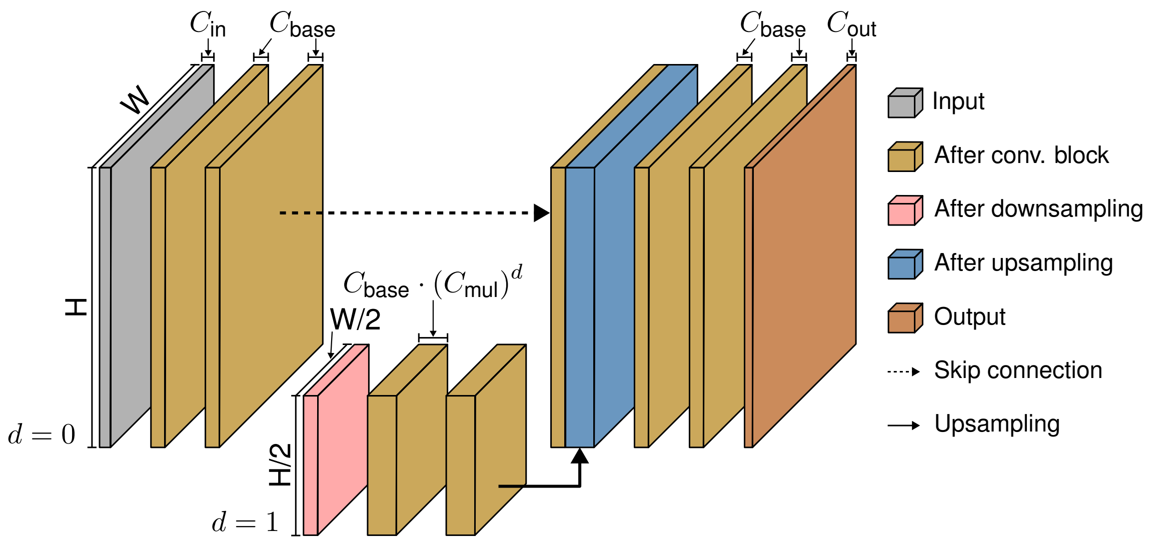


Figure 3.4: Parametrized U-Net model

The input images have dimensions $H \times W$ and consist of C_in input channels. The encoder starts with a convolution block that consists of a convolution operation followed by BN and a LeakyReLU activation function resulting in an increase of the number of channels to C_base. This is followed by another convolution block, after which a downsampling operation is applied to halve the image dimensions. The downsampling is performed using a convolution operation with a kernel size $K = 2$, a stride $S = 2$, and no padding $P = 0$. This operation is learned since the weights and biases are adjusted during training. Alternatively, max-pooling or average pooling can be applied as a not learned downsampling operation, which have a smaller expressive power and are hence not utilized.

After downsampling, two convolution blocks follow that increase the number of channels by the factor C_mul. For models with a depth greater than one, this procedure is repeated, resulting in further halving of the image dimensions and an increase in the number of channels.

The decoder component of the model begins by upsampling the images from the previous depth dimension d received from the encoder. Upsampling can be either a learned or a not learned operation. The current model uses an upsampling method based on linear interpolation, which is a not learned operation. Alternatively, transposed convolutions are a learned upsampling strategy, which applies the basic convolution operation in reverse, where one input value is expanded into multiple output values. Due to the proneness of transposed convolutions to checkerboard patterns (ODENA et al., 2016), upsampling based on interpolation is preferred in this work.

The upsampled images are concatenated with images from the encoder that were forwarded unchanged to the decoder via skip connections. The basic convolution block is then applied twice to the concatenated images. Finally, a convolution is applied to obtain the output channels C_{out}, without the application of BN and activation function.

Chapter 4

Data Generation

To train the CNN in a supervised fashion a labeled dataset containing both input and target variables is necessary. The methodology for acquiring the labeled dataset is outlined in the chapters ahead.

4.1 Geometry Creation

The input data includes the geometry, which is defined using a voxel-based description. The geometries in this thesis were generated using the Hyperganic core voxel engine, which follows a set of instructions known as a recipe. The recipe outlines the sequence of operations to produce the desired geometry and depends on specific input parameters defining a low-dimensional subspace in which the investigated geometries exist. Theoretically, the subspace should be well-suited for the learning process of NNs.

This work employs a layered stochastic lattice recipe. This geometric recipe involves the definition of a volume by its surface and subsequent population with a series of interconnected layers featuring stochastic lattice structures. The overarching structure of the recipe is summarized in Algorithm 4.1. To simplify the explanation, the input arguments are sometimes not specified explicitly, but are described in the following. It is important to note that this recipe is just one example of a data generation method and can be substituted with any other recipe.

Initially, the recipe transforms an input volume, specified by its boundary representation, into a voxel description. A voxel denotes a three-dimensional pixel that represents a value in 3D space. The data structure used to represent volumetric data, where the value of each voxel describes the material density is called density field.

Next, the layer properties are calculated utilizing the following parameters:

- The number of layers n_{Layers}
- The minimum layer thickness
- The maximum layer thickness
- The minimum beam thickness for connecting points within layers
- The maximum beam thickness
- The distribution method: determines whether the beam and layer thickness increase or decrease linearly towards the interior of the volume

Algorithm 4.1: Geometric Recipe

Input : Geometric Parameters

Output : Layered Stochastic Lattice

```
Function layeredStochasticLattice = geometricRecipe(geometricParameters)
    inputDensityField = voxelize(inputGeometry)  // converts geometry to voxels
    layerProperties = calculateLayerProperties(...)
    surfacePointsList = [ ]
    volumePointsList = [ ]
    // For the first layer, the whole object is considered to distribute
       the points.
    remainingDensityField = inputDensityField
    // Zero based indexing is used here.
    for $layer < n_{\text{Layers}}$ do
        layerSurfacePoints = distributeSurfacePoints(remainingDensityField)
        surfacePointsList.append(layerSurfacePoints)
        // Object is offset inward.  Removed density is the shell.
           RemainingDensityField is used for next layer.
        shell, remainingDensityField = offsetDensityField(remainingDensityField,
         currentLayerThickness)
        layerVolumePoints = distributeVolumePoints(shell, ...)
        // Only volume points inside the shell points are retained.
        volumePointsList.append(layerVolumePoints)
    end for
    latticesList = [ ]
    // Zero based indexing is used here.
    for $layer < n_{\text{Layers}} - 1$ do
        // One lattice is created per layer, which connects surface points
           from the current and next layer, as well as volume points from
           the current layer.  This ensures connectivity between layers.
        layerLattice = connectClosestPoints(surfacePoints[layer:layer+1],
         volumePoints[layer])
        latticesList.append(layerLattice)
    end for
    // Only final surface and volume points are used.
    latticeFinalLayer = connectClosestPoints(surfacePoints[-1], volumePoints[-1])
    latticesList.append(latticeFinalLayer)
    layeredStochasticLattice = mergeLattices(latticesList)
    return layeredStochasticLattice
end
```

The main part of the geometric recipe comprises two distinct parts. The first part generates points in space, while the subsequent section connects the generated points to create the desired geometry.

For each layer, a series of points is generated on the surface of the remaining density field, which is subsequently offset inward by a thickness equivalent to the layer thickness. The shell generated via this offsetting procedure is utilized to create volume points, achieved through the stochastic generation of points within the bounding box of the shell and confinement within the shell itself.

The following parameters govern this operation:

- The grid size: for each cell in the grid one point is generated.
- The noise ratio and the random seed: controls the stochasticity of the generated points.

The generated surface and volume points are subsequently connected based on their proximity and the parameter specifying the number of points to connect. During this connecting procedure, the surface and volume points from the current layer, as well as the surface points from the subsequent layer, are considered for all layers except the final one.

The images presented in Figure 4.1 depict the diverse geometries produced through the use of this recipe, resulting in notable differences in fill ratios.

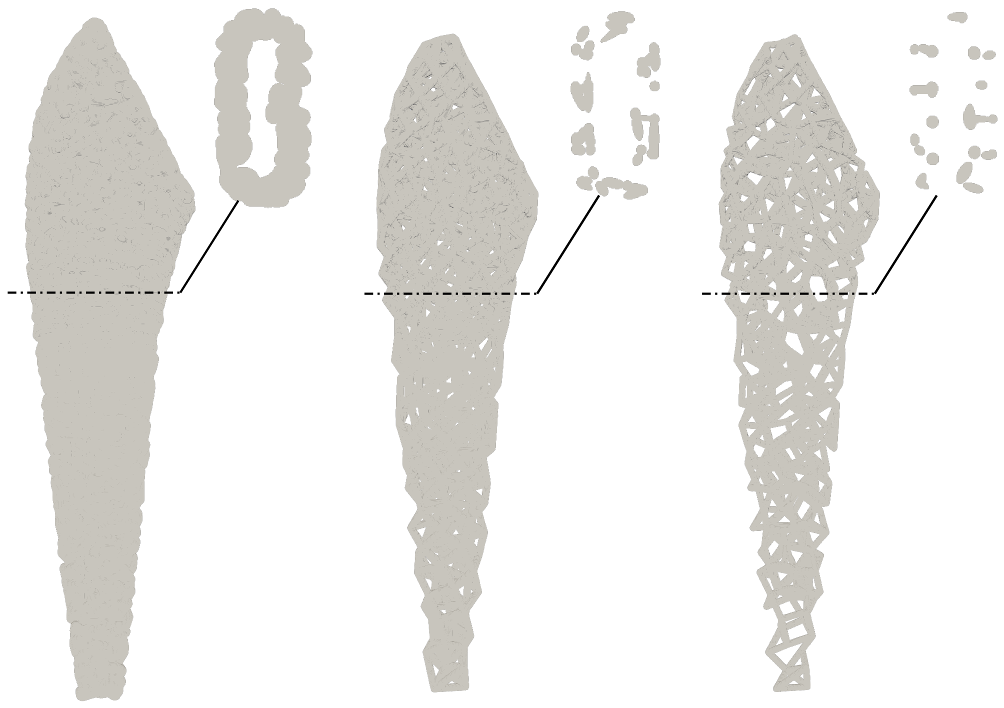


Figure 4.1: Example geometries created with the geometric recipe

4.2 Simulation Setup

Once a voxel-based geometry is obtained using Algorithm 4.1, the following simulation parameters must be specified to set up a valid simulation:

- Analysis type, which specifies the physics
- Material properties
- Discretization parameters for the simulation
- Boundary conditions

In this work, the analysis type is set to static linear elastic and the material is assumed to be isotropic and homogenous.

Consequently, the specification of two essential material parameters like Young's modulus (E) and Poisson's ratio (ν) is required. Alternatively, other material constants like the Lamé parameters could be used. The Partial Differential Equation (PDE) governing the physical phenomena is expressed as:

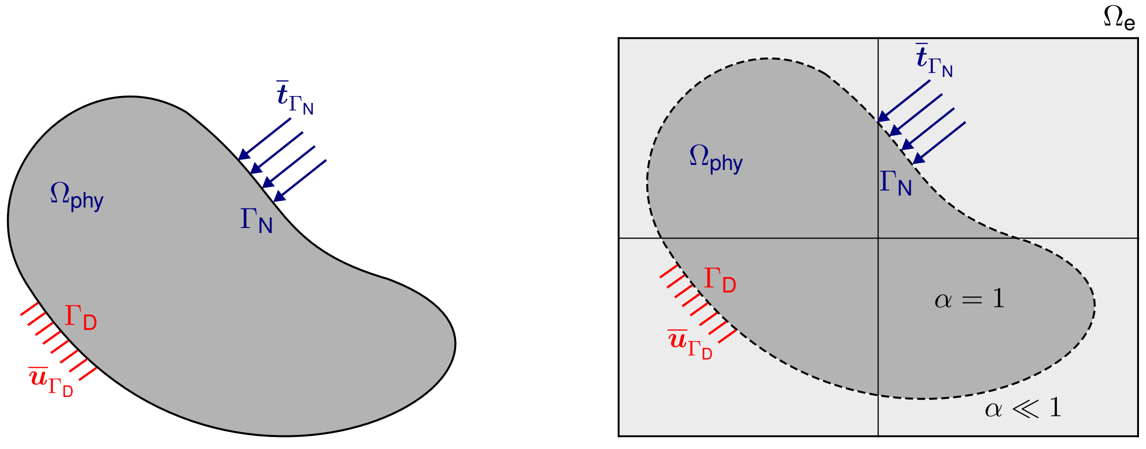
$$\frac{E}{2(1+\nu)}\left[\Delta \boldsymbol{u} + \frac{1}{1-2\nu}\nabla(\nabla\cdot\boldsymbol{u})\right] + \boldsymbol{f} = 0 \quad \text{in } \Omega_{\text{phy}} \tag{4.1}$$

$\boldsymbol{u}$ is the displacement vector of each point in the domain, $\boldsymbol{f}$ is the force density resulting from sources like gravity. To solve this PDE, at least a valid set of prescribed displacements $\overline{\boldsymbol{u}}_{\Gamma_{\text{D}}}$ on the Dirichlet boundary Γ_{D} must be specified along with the possibility of specifying prescribed traction vectors $\overline{\boldsymbol{t}}_{\Gamma_{\text{N}}}$ on the Neumann boundary Γ_{N}.

