Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Artificial Neural Network Approach to Design Optimization

Yudin Yehor

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Artificial Neural Network Approach to Design Optimization

Author:	Yudin Yehor
1st examiner:	Prof. Dr. Hans-Joachim Bungartz
2nd examiner:	PD Dr. Slobodan Ilic
Assistant advisors:	Stefan Gavranovic, Sergey Zakharov
Submission Date:	July 25th, 2018

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

July 25th, 2018 Yudin Yehor

Acknowledgments

First of all, I would like to thank the chair of Scientific Computing for the opportunity to study at the Technical University of Munich. I would also like to thank Siemens Corporate Technology for providing me with an opportunity to conduct this work in collaboration with them. Furthermore, I would like to thank Univ.-Prof. Dr. Hans-Joachim Bungartz and PD Dr. Slobodan Ilic for accepting to be examiners of my work. I am very grateful to Stefan Gavranovic and Sergey Zakharov for their constant guidance and research directions throughout the thesis. Finally, I would like to thank my family and friends for their support during my studies.

Abstract

Design Optimization is a perspective engineering approach which can find solutions for many engineering problems under multiple constraints in a non-analytic manner. The existing algorithms are computationally expensive and could be accelerated by finding approximate results using data-driven methods. This work proposes an approach to predict the material layout of elements with minimal compliance using a convolutional neural network model. Furthermore, the pre-processing and post-processing approaches for the data are proposed and the influence of such approaches on the model is estimated. The work proposes the corresponding CNN architecture. During the work, a framework for both easy performance analysis and the generation of a resulting dataset for future topology optimization acceleration is implemented using Python and Tensorflow. The paper analyzes the ability of the model to interpolate various parameters and its capacity to be trained on small datasets. Both 2D and 3D cases are considered and an approach to use the model for TO performance improvement is proposed and examined. Finally, it analyzes the possibility of using model-produced results as an educated guess for further TO process and estimates the speedup.

Keywords: Topology Optimization, Applied Machine Learning, Deep Learning

Contents

Outline of the Thesis

Part I: Introduction

CHAPTER 1: INTRODUCTION

This chapter presents an overview of the thesis and its motivation. It discusses the challenges of the topology optimization and the possibilities to improve its performance. It considers the Machine Learning approaches to this problem and discusses the scope of applicability of Deep Learning algorithms to topology optimization.

Part II: Theory

CHAPTER 2: TOPOLOGY OPTIMIZATION

In the first chapter of this part, the Design Optimization field is introduced, as well as the range of its applications and the models used for them. It describes what the approaches for the algorithms are and the challenges in the field.

CHAPTER 3: ARTIFICIAL NEURAL NETWORKS

The second chapter of the theory part introduces the reader to the field of artificial neural networks. The chapter discusses the principles behind the approach, the challenges during the design process and techniques to tackle them.

Part III: Methodology

CHAPTER 4: METHOD AND MODEL

This chapter describes the steps that were done to design the inference system and evaluate its performance. It gives the description of the aspects of the architecture, the training and testing processes and their implementation.

CHAPTER 5: DATA PREPARATION

The fifth chapter explains the parameters that define topology optimization problems, ways to generate samples and how to use them to create training datasets for the inference model. It also discusses the issues concerned with data representations suitable for the ANN models.

CHAPTER 6: OUT-PUT POST-PROCESSING

In this chapter, the requirements on the model outputs are discussed and issues concerned with them are explained. The chapter also proposes a data post-processing algorithm to enforce meeting the requirements on resulting layouts.

Part IV: Results and Conclusions

Part I.

Introduction

1. Introduction

This chapter discusses the principles and application scope of topology optimization (TO) algorithms and explains the performance challenges of this method. It presents an approach to improve the performance of the method based on an artificial neural network (ANN), justifies the applicability and novelty of such an approach and demonstrates ways it can help improve the performance of TO.

1.1. Motivation

Topology optimization is a mathematical method to optimize material layout within a given design space for a given set of constraints, including loads, boundary conditions (BC), etc., in order to maximize the performance of the system. It is used for manufacturing in areas where the weight of elements is crucial, or it is important how the material is used; for instance, in aerospace engineering[12]. TO is essentially a computationally complex problem and various approaches forwards improving its performance are being researched[22].

TO is an iterative process, during which for every iteration the Finite Element Analysis (FEA) is performed in order to assure that the result is correct and corresponds to the physics of the problem, and to derive the functional gradient. This step is the most expensive step of all and can be internally optimized to increase the speed of computation[13]. However, the other aspect of performance tuning is reducing the number of optimization iterations made.

One of the ways to achieve this is to start optimization with some educated guess of the material layout. This is instead of the default way, which is initializing the whole domain covered with the material. In the ideal case, an initial material layout for a TO problem could be initialized with an approximate solution. With that, the TO process will require only a couple iterations to converge under the chosen criteria and multiple initial steps of analysis will be eliminated. In most cases, the first step of the optimization drastically changes the layout of the material and leads to a very coarse solution. With that, the objective function significantly decreases, but the value still stays much higher than for the desired outcome. Furthermore, using an educated guess allows reducing the computational domain of the problem by leaving out elements which will certainly be free of material.