Although the model problem presented herein serves as the basis for the subsequent investigations, other PDEs can be employed to generate data for the learning process of the CNN.

The solution to the PDE is approximated using the Finite Cell Method (FCM), an embedded domain method that leverages simple unfitted structured grids, thus avoiding the need for complex body-conforming mesh generation. Figure 4.2 illustrates the general idea of the FCM. The physical domain is embedded within the extended domain Ω_{e}, where the structured grid is applied. Continuous shape functions are defined on this structured grid. However, since the true solution is discontinuous, the integration procedure must account for this discrepancy by introducing the indicator function α. Specifically, the value of α is set to 1 inside and close to 0 outside the physical domain. For more detailed explanations, refer to DÜSTER et al. (2017).

In the general case, the embedded domain approach entails incorporating Boundary Conditions (BCs) within elements, which necessitates weakly enforcing DBCs (RUESS et al., 2013). In this study, DBCs are treated as volumetric constraints on voxel geometries and Neumann Boundary Conditions (NBCs) are converted to local force densities.



(a) Physical domain Ω_{phy} (b) Extended domain Ω_{e} with structured grid

Figure 4.2: General idea of the FCM

The simulation workflow is presented in Figure 4.3. Initially, the analysis type and the required material properties E and ν are specified for the geometries generated by Algorithm 4.1. The simulation domain is then set up using the number of voxels per element as a discretization parameter, which is determined from a convergence study presented in Chapter 4.3. Subsequently, the volumetric constraints originating from the DBCs and the local force densities arising from the NBCs are imposed on the voxel geometry, resulting in a continuous solution for the displacement field.

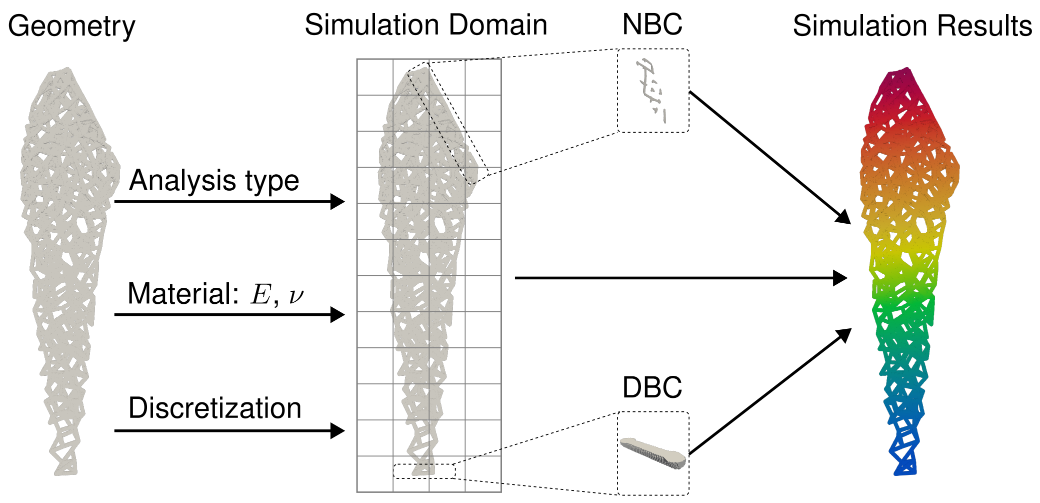


Figure 4.3: Simulation workflow

4.3 Convergence

When applying deep learning-based surrogate models in practice, it is crucial to learn the actual solution to the problem rather than one that results from a coarse discretization. Therefore, a convergence study is conducted to determine the appropriate number of voxels per element for achieving a converged solution. The convergence is measured by the energy norm, which depends on the stress tensor $\boldsymbol{\sigma}$ and the strain tensor $\boldsymbol{\epsilon}$ and is given by Equation (4.2). The convergence behavior is shown in Figure 4.4, where the energy norm is plotted against the Degrees of Freedom (DoFs). The voxel size is kept constant at 0.1 length units, while the number of voxels per element is decreased and labeled accordingly.

$$\|\boldsymbol{u}\|_{E(\Omega_{\text{phy}})} := \sqrt{\frac{1}{2}\int_{\Omega_{\text{phy}}} \boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\epsilon}(\boldsymbol{u})\, d\Omega_{\text{phy}}} \tag{4.2}$$

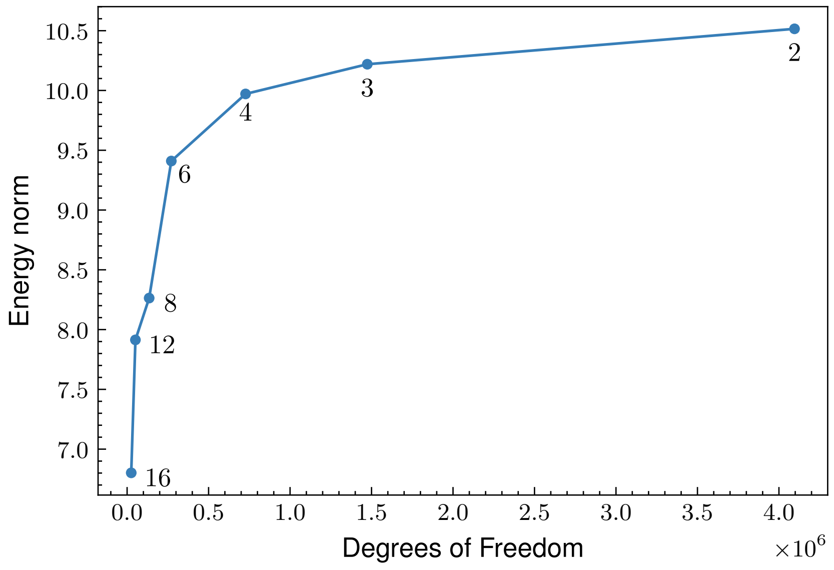


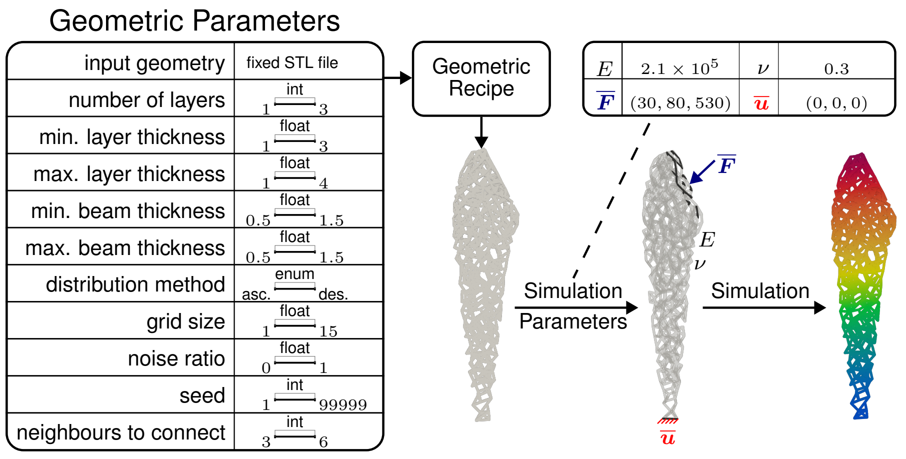
Figure 4.4: Convergence of the energy norm, depending on the DoFs. The number of voxels per element as the discretization parameter are used as a label.

At 4 voxels per element a flattening-out behavior is observed. Although the energy norm has not achieved full convergence, a discretization of 4 voxels per element is utilized throughout this study due to limited computer resources for creating the dataset. This decision does not adversely affect the conclusions obtained regarding the training of the CNNs. However, it is important to consider this when evaluating errors in practical cases.

4.4 Sampling

To create the labeled dataset necessary for the supervised learning approach, the generic geometric recipe, specified in Algorithm 4.1, along with the simulation setup from Chapter 4.2 needs to be sampled. Figure 4.5 summarizes the sampling strategy. A fixed Stereolithography (STL) file, defining a part of a hip implant, is used to establish the input geometry for all simulations. Every geometric parameter is uniformly sampled within the specified range and according to its type, meaning integer, floating-point number or enumeration. It is important to note that creating the dataset is a time-intensive process, as a single simulation resulting in one sample point takes approximately three minutes.

These parameters serve as the input to the geometric recipe, resulting in a certain voxel geometry. However, for certain parameter combinations the geometric recipe may lead to an invalid geometry. In all conducted simulations, the material parameters are fixed at $E = 2.1 \times 10^5$ and $\nu = 0.3$. The volumetric constraints arising from the DBCs are also kept constant, with the volumetric displacement constraint $\overline{\boldsymbol{u}}$ applied to the bottom voxels within a single length unit and possessing the components $(0, 0, 0)$. Similarly, the NBCs are considered as a volumetric total force constraint $\overline{\boldsymbol{F}}$ with the components $(30, 80, 530)$, which is applied to the voxels situated atop the inclined plane within a single length unit. Given the varying nature of the geometries across different samples, the volumes of the prescribed constraints may exhibit slight variations.



Geometric Parameters

input geometry	fixed STL file
number of layers	int 1 – 3
min. layer thickness	float 1 – 3
max. layer thickness	float 1 – 4
min. beam thickness	float 0.5 – 1.5
max. beam thickness	float 0.5 – 1.5
distribution method	enum asc. – des.
grid size	float 1 – 15
noise ratio	float 0 – 1
seed	int 1 – 99999
neighbours to connect	int 3 – 6

E	2.1×10^5	ν	0.3
$\overline{\boldsymbol{F}}$	$(30, 80, 530)$	$\overline{\boldsymbol{u}}$	$(0, 0, 0)$

Figure 4.5: Sampling strategy

Chapter 5

Learning Methodology

This chapter describes the methodology used to prepare the simulation results obtained in Chapter 4 for training the investigated CNNs. Two models are presented, each utilizing two preprocessing techniques: data encoding and scaling of input and target data. Data encoding involves transforming the simulation data into a format that can be fed into the CNNs, as explained in Chapter 5.1. The encoding process is unidirectional, meaning that the simulation data is encoded in the unscaled dataset, but it cannot be uniquely reconstructed from the unscaled data.
In contrast, data scaling, described in Chapter 5.2, is bidirectional and there exists a unique relationship between scaled and unscaled data.
The two examined models differ in their prediction and target variables. The loss function used to train the models compares the model's prediction with a scaled target. The performance evaluation, on the other hand, relies on metrics that compare the unscaled predictions and targets.

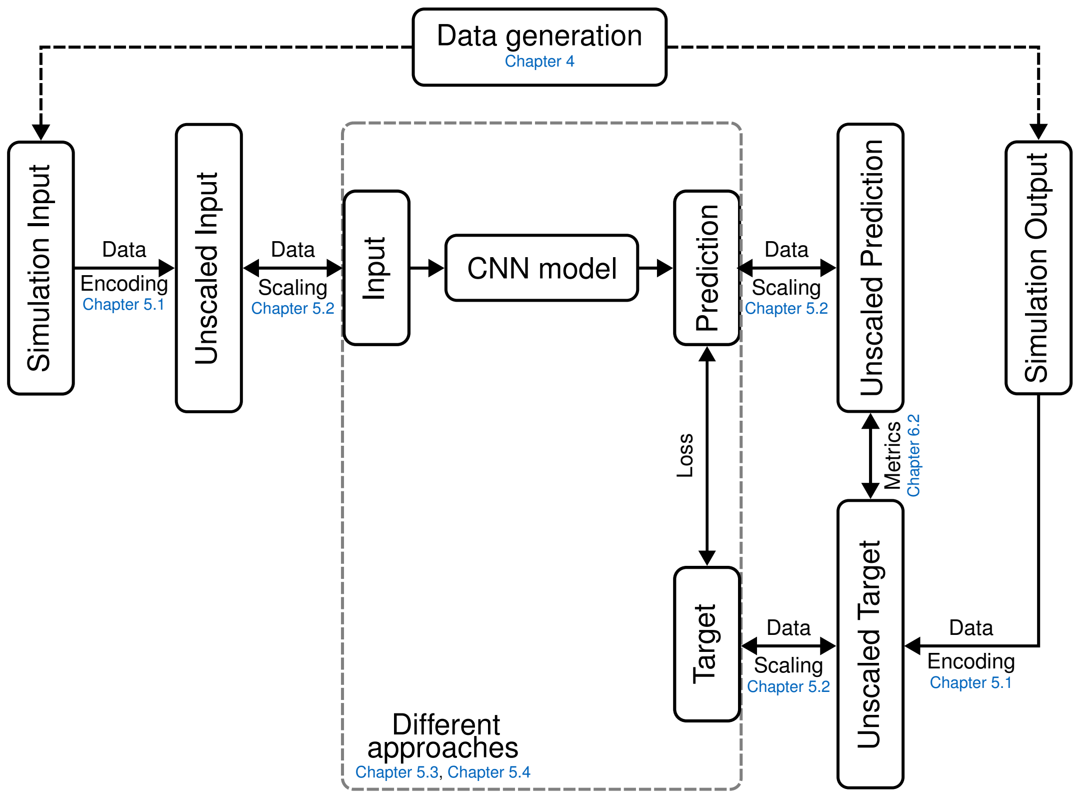


Figure 5.1: Summary of the learning methodology

5.1 Data Encoding

The learning task will be formulated as an image-to-image regression problem, wherein the input and target consist of floating-point number images. While the geometries originate from a low-dimensional subspace, they are encoded as images because this encoding method is not limited to a specific geometric recipe, enabling the use of geometries from similar recipes. The unscaled input is composed of 11 images, as outlined below:

- Two images representing the spatial distribution of the material parameters, namely E and ν. In the case of a homogenous material, all voxels within each image in the physical simulation domain Ω_{phy} exhibit identical values.
- Six images are designated for the volumetric constraints arising from the DBCs. For every component of $\overline{\boldsymbol{u}}$, a binary mask image and a prescribed value image are employed. The binary mask enables the CNN to distinguish between voxels with and without volumetric constraints, since the voxel value zero in the prescribed value image would not be uniquely defined. The value zero would encode both the absence of prescribed volumetric constraints and a prescribed component of zero.
- Three images defining the components of the applied force density in the physical simulation domain Ω_{phy} resulting from the NBCs or, also not considered in this work, other volumetric sources like gravity.

The unscaled target images encompass the three components of the displacement.

To generate the simulation input and output, the sampling strategy outlined in Chapter 4.4 is employed, and the resulting data is preprocessed to create the aforementioned images. The subsequent steps are followed:

1. The continuous simulation results are evaluated at the voxel midpoint forming 3D image tensors.
2. The simulation results, domain geometry, and volumetric constraints are downsampled to reduce computational cost and leverage the approximation capabilities of CNNs. In this work, $6 \times 6 \times 6$ voxels are merged into one voxel by selecting the value of every sixth voxel. However, alternative downsampling techniques can also be used. It is worth mentioning that this step may be skipped in certain scenarios.
3. The value of voxels belonging to the extended domain Ω_{e} but not to the physical domain Ω_{phy} is set to zero.
4. To enable the up- and downsampling operations performed by the CNN models, the 3D image tensor is subjected to zero padding, wherein zeros are appended to the edges of the tensor until its dimensions become a power of two.

The following serves as an example for the development of the tensor dimensions during the aforementioned steps:

- After step 1: $1335 \times 335 \times 150$
- After step 2: $223 \times 56 \times 25$
- After step 3: $223 \times 56 \times 25$
- After step 4: $256 \times 64 \times 32$, defining the image dimensions D,H,W
- Resulting input dimensions (N,C_{in},D,H,W): $N \times 11 \times 256 \times 64 \times 32$
- Resulting output dimensions (N,C_{out},D,H,W): $N \times 3 \times 256 \times 64 \times 32$

Due to this specific encoding method, the major part of the images is outside the physical domain Ω_{phy} and encoded as zero, hence the images are quite sparse. Moreover, the images of the volumetric constraints arising from the DBCs contain only a few non-zero voxels.
This step can be performed independently for each simulation sample obtained with the sampling strategy described in Chapter 4.4.

5.2 Data Scaling

Scaling the labeled dataset to a uniform data range is a frequently used preprocessing step in deep learning. This technique improves the optimization process by improving the conditioning of the loss function. When the data spans a wide range of values, the gradients of the loss function can become too large or too small, causing slow convergence or even preventing convergence altogether (LeCun et al., 2012). Another advantage is the reduced sensitivity to the initialization of the parameters of the CNN, potentially improving the training process's stability (Bishop, 1995).

In this work, standardization is used as a data scaling method, which applies the linear transformation given by Equation (5.1) to a distribution z, resulting in a distribution $\tilde{z}$ with mean 0 and standard deviation 1. μ^z and σ^z denote the mean and standard deviation of the distribution z.

$$\tilde{z} = \frac{z - \mu^z}{\sigma^z} \tag{5.1}$$

Other commonly used scaling techniques include Min-Max scaling, which scales data to a specified range such as $[-1, 1]$ by also applying a linear transformation. Standardization, on the other hand, does not limit the data to a predetermined range but instead adjusts the statistics of the scaled and shifted distribution. All linear transformations preserve the linear correlation coefficient between variables (Navidi, 2011), which is important for the CNN to extract the underlying patterns of the data.

For the image-to-image regression problems, the input and target channels are standardized separately. Since a binary encoding of the physical domain Ω_{phy} and the volumetric constraints arising from the DBCs is utilized, only voxel values belonging to their specific binary mask are scaled. The binary masks extract a simulation and channel dependent subset of the voxel values from the input dataset tensor X_{ijklm} and target dataset tensor Y_{ijklm}, where i represents the simulation, j represents the channel index, and klm represent spatial locations. The masked versions of the input and target datasets are denoted as X_{ijk}^{masked} and Y_{ijk}^{masked}, respectively. The size of the third dimension of the masked datasets varies depending on the simulation i and channel j since the binary masks are simulation and channel-specific. Although the length of the index k of the masked datasets varies, the common index notation of tensors is used, albeit with an abuse of notation. Statistical quantities such as mean and standard deviation vectors $\boldsymbol{\mu}^X$, $\boldsymbol{\sigma}^X \in \mathbb{R}^{C_{\text{in}}}$ and $\boldsymbol{\mu}^Y$, $\boldsymbol{\sigma}^Y \in \mathbb{R}^{C_{\text{out}}}$ are calculated based on the first and third dimensions of the masked tensors, where the simulation belongs to the training set. These statistical quantities are then used to apply a linear transformation on the masked dataset tensors, resulting in their standardized form $\tilde{X}_{ijk}^{\text{masked}}$ and $\tilde{Y}_{ijk}^{\text{masked}}$.

$$\tilde{X}_{ijk}^{\text{masked}} = \frac{X_{ijk}^{\text{masked}} - \mu_j^X}{\sigma_j^X} \tag{5.2}$$

$$\tilde{Y}_{ijk}^{\text{masked}} = \frac{Y_{ijk}^{\text{masked}} - \mu_j^Y}{\sigma_j^Y} \tag{5.3}$$

If the voxel values are homogenous across the entire dataset, such as in the case of material parameters like E and ν, the standard deviation is zero. In such cases, the transformation is adjusted such that the scaled value is 1 if the mean is not equal to zero, or 0 if the mean is zero. This results in a scaled version of the input and target image tensors, denoted as $\tilde{X}_{ijklm}$ and $\tilde{Y}_{ijklm}$.

5.3 Base Model

The U-Net serves as a base model, which learns to map the 11 channel-wise scaled input images to the channel-wise scaled target images, consisting of the three displacement components. The model learns the input-output mapping based on the scaled dataset tensors belonging to the training set. Scaled input data samples are denoted as $\tilde{\boldsymbol{x}}_i \in \mathbb{R}^{C_{\text{in}} \times D \times H \times W}$ and scaled target samples as $\tilde{\boldsymbol{y}}_i \in \mathbb{R}^{C_{\text{out}} \times D \times H \times W}$. The model's predictions $\hat{\tilde{\boldsymbol{y}}}_i \in \mathbb{R}^{C_{\text{out}} \times D \times H \times W}$ are compared with the scaled target based on the loss function. Figure 5.2 summaries the aforementioned.

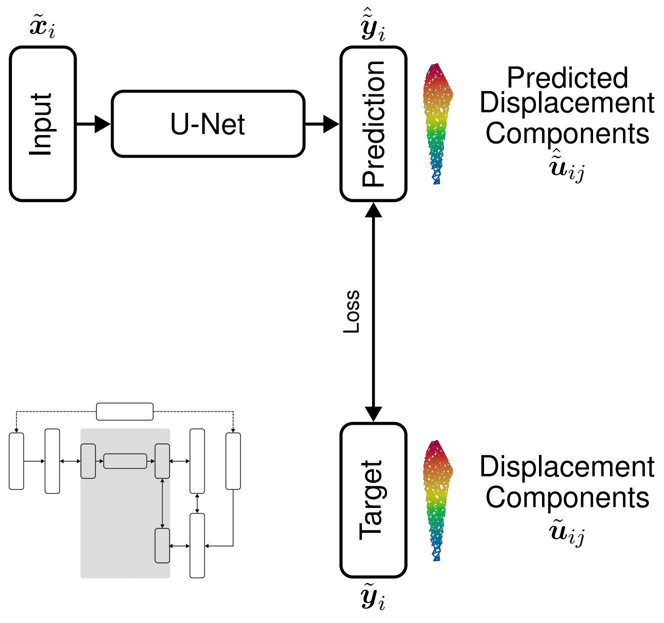

$\tilde{\boldsymbol{x}}_i$
$\hat{\tilde{\boldsymbol{y}}}_i$
Input
U-Net
Prediction
Predicted Displacement Components $\hat{\tilde{\boldsymbol{u}}}_{ij}$
Loss
Target
Displacement Components $\tilde{\boldsymbol{u}}_{ij}$
$\tilde{\boldsymbol{y}}_i$

Figure 5.2: Base model: U-Net

5.4 Image Network and Scaling Network

The second approach examined in this study involved a CNN model comprised of two parts: the Scaling-Net and the Image-Net. The Image-Net is similar to the basic U-Net model, but it is trained to learn a different target. Specifically, it learns the spatial distribution of values within the simulation- and channel-wise standardized target dataset denoted as $\tilde{\tilde{Y}}_{ijklm}$, hence it is referred to as the Image-Net. An additional standardization step is necessary, which ensures that all masked images $\tilde{\tilde{Y}}^{\text{masked}}_{ijk}$ of the simulations have a mean of 0 and a standard deviation of 1 individually. The scaling factors $\boldsymbol{\mu}^{\tilde{Y}}$, $\boldsymbol{\sigma}^{\tilde{Y}} \in \mathbb{R}^{N \times C_{\text{out}}}$ apply a linear transformation defined as:

$$\tilde{\tilde{Y}}^{\text{masked}}_{ijk} = \frac{\tilde{Y}^{\text{masked}}_{ijk} - \mu^{\tilde{Y}}_{ij}}{\sigma^{\tilde{Y}}_{ij}} \tag{5.4}$$

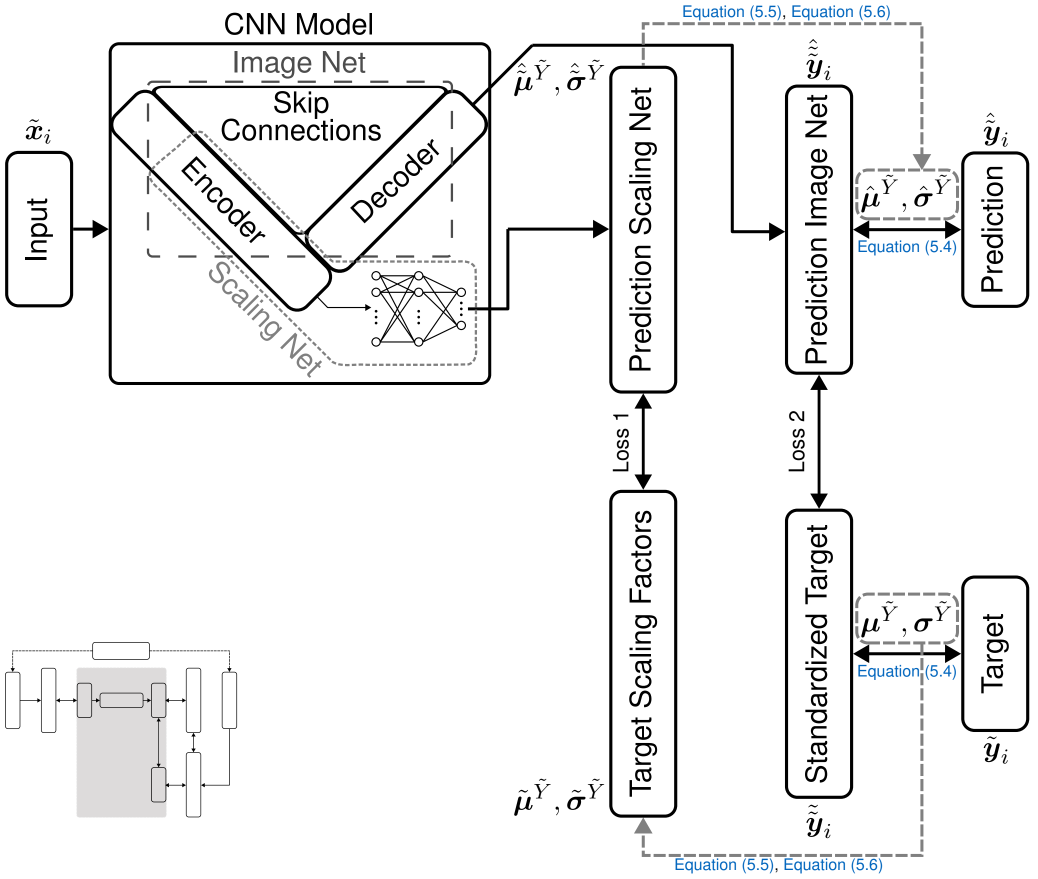
The prediction of the Image-Net, denoted as $\hat{\tilde{\tilde{Y}}}_{ijklm}$, is compared with the unmasked image tensor $\tilde{\tilde{Y}}_{ijklm}$ using an appropriate loss function. Since the additional scaling factors are dependent on the individual simulation, they cannot be precomputed but need to be learned.

The Scaling-Net consists of the encoder of the U-Net and two fully connected layers with a LeakyReLU activation function after the first linear layer. The purpose of the Scaling-Net is to learn a standardized version of the scaling factors $\boldsymbol{\mu}^{\tilde{Y}}$ and $\boldsymbol{\sigma}^{\tilde{Y}}$, which are denoted as $\tilde{\boldsymbol{\mu}}^{\tilde{Y}}$ and $\tilde{\boldsymbol{\sigma}}^{\tilde{Y}}$. To obtain this standardized version, additional scaling factors are precomputed based on the training dataset, and are denoted as $\boldsymbol{\mu}^{\mu^{\tilde{Y}}}$, $\boldsymbol{\mu}^{\sigma^{\tilde{Y}}}$, $\boldsymbol{\sigma}^{\mu^{\tilde{Y}}}$, $\boldsymbol{\sigma}^{\sigma^{\tilde{Y}}} \in \mathbb{R}^{C_{\text{out}}}$. These factors are used to apply linear transformations defined as:

$$\tilde{\mu}^{\tilde{Y}}_{ij} = \frac{\mu^{\tilde{Y}}_{ij} - \mu^{\mu^{\tilde{Y}}}_{j}}{\sigma^{\mu^{\tilde{Y}}}_{j}} \tag{5.5}$$

$$\tilde{\sigma}^{\tilde{Y}}_{ij} = \frac{\sigma^{\tilde{Y}}_{ij} - \mu^{\sigma^{\tilde{Y}}}_{j}}{\sigma^{\sigma^{\tilde{Y}}}_{j}} \tag{5.6}$$

The Scaling-Net predicts $\hat{\tilde{\boldsymbol{\mu}}}^{\tilde{Y}}$ and $\hat{\tilde{\boldsymbol{\sigma}}}^{\tilde{Y}} \in \mathbb{R}^{C_{\text{out}}}$, which are compared with the target scaling factors $\tilde{\boldsymbol{\mu}}^{\tilde{Y}}$ and $\tilde{\boldsymbol{\sigma}}^{\tilde{Y}}$ using a second loss function. The predicted scaling factors $\hat{\tilde{\boldsymbol{\mu}}}^{\tilde{Y}}$ and $\hat{\tilde{\boldsymbol{\sigma}}}^{\tilde{Y}}$ are then used to unscale the prediction of the Image-Net. The total loss of the model is the sum of both loss functions. The aforementioned is summarized in Figure 5.3, as shown below.



CNN Model
Image Net
Skip Connections
Encoder
Decoder
Scaling Net
$\tilde{\boldsymbol{x}}_i$
Input
$\hat{\tilde{\boldsymbol{\mu}}}^{\tilde{Y}}, \hat{\tilde{\boldsymbol{\sigma}}}^{\tilde{Y}}$
Prediction Scaling Net
Equation (5.5), Equation (5.6)
$\hat{\tilde{\tilde{\boldsymbol{y}}}}_i$
Prediction Image Net
$\hat{\boldsymbol{\mu}}^{\tilde{Y}}, \hat{\boldsymbol{\sigma}}^{\tilde{Y}}$
Equation (5.4)
$\hat{\tilde{\boldsymbol{y}}}_i$
Prediction
Loss 1
Loss 2
Target Scaling Factors
$\tilde{\boldsymbol{\mu}}^{\tilde{Y}}, \tilde{\boldsymbol{\sigma}}^{\tilde{Y}}$
Standardized Target
$\tilde{\tilde{\boldsymbol{y}}}_i$
$\boldsymbol{\mu}^{\tilde{Y}}, \boldsymbol{\sigma}^{\tilde{Y}}$
Equation (5.4)
Target
$\tilde{\boldsymbol{y}}_i$
Equation (5.5), Equation (5.6)

Figure 5.3: Image-Net and Scaling-Net

Chapter 6

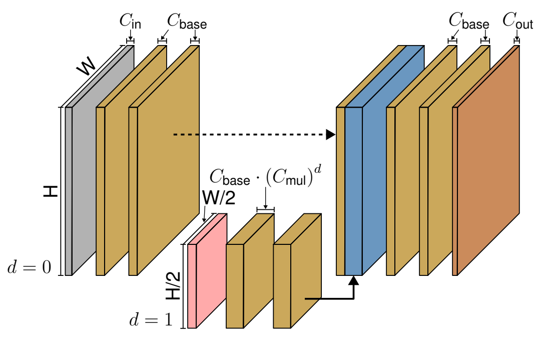
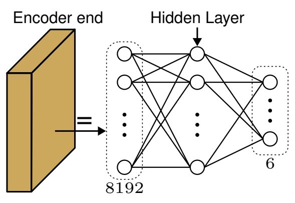
Results

In Chapter 5.3 and Chapter 5.4, two distinct models were introduced. This chapter delves into the specifics of the model and training procedures, followed by a description of metrics employed to evaluate the performance of the model. Particular attention is given to metrics that are relevant for potential real-world applications, which helps to bridge the gap between the cost function used for model training and the actual performance of the model. Subsequently, the results are presented for different scenarios.

6.1 Model and Training Setup

In Figure 3.4, a parametrized 2D U-Net version was presented. To adapt the model to the 3D scenario, an extra image depth dimension D is required. The specific model specifications are outlined in Table 6.1.

Table 6.1: Network specifications

Base Model/Image-Net		
Input channels (C_{in})	11	
Output channels (C_{out})	3	
Image depth (D)	256	
Image height (H)	64	
Image width (W)	32	
Base channels (C_{base})	16	
Model depth (D_{m})	5	
Channel multiplier (C_{mul})	2	
Scaling-Net Extension		
Number of neurons $D \cdot H \cdot W / 2^{3D_{\text{m}}} \cdot C_{\text{base}} \cdot {C_{\text{mul}}}^{D_{\text{m}}}$	8192	
Number of hidden layers	1	
Activation function	LeakyReLU	
Output channels ($2C_{\text{out}}$)	6	

All training images used have dimensions of $256 \times 64 \times 32$ derived from the encoding procedure elaborated in Chapter 5.1. The model predicts the three displacement components from the 11 input channels C_{in}. The model's size is controlled by the number of base channels C_{base}, the channel multiplier factor C_{mul}, and the model depth D_{m}. Likewise, the number of neurons in the encoder extension for the Scaling-Net is also determined by these parameters. Given that the Scaling-Net learns the mean and standard deviation for each output channel of the Image-Net, it possesses $2C_{\text{out}}$ output channels.

The number of parameters is a crucial factor in controlling the model's capacity and its ability to approximate any continuous function, as stated by the universal approximation theorem discussed in Chapter 2.2. The network specifications provided in Table 6.1 determine the number of parameters for the different model types, as shown in Table 6.2.

Table 6.2: Network parameters

Network type	Number of parameters
Base Model/Image-Net	24.3M
Image-Net + Scaling-Net	91.5M

The models are implemented in PyTorch and trained with the Adam optimizer, employing different loss functions, as described in subsequent chapters. Training is performed on an NVIDIA Quadro RTX 4000 with 8 GB memory. To maximize training throughput and optimize the training process for the given hardware a batch size of ten is utilized. A learning rate of 10^{-5} was found to be suitable for training the different models.

The dataset is divided such that 80% is designated for training the models, and the remaining 20% is set aside for validation. Models are trained for a maximum of 1500 epochs, resulting in approximately 21 hours training time for the largest dataset. To expedite the training process, early stopping is employed if the validation cost fails to improve for 500 epochs and mixed precision training is used, which is a technique that was proposed in MICIKEVICIUS et al. (2017) to reduce memory usage and accelerate computations while maintaining model accuracy. The models are trained deterministically by fixing the seed for the random number generators responsible for model parameter initialization and dataset partitioning. Once training is completed, the model parameters are loaded based on the epoch exhibiting the lowest validation cost, which serves to establish the final model performance. A summary of the training specifications can be found in Table 6.3.

Table 6.3: Training specifications

Training parameter	Value
Training dataset size	$0.8N$
Validation dataset size	$0.2N$
Max epochs	1500
Batch size	10
Loss function	see Chapter 6.3
Optimizer	Adam
Learning rate γ	10^{-5}

6.2 Metrics

The proposed metrics aim to evaluate the accuracy of the predicted simulation outcomes. The displacement error DE is computed by calculating the magnitude of the difference between the predicted displacement $\hat{\boldsymbol{u}}$ and the actual displacement $\boldsymbol{u}$, divided by the maximum displacement magnitude for the corresponding simulation. This method ensures equal treatment for all simulations, irrespective of their individual displacement magnitude. The resulting scalar field, which is dependent on the spatial position $\boldsymbol{r}$, can be formulated as:

$$\text{DE}_i(\boldsymbol{r}) = \frac{\|\boldsymbol{u}_i(\boldsymbol{r}) - \hat{\boldsymbol{u}}_i(\boldsymbol{r})\|_2}{\max\limits_{\boldsymbol{r}} \|\boldsymbol{u}_i(\boldsymbol{r})\|_2} \tag{6.1}$$

To derive a value unique to each simulation, the displacement error is spatially averaged according to the simulation's physical domain. The average displacement error (ADE) is calculated by averaging the displacement error over all voxels situated within the physical domain Ω_{phy}. The number of inside voxels n_{invox} can significantly vary between simulations due to the differences in fill ratio, as depicted in Figure 4.1.

$$\text{ADE}_i = \frac{1}{n_{\text{invox},i}} \sum_{j=1}^{n_{\text{invox},i}} \text{DE}_i(\boldsymbol{r}_j) \tag{6.2}$$

where $\boldsymbol{r}_j$ collects all contributions of the inside voxels for the specific simulation i.

Furthermore, the error of the maximum displacement magnitude relative to the maximum displacement magnitude of the specific simulation (DEMAXU) is observed. DEMAXU has significant importance in engineering design decisions.

$$\text{DEMAXU}_i = \frac{\left|\max\limits_{\boldsymbol{r}} \|\boldsymbol{u}_i(\boldsymbol{r})\|_2 - \max\limits_{\boldsymbol{r}} \|\hat{\boldsymbol{u}}_i(\boldsymbol{r})\|_2\right|}{\max\limits_{\boldsymbol{r}} \|\boldsymbol{u}_i(\boldsymbol{r})\|_2} \tag{6.3}$$

To assess the overall predictive performance, the average and max of both metrics can be computed for a given dataset.

6.3 Comparison of Loss Functions

Initially, the basic U-Net model explained in Chapter 5.3 is used to investigate various loss functions. As outlined in Chapter 2.3.1, squared loss L_2 and absolute loss L_1 are commonly used for regression problems. L_1 is more robust to outliers, whereas L_2 offers better learning properties. However, when training the displacement surrogate model, both loss functions encounter a conceptional issue because all errors are treated absolutely. The tolerable absolute difference varies significantly depending on the magnitude of the simulation results. Smaller absolute errors are acceptable for smaller displacement results, while larger absolute errors are permissible for larger displacement results.

To address this issue, a weighted version of the squared error is proposed. For the base model the $L_{2,\text{w}}$ for the simulation i is defined as:

$$L_{2,\text{w},i} = \frac{1}{C_{\text{out}}} \sum_{j=1}^{C_{\text{out}}} \frac{1}{n_{\text{invox}}} w_{ij} \sum_{k=1}^{n_{\text{invox}}} (\tilde{Y}_{ijk}^{\text{masked}} - \hat{\tilde{Y}}_{ijk}^{\text{masked}})^2 \tag{6.4}$$

where the model prediction inside the physical domain $\hat{\tilde{Y}}_{ijk}^{\text{masked}}$ is compared to the target value $\tilde{Y}_{ijk}^{\text{masked}}$, and is subject to simulation- and channel-dependent weighting. The weights $\boldsymbol{w}$ are defined as:

$$w_{ij} = \frac{\frac{1}{N}\sum_{k=1}^{N} \max_{\boldsymbol{r}} |u_{kj}(\boldsymbol{r})|}{\max_{\boldsymbol{r}} |u_{ij}(\boldsymbol{r})|} \tag{6.5}$$

The numerator represents the simulation-wise mean of the maximum magnitude of the unscaled displacement component, whereas the denominator is the simulation- and channel-wise maximum magnitude of the unscaled displacement component. The denominator ensures that more importance is given to simulations with small displacement results, while the numerator ensures that the average weight is one.

L_1 and L_2, as defined in Equation (2.4) and Equation (2.3), can be easily adapted to this specific case by ensuring that the sum over n_{feat} collects all the contributions for each channel within the physical domain Ω_{phy}.

Figure 6.1 shows the standard deviation of displacement components for a dataset consisting of $N = 100$ observations. The histogram is further smoothed using a kernel density estimator and reveals that the distribution is highly skewed and contains numerous outliers. This skewed feature becomes increasingly noticeable in the larger dataset of $N = 800$, as shown in Figure 6.2.

It is worth noting that the values presented in the figures are unscaled. Applying a linear transformation through standardization, as shown in Equation (5.1), would only alter the axis values, keeping the visual appearance of the images unaltered.

The evolution of the mean and max for both metrics, namely ADE and DEMAXU, during training is depicted in Figure 6.3 for the validation set and a dataset size $N = 100$. The L_2 loss function exhibit significant oscillations, likely due to its sensitivity to outliers. Although the L_1 demonstrates more favorable behavior, it is outperformed by the proposed weighted loss $L_{2,\text{w}}$ for all metrics. Moreover, the weighted loss function requires less training time as the training stops earlier due to stagnation of the validation cost. Due to these favorable outcomes, the subsequent chapters use the weighted loss $L_{2,\text{w}}$. The best model score is achieved at epoch 276, with a mean average displacement error of 12.2% and a maximum of 24.7%. On average, the error of the maximum displacement is 38.3%, while in the worst-case scenario it reaches 86.9%.

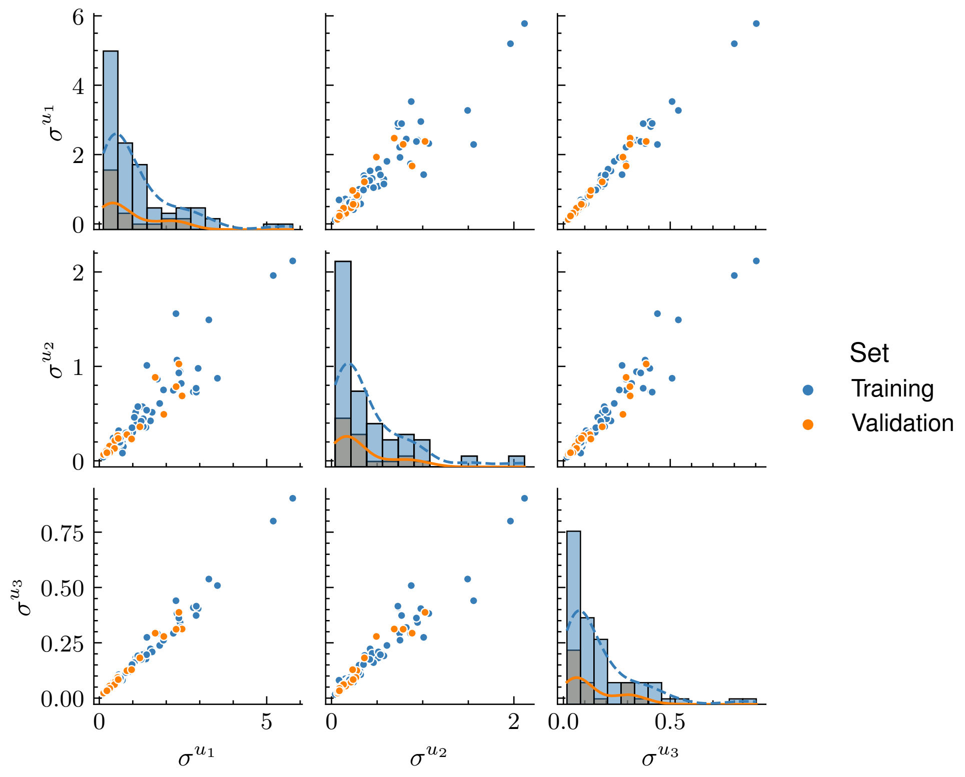


Figure 6.1: Standard deviations of the displacement components for $N = 100$

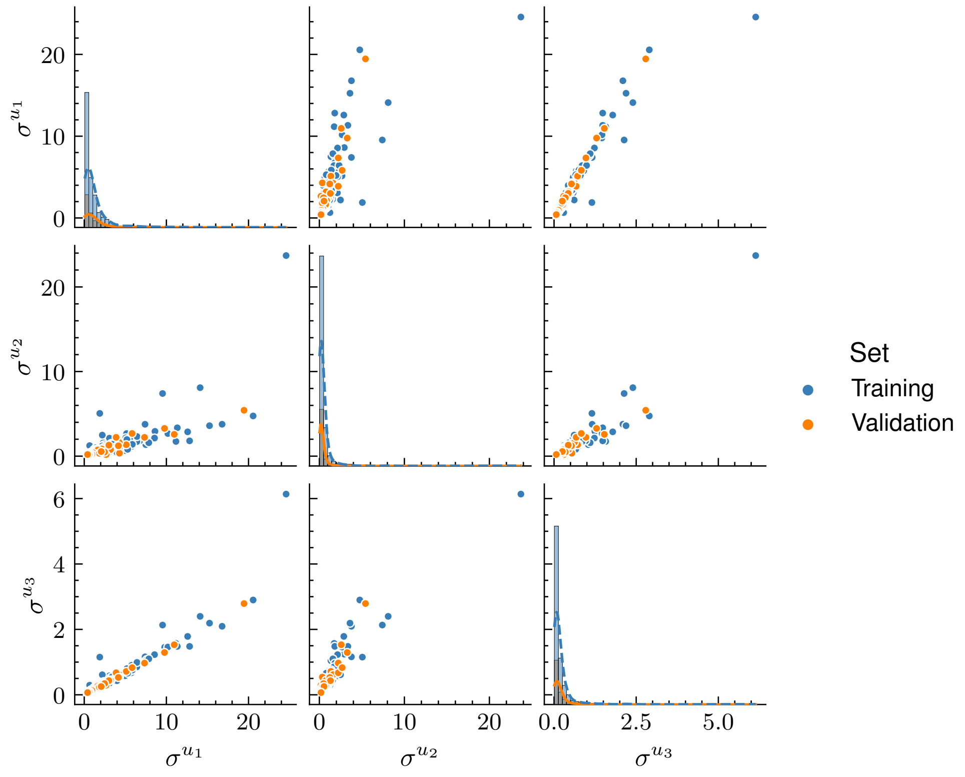
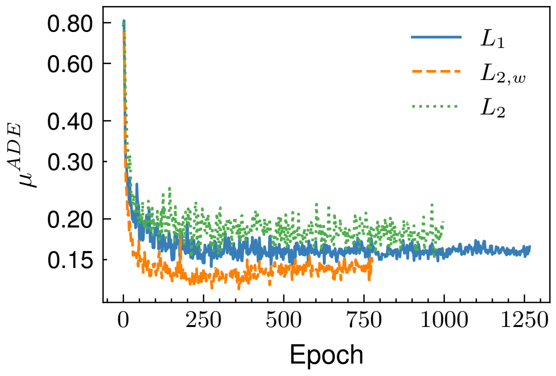
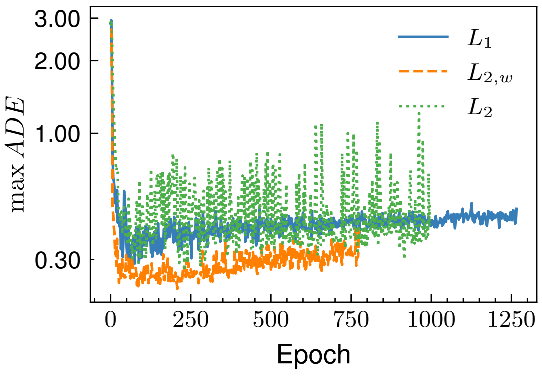


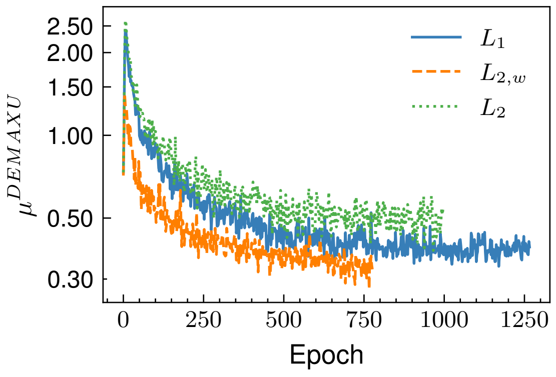
Figure 6.2: Standard deviations of the displacement components for $N = 800$



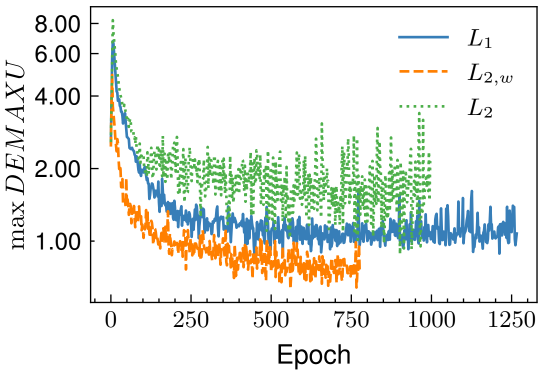
(a) Mean average displacement error



(b) Max average displacement error



(c) Mean error of max displacement



(d) Max error of max displacement

Figure 6.3: Development of the metrics for the validation set during training with $N = 100$. A shorter time series means that the early stopping criteria was reached earlier.

6.4 Generalization Capabilities of the Base Model

Chapter 6.3 pinpointed a suitable loss function for the given problem, yet it revealed an undesirably large error. The subsequent subchapters discuss the base model's generalization abilities.

6.4.1 Increasing the Dataset Size

To enhance the model's performance, the dataset size is doubled incrementally until it reaches a size of 800. This larger dataset provides the model with more support points for interpolation. However, this advantage may be offset by increased skewness of the dataset, as illustrated in Figure 6.1 and Figure 6.2, which highlights an increase in the interpolation space.

The evolution of the simulation means of the two metrics ADE and DEMAXU for the validation set is presented in Figure 6.4. The metrics display no substantial improvement between $N = 100$ and $N = 400$. When the dataset size is increased to 800, there is a slight improvement in the mean of the average displacement error and a significant improvement in the mean of the error of the maximum displacement. Despite these improvements, the validation errors remain high due to the discrepancy between training and validation cost, which is demonstrates in Figure 6.5.

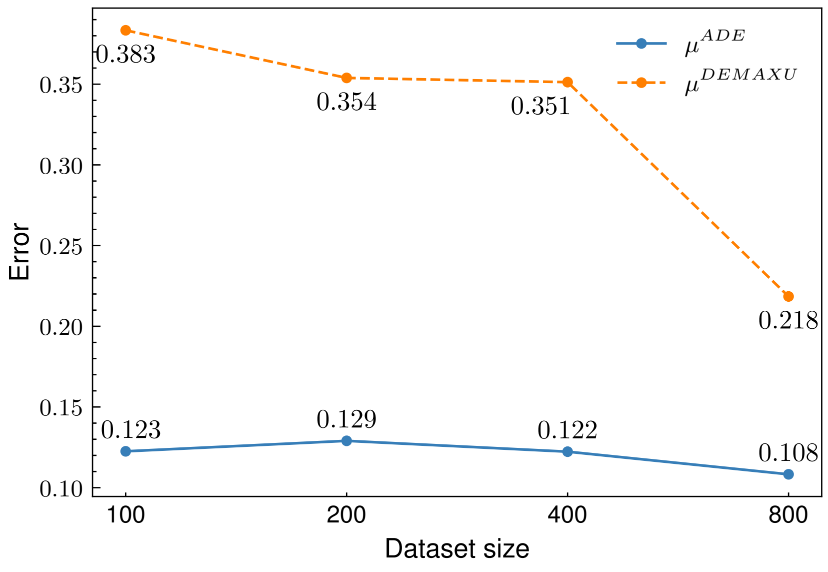


Figure 6.4: Evolution of the metric means when increasing the dataset size N

During the training process, the training cost and associated metrics steadily decrease. However, the validation cost remains high, which may indicate that the model is overfitting to the training data and struggling to generalize effectively.

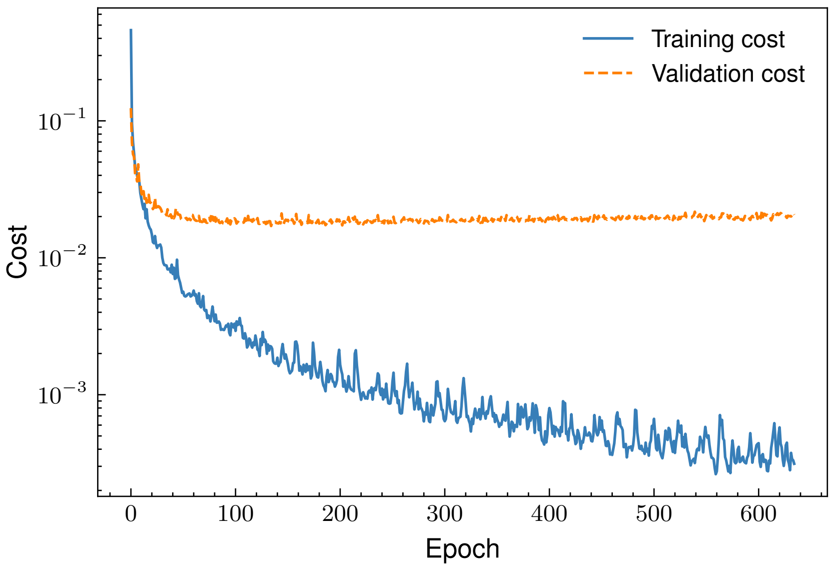
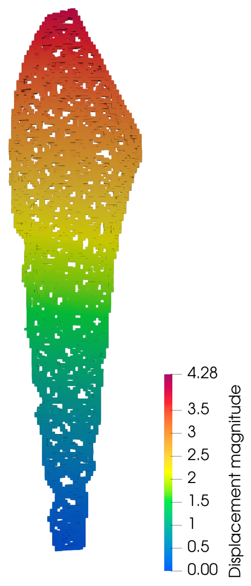
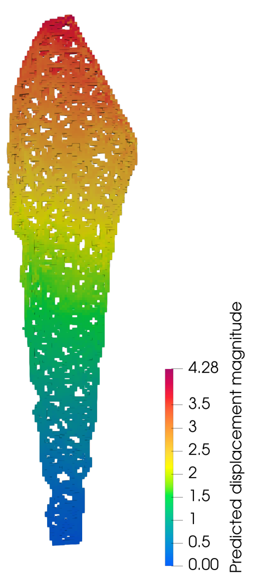


Figure 6.5: Development of training and validation cost during training of the base model with a dataset size $N = 800$

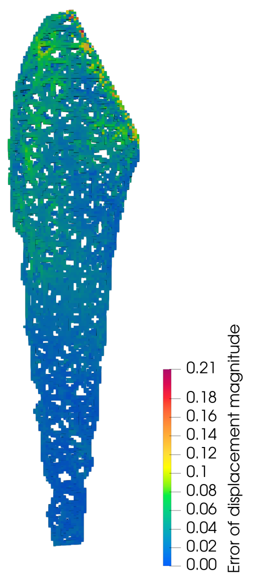
In Figure 6.6, the expected accuracy of the model prediction is illustrated. The data ranges of the model prediction and target are rescaled to the largest range to emphasize the differences. The right column displays the error of the displacement magnitude, as defined by Equation (6.1). The first row of the figure depicts the best-case scenario, showing the results for the simulation with the smallest average error within the validation set. Only a few voxels exhibit significant error, and the target and the prediction show good agreement. The second row illustrates the worst-case scenario based on the simulation within the validation set with the highest average error. In this case, the prediction and target data do not agree, with large regions showing significant errors, and some points being off by almost 100%. To mitigate this problem, the next chapter will investigate the use of weight decay as a regularization method.



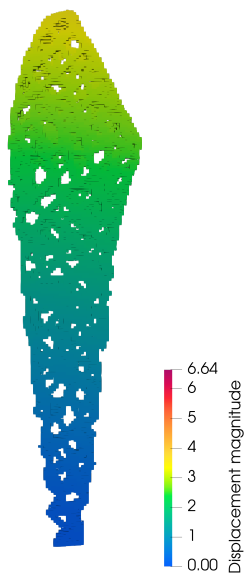
(a) Best-case target



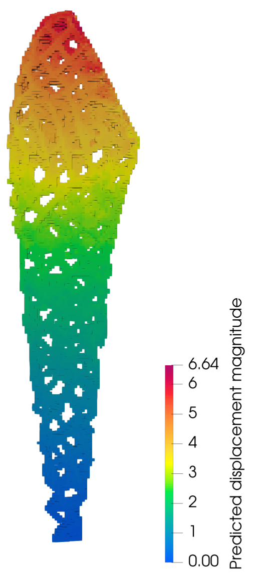
(b) Best-case prediction



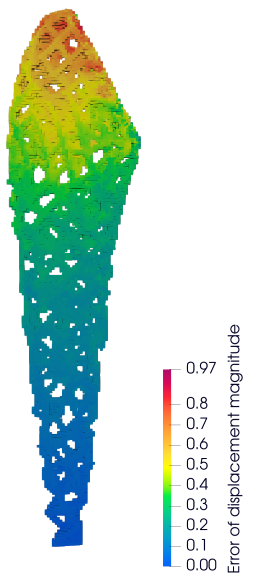
(c) Best-case error



(d) Worst-case target



(e) Worst-case prediction



(f) Worst-case error

Figure 6.6: Comparison of the expected displacement error of the base model in the best- and worst-case scenario for a dataset size $N = 800$

6.4.2 Weight Decay

Weight decay is a regularization technique that penalizes large model parameters, aiming to reduce overfitting by encouraging the model to learn simpler patterns. It is implemented as part of the optimizer by modifying the gradient with respect to the model parameters, as explained in Chapter 2.3.3. In Figure 6.7, the impact of weight decay on the two metrics is demonstrated for a dataset size of 800, with values ranging from 10^{-6} to 10^{-2}. The results show that a small weight decay of $\lambda = 10^{-5}$ improves the error of the maximum displacement but gradually worsens the mean of the average displacement error. According to THUEREY et al. (2021), convolutional neural networks CNNs usually require minimal regularization. Thus, it is plausible that weight decay did not improve the overall performance of the model.

Furthermore, the high number of outliers in the dataset may also contribute to the discrepancy between training and validation cost. The upcoming chapter will explore the effect of outliers in further detail.

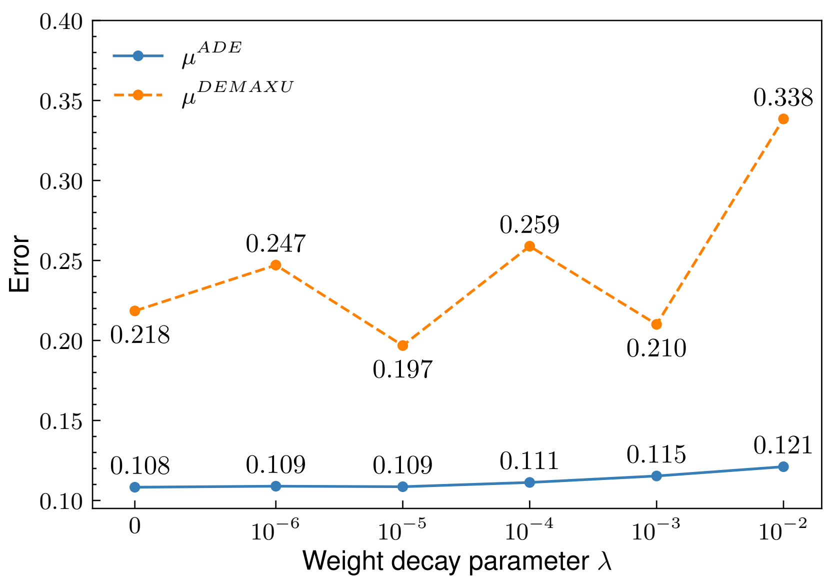


Figure 6.7: Evolution of the metric means for different weight decay parameters and $N = 800$

6.4.3 Removing Outliers

As outlined in Chapter 6.3, the dataset exhibits significant skewness and contains numerous outliers, which may hinder the model's ability to generalize. To examine the impact of outliers, an outlier-free version of the dataset with $N = 800$ was generated by removing data points with z-scores greater than three. The z-score z^{score} is equivalent to $\tilde{z}$ in Equation (5.1), therefore standardized values larger than three are eliminated. The mean and standard deviation are computed channel-wise based on the voxels inside the physical domain, and the maximum value of each displacement component is used as z.

$$z_{ij}^{score} = \frac{\max Y_{ijk}^{\text{masked}} - \mu_j^Y}{\sigma_j^Y} \tag{6.6}$$

This procedure is performed recursively until it converges, as it alters the statistical properties of the dataset. The resulting outlier-free dataset has a size of $N = 560$, and its standard deviation distribution is illustrated in 6.8.

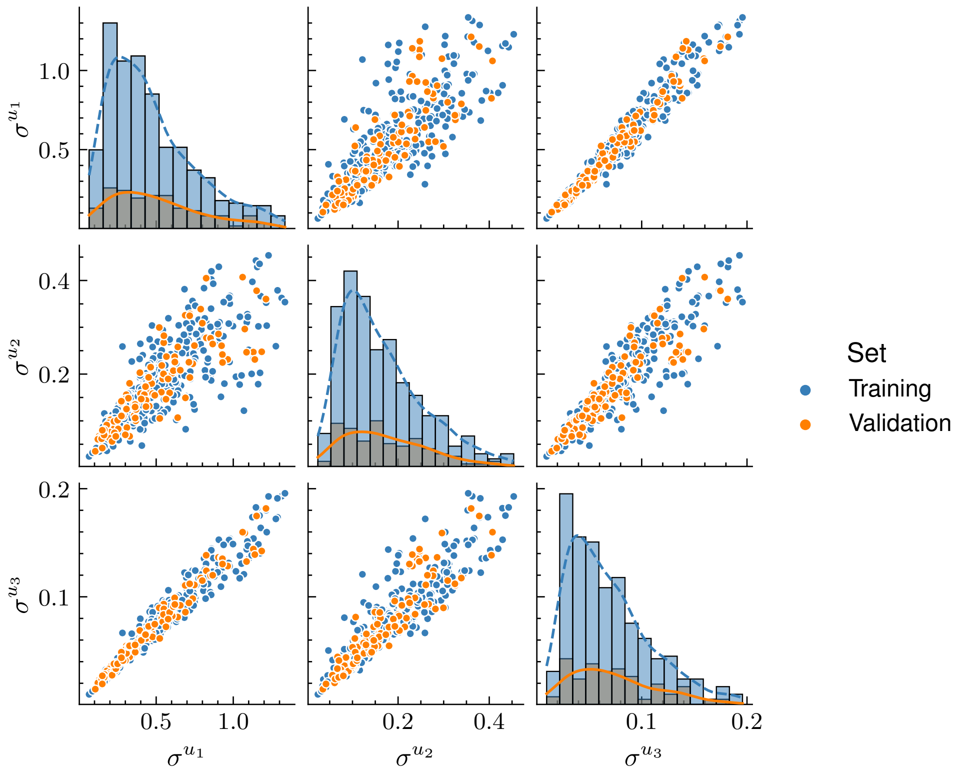
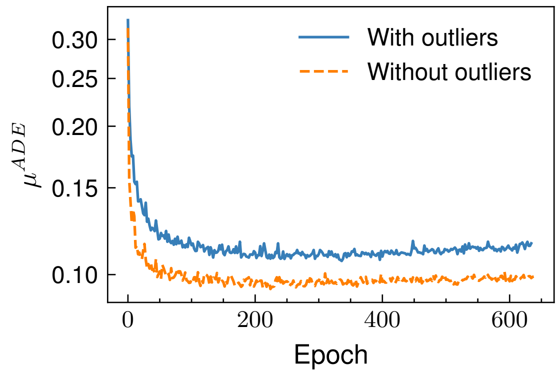
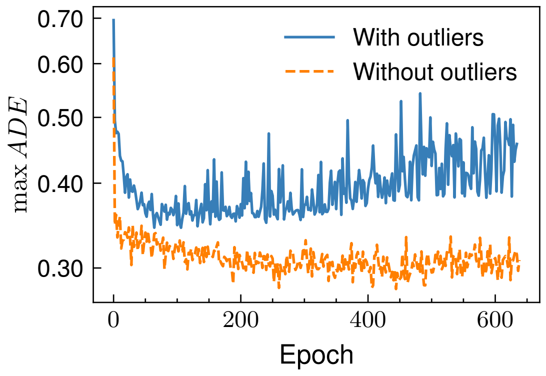


Figure 6.8: Standard deviations of the displacement components for the outlier-free dataset with $N = 560$

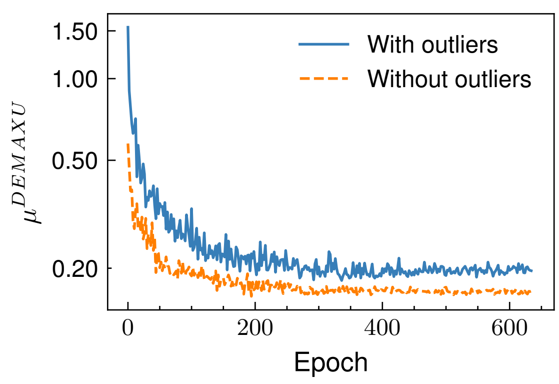
A comparison between the original dataset with $N = 800$ and its outlier-free version is shown in Figure 6.9, presenting the mean and maximum of both metrics. Despite a slight improvement in both metrics, there is no significant impact on the model performance. The average displacement error decreases slightly from 10.8% to 9.5%, and the average error of the maximum displacement decreases from 21.8% to 17.6%.



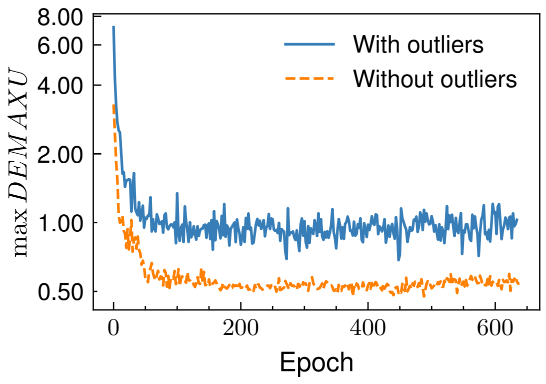
(a) Mean average displacement error



(b) Max average displacement error



(c) Mean error of max displacement



(d) Max error of max displacement

Figure 6.9: Comparison of the metrics development for the validation set during training for the dataset with $N = 800$ and its outlier-free version with $N = 560$

6.5 Image-Net: Displacements

The second type of model discussed, as explained in Chapter 5.4, is composed of two network parts: the Image-Net and the Scaling-Net. The idea is to split the image-to-image regression task into two distinct parts. Firstly, the Image-Net learns the distribution of values within standardized images, and secondly, the scaling reverts the standardization with a learned linear transformation.

To evaluate the performance of the Image-Net, the metric development is observed as the dataset size increases, assuming perfect rescaling. This implies no errors from the Scaling-Net and therefore sets $\hat{\tilde{\boldsymbol{\mu}}}^{\tilde{Y}}$ and $\hat{\tilde{\boldsymbol{\sigma}}}^{\tilde{Y}}$, equal to the true scaling factors $\tilde{\boldsymbol{\mu}}^{\tilde{Y}}$ and $\tilde{\boldsymbol{\sigma}}^{\tilde{Y}}$. Since the standardization of each image already treats all simulations equally, the L_2 loss function is used. The evolution of the metric means is illustrated in Figure 6.10, indicating that errors are minimal even for a dataset size of 100 and continue to decrease steadily with increasing dataset size. However, there is a significant discrepancy between the training and validation cost even for the largest dataset with $N = 800$, as demonstrated in Figure 6.11.

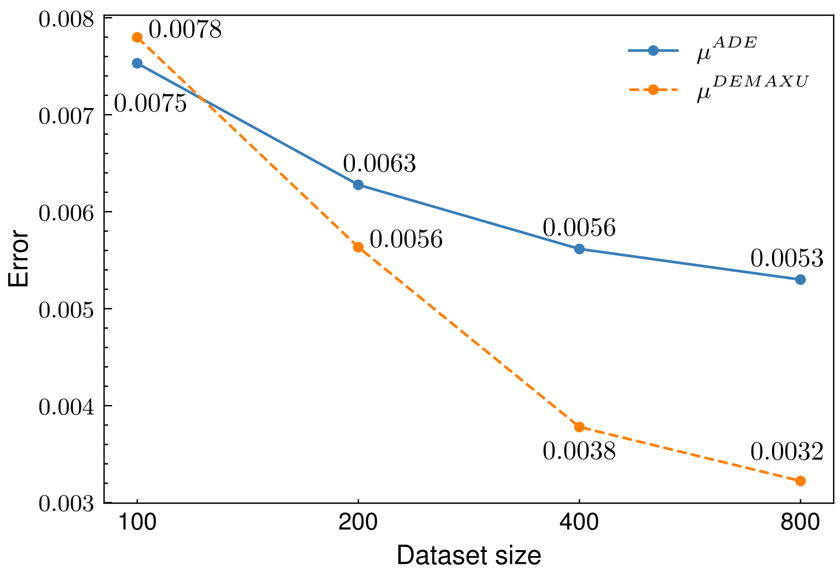


Figure 6.10: Evolution of the metric means with increasing dataset size N for the Image-Net

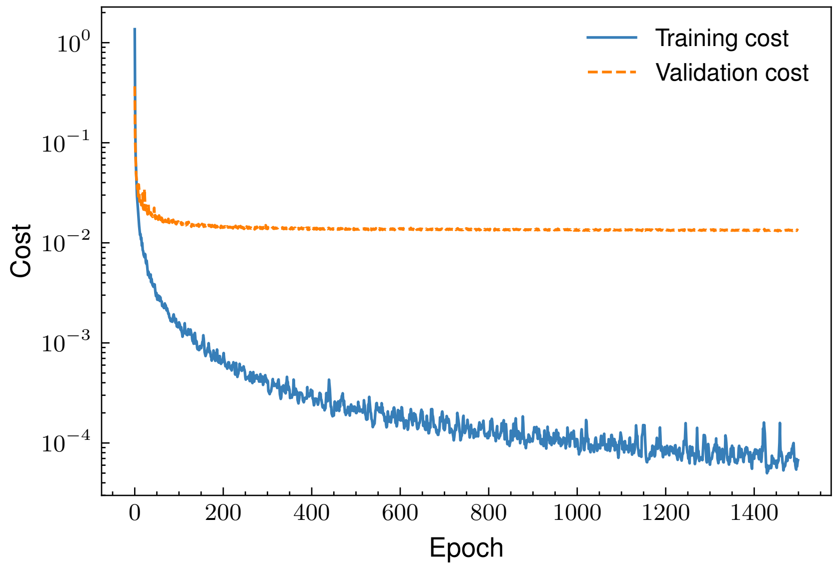


Figure 6.11: Development of training and validation cost during training of the Image-Net with a dataset size $N = 800$

6.6 Image-Net and Scaling-Net

As discussed in Chapter 5.4, the loss function for the Image-Net and Scaling-Net model is composed of two parts: one comparing the images and the other the scaling factors. The total loss is the sum of these two parts. Following similar arguments as in Chapter 6.3 the weighted loss $L_{2,\mathrm{w}}$ is used for the scaling factors and a normal L_2 for the image part as stated in Chapter 6.5.

The performance of the Image-Net and Scaling-Net model is compared with the base model using the average values of the metrics. As depicted in Figure 6.12, the combined model outperforms the base model in terms of the maximum displacement error, particularly for small dataset sizes. However, this advantage diminishes with increasing dataset size. Additionally, the average displacement error is slightly improved. For a dataset size $N = 800$, the average displacement error is 10.8% for the base model and 9.7% for the combined model, while the maximum displacement error is 21.8% and 15.5%, respectively. The addition of weight decay to the combined model does not lead to improved metrics, similar to the case of the base model.

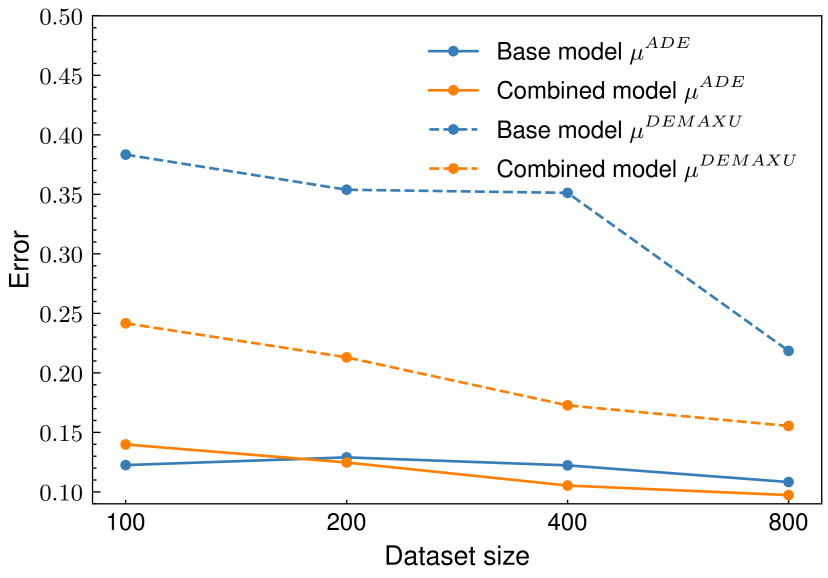


Figure 6.12: Evolution of metric means with increasing dataset size N: Base Model vs Image-Net and Scaling-Net

6.7 Image-Net: Extension to Stresses

This chapter presents the performance of the Image-Net when learning stresses in addition to displacements. Learning stresses is an even more complex task for the studied geometries, as they lead to stress concentrations due to geometric effects. The presented results do not account for rescaling errors, which were found to dominate the total error of the combined model in Chapter 6.6, but assume perfect rescaling. This serves as a theoretical example, which could become relevant if a more sophisticated rescaling method would be developed.

The model's architecture remains unchanged, with only the number of output channels C_{out} increasing to nine. This slightly changes the number of model parameters, since it alters the last convolution layer. However, the rounded number of parameters nevertheless stays at 24.3M.

Similarly to the error of the maximum displacement, the error of the maximum von Mises stress SEMAXVM can be defined as:

$$\text{SEMAXVM}_i = \frac{\left| \max_{\boldsymbol{r}} \|\boldsymbol{\sigma}_i(\boldsymbol{r})\|_{VM} - \max_{\boldsymbol{r}} \|\hat{\boldsymbol{\sigma}}_i(\boldsymbol{r})\|_{VM} \right|}{\max_{\boldsymbol{r}} \|\boldsymbol{\sigma}_i(\boldsymbol{r})\|_{VM}} \tag{6.7}$$

where $\|\cdots\|_{VM}$ denotes the computation of the von Mises stress. The stress prediction of the model is denoted as $\hat{\boldsymbol{\sigma}}$. As stress is a local phenomenon in this specific case, reporting the spatial average of the error would be misleading, and hence is not done. Figure 6.13 shows the evolution of the mean values of the metrics for the validation set during training. The best epoch value is reached close to the end of training, indicating that the model may benefit from further training time and may reach slightly better performance. Since the learning task becomes more complex, the average displacement error increases from 0.53% to 1.42%, while the error of the maximum displacement increase from 0.32% to 1.12% compared to the displacement Image-Net from Chapter 6.5. The average error of the maximum von Mises stress reaches 12.65%.

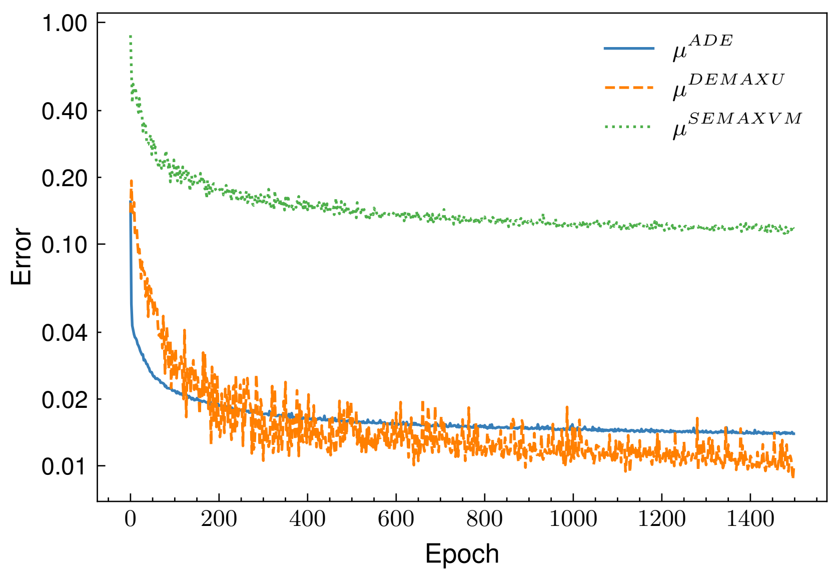


Figure 6.13: Evolution of metric means for the Image-Net during training with a dataset size $N = 800$

Chapter 7

Conclusion and Outlook

The aim of this thesis was to investigate deep learning-based surrogate models for Linear Elasticity. In contrast to previous research, which mainly focused on 2D Computational Fluid Dynamics (CFD) or thermal problems, the following new contributions were made:

- Expanded surrogate models to encompass 3D Linear Elasticity.
- Encoded the image-to-image learning task with a binary mask resulting in strong variations of the simulation domain and hence a changing learning domain for the network.
- Developed a weighted loss function to enhance the model performance for predicting the displacement results of the simulations.
- Investigated the performance of a basic U-Net model for predicting the displacement results of the simulation.
- Examined the impact of outliers on the model's accuracy.
- Proposed a splitting of the learning task into two simpler parts: learning standardized images (Image-Net) and associated scaling factors (Scaling-Net) to manage large datasets containing numerous outliers.

The task of learning the displacement solution for the 3D linear elastic problem posed significant challenges for the investigated CNN models. Formulating the problem as an image-to-image task based on binary masks resulted in a large discrepancy between the training and validation cost, leading to unacceptable errors. The presence of numerous outliers in the dataset resulted in slightly worse performance compared to a dataset without outliers. Only the Image-Net model performed well, however only under the assumption of perfect rescaling, which requires the development of new rescaling methods. This model also showed a significant gap between the training and validation cost. To improve performance on unseen data, it may be beneficial to use a larger dataset or more sophisticated sampling strategies that increase the coverage of the sampling space. These steps could be taken to further develop deep learning-based surrogate models for Linear Elasticity based on the problem formulation and encoding process, used in this study.

However, it may be worthwhile to explore alternative surrogate modeling techniques, such as the following:

- One approach is to utilize various encoding techniques for geometric data. For example, signed distance functions can be employed to increase the information density of mask images, as demonstrated for airfoils in BHATNAGAR et al. (2019). Another option is geometric encoding using modes that describe the shape based on Principal Component Analysis (PCA) as shown in HEIMANN and MEINZER (2009), or learned representations such as those presented in ZUO et al. (2022).

- Another possibility is to switch to unsupervised learning techniques based on Physics-informed Neural Networks (PINNs), where the residual of the PDE is used as a cost function. The necessary derivatives for the residual can be computed through finite differences, as demonstrated in WANDEL et al. (2020) and WANDEL et al. (2021), or by directly utilizing the NN, as outlined in KOLLMANNSBERGER et al. (2021).

- Exploring operator learning based on the universal approximation theorem of continuous operators, as shown in T. CHEN and CHEN (1995), may also prove fruitful. Prominent network architectures in this field include the DeepONet (LU, JIN, et al., 2021) and the Fourier Feature Networks (TANCIK et al., 2020). A comparison of these two variants is available in LU, MENG, et al. (2021).

Bibliography

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer Normalization. https://doi.org/10.48550/ARXIV.1607.06450

Bhatnagar, S., Afshar, Y., Pan, S., Duraisamy, K., & Kaushik, S. (2019). Prediction of aerodynamic flow fields using convolutional neural networks. *Computational Mechanics*, *64*(2), 525–545. https://doi.org/10.1007/s00466-019-01740-0

Bishop, C. M. (1995). *Neural networks for pattern recognition*. Clarendon Press ; Oxford University Press. ISBN:978-0-19-853864-6

Chen, L.-W., Cakal, B. A., Hu, X., & Thuerey, N. (2020). Numerical investigation of minimum drag profiles in laminar flow using deep learning surrogates. *arXiv: Fluid Dynamics*. https://doi.org/10.1017/jfm.2021.398

Chen, L., & Thuerey, N. (2022). Towards high-accuracy deep learning inference of compressible flows over aerofoils. *Computers & Fluids*, *250*, 105707–105707. https://doi.org/10.1016/j.compfluid.2022.105707

Chen, T., & Chen, H. (1995). Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, *6*(4), 911–917. https://doi.org/10.1109/72.392253

Chen, X., Chen, X., Zhao, X., Gong, Z., Zhang, J., Zhou, W., Chen, X., & Yao, W. (2021). A deep neural network surrogate modeling benchmark for temperature field prediction of heat source layout. *Science China-physics Mechanics & Astronomy*, *64*(11). https://doi.org/10.1007/s11433-021-1755-6

Dubey, S. R., Singh, S. K., & Chaudhuri, B. B. (2021). Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark. https://doi.org/10.48550/ARXIV.2109.14545

Duru, C., Alemdar, H., & Baran, O. U. (2022). A deep learning approach for the transonic flow field predictions around airfoils. *Computers & Fluids*, *236*, 105312. https://doi.org/10.1016/j.compfluid.2022.105312

Düster, A., Rank, E., & Szabó, B. A. (2017). The p-Version of the Finite Element and Finite Cell Methods, 1–35. https://doi.org/10.1002/9781119176817.ecm2003g

Godbole, V., Dahl, G. E., Gilmer, J., Shallue, C. J., & Nado, Z. (2023). Deep Learning Tuning Playbook. https://github.com/google-research/tuning_playbook

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org

Griewank, A., & Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition* (Second). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898717761

Heimann, T., & Meinzer, H.-P. (2009). Statistical shape models for 3D medical image segmentation: A review. *Medical Image Analysis*, *13*(4), 543–563. https://doi.org/10.1016/j.media.2009.05.004

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv: Learning*. https://doi.org/10.48550/arXiv.1505.04597

Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*. https://doi.org/10.48550/arXiv.1412.6980

Kollmannsberger, S., D'Angella, D., Jokeit, M., & Herrmann, L. (2021). *Deep Learning in Computational Mechanics: An Introductory Course*. Springer. http://dx.doi.org/10.1007/978-3-030-76587-3

LeCun, Y. A., Bottou, L., Orr, G. B., & Müller, K.-R. (2012). Efficient BackProp. In G. Montavon, G. B. Orr, & K.-R. Müller (Eds.), *Neural Networks: Tricks of the Trade* (pp. 9–48, Vol. 7700). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35289-8_3

Lu, L., Jin, P., Pang, G., Zhang, Z., & Karniadakis, G. E. (2021). Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, *3*(3), 218–229. https://doi.org/10.1038/s42256-021-00302-5

Lu, L., Meng, X., Cai, S., Mao, Z., Goswami, S., Zhang, Z., & Karniadakis, G. E. (2021). A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data. https://doi.org/10.48550/ARXIV.2111.05512

Mallik, W., Farvolden, N., Jelovica, J., & Jaiman, R. K. (2022). Deep convolutional neural network for shape optimization using level-set approach. https://doi.org/10.48550/ARXIV.2201.06210

Messner, M. C. (2020). Convolutional Neural Network Surrogate Models for the Mechanical Properties of Periodic Structures. *Journal of Mechanical Design*, *142*(2), 024503. https://doi.org/10.1115/1.4045040

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., & Wu, H. (2017). Mixed Precision Training. https://doi.org/10.48550/ARXIV.1710.03740

Navidi, W. C. (2011). *Statistics for engineers and scientists* (3rd ed). McGraw-Hill. ISBN: 978-0-07-337633-2

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press. http://neuralnetworksanddeeplearning.com/

Odena, A., Dumoulin, V., & Olah, C. (2016). Deconvolution and Checkerboard Artifacts. *Distill*, *1*(10). https://doi.org/10.23915/distill.00003

Peng, J.-Z., Liu, X., Aubry, N., Chen, Z., & Wu, W.-T. (2020). Data-Driven Modeling of Geometry-Adaptive Steady Heat Transfer based on Convolutional Neural Networks: Heat Conduction. https://doi.org/10.48550/ARXIV.2010.03854

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. In N. Navab, J. Hornegger, W. M. Wells, &

A. F. Frangi (Eds.), *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015* (pp. 234–241, Vol. 9351). Springer International Publishing. https://doi.org/10.1007/978-3-319-24574-4_28

Ruess, M., Schillinger, D., Bazilevs, Y., Varduhn, V., & Rank, E. (2013). Weakly enforced essential boundary conditions for NURBS-embedded and trimmed NURBS geometries on the basis of the finite cell method. *International Journal for Numerical Methods in Engineering*, *95*(10), 811–846. https://doi.org/10.1002/nme.4522

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. https://doi.org/10.5555/2627435.2670313

Tancik, M., Srinivasan, P. P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., Ramamoorthi, R., Barron, J. T., & Ng, R. (2020). Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains. https://doi.org/10.48550/ARXIV.2006.10739

Thuerey, N., Holl, P., Mueller, M., Schnell, P., Trost, F., & Um, K. (2021). *Physics-based Deep Learning*. WWW. https://physicsbaseddeeplearning.org

Thuerey, N., Weissenow, K., Prantl, L., & Hu, X. (2020). Deep Learning Methods for Reynolds-Averaged Navier–Stokes Simulations of Airfoil Flows. *AIAA Journal*, *58*(1), 25–36. https://doi.org/10.2514/1.j058291

Ulyanov, D., Vedaldi, A., & Lempitsky, V. (2016). Instance Normalization: The Missing Ingredient for Fast Stylization. https://doi.org/10.48550/ARXIV.1607.08022

Um, K., Brand, R., Yun, Fei, Holl, P., & Thuerey, N. (2020). Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers. https://doi.org/10.48550/ARXIV.2007.00016

Wandel, N., Weinmann, M., & Klein, R. (2020). Learning Incompressible Fluid Dynamics from Scratch – Towards Fast, Differentiable Fluid Models that Generalize. https://doi.org/10.48550/ARXIV.2006.08762

Wandel, N., Weinmann, M., & Klein, R. (2021). Teaching the incompressible Navier–Stokes equations to fast neural surrogate models in three dimensions. *Physics of Fluids*, *33*(4), 047117. https://doi.org/10.1063/5.0047428

Wang, Q., Ma, Y., Zhao, K., & Tian, Y. (2020). A Comprehensive Survey of Loss Functions in Machine Learning. *Annals of Data Science*, 1–26. https://doi.org/10.1007/s40745-020-00253-5

Zuo, K., Bu, S., Zhang, W., Hu, J., Ye, Z., & Yuan, X. (2022). Fast sparse flow field prediction around airfoils via multi-head perceptron based deep learning architecture. https://doi.org/10.48550/ARXIV.2207.00936

Declaration

I hereby affirm that I have independently written the thesis submitted by me and have not used any sources or aids other than those indicated.

München, den 31.03.2023 [illegible]

Location, Date, Signature