These initial steps could be made by an algorithm that is much less computationally expensive than the TO, which will reduce the time spent on FEA. Despite TO results being

possible only to predict heuristically, there exist some tendencies in the dependence on the BC and forces applied to the element.

As one of the possible approaches to find suitable initial guess for TO, we propose using a predictive model based on the ANN. The idea behind it is to build a Machine Learning (ML) system able to infer an approximate result of a TO process based on the inputs for the TO problem. Here, the inputs of the model are initial conditions, constraints, and parameters of the problem and the outputs are approximate optimal material layouts for minimal compliance for elements with constrained volume under stress.

In this work, we design such an ANN system and train the model on a dataset which consists of the known solved cases of TO problems. We present the architecture of the model and analyze the influence of dataset parameters and the data representation on the quality of the inference results. Furthermore, we analyze the general capacity of the system to improve the performance of the TO process.

Unlike many of the ML applications, the training dataset is hard to obtain in our case. Every sample of the training dataset would be a pair of some initial conditions and the material layout, which is a result of an expensive TO process. One of the most important questions in the work is the trade-off between the size and properties of the training dataset and the quality of the inferred result with respect to their accuracy and suitability for the initial layout for the TO process.

1.2. Related Work

The goal of this section is to give an outlook of the previous research on the possibilities to improve TO performance. It also overviews the application of ML and data-driven approaches to simulations in engineering, especially the cases dealing with spatial 3D data and inferring the results of some simulations.

During the last two decades, multiple studies have been carried out on TO acceleration. One of the approaches is to apply GPU parallelization for solving a linear system of equations describing FEM, which was first demonstrated by Schmidt[28]. A recent work by Gavranovic[13] describes an implementation of an efficient parallel Multigrid method for TO acceleration using GPGPU.

A recent work by Ulu[35] suggests applying dimensionality-reduction techniques for 2D optimization domains, in particular finding an eigenvector representation using Principal Component Analysis and treating the loading vectors as inputs for a simple feed-forward neural network to obtain approximate results that can be used as an initial guess for the conventional TO process.

During the last decade, various CNN architectures dealing with spatial data were developed. A work by Sinha[34] suggests a deep residual network to generate various 3D shape surfaces learning features from RGB images. One of the most used architectures applied

to create generative models producing 3D or 2D data structures is the encoder-decoder architecture. The PixelCNN++ architecture as an image generating model is proposed in the work Salimans[27]. This work suggests down-sampling and up-sampling techniques for feature extraction and reconstruction, as well as applying various residual connections to facilitate the model's training process. A work by Wang[36] describes a 3D encoder-decoder generative adversarial network combined with a long-term recurrent convolutional network applied to produce 3D shapes of high resolution based on distorted data.

There were several works about applying Deep Learning approach to problems in engineering that treat discretized spacial fields. These cases give insight about model architectures and spatial data representations for simulations. The work by Guo[15] describes a CNN for prediction of approximate results for Computational Fluid Dynamics steady flow simulations. Authors suggest treating resulting velocity fields and representation of boundary conditions in an image-like fashion and propose an encoder-decoder CNN architecture for the model. An open-source implementation of this model by Hennigh[17] allows proving the ability to predict the simulation outcomes based on images of initial conditions.

Part II.

Theory

2. Topology Optimization

There are various TO techniques that allow finding optimal material layouts of the element that meets additional requirements on the physics of the element. Among others, restrictions on the weight and amount of material used for a certain mechanical element are common in the engineering.

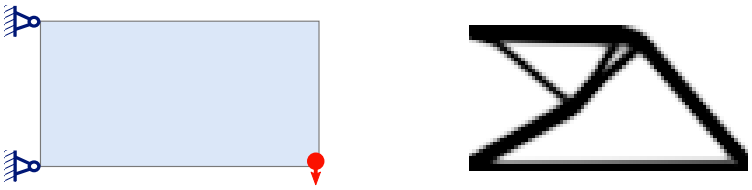


Figure 2.1.: The first picture shows TO problem formulation: the domain, BC, and loads. The second picture shows the resulting optimal material layout for the desired volume fraction of $V_f = 0.2$.

One of the approaches used for TO is Solid Isotropic Microstructure with Penalization (SIMP). This method uses the idea of "artificial density"[6], where for every hexahedral element of a discretized model a value of density variable ρ is assigned. These density values play the role of design variables in the optimization process and are used to present the goal function as a weighted sum of element-wise compliance. This approach has both well-established theoretical and empirical base and is easy to implement and interpret. In the end, the TO problem using SIMP approach is formulated as follows:

$$\begin{aligned} \underset{\rho}{\text{minimize}} \quad & c(\rho) = \mathbf{f}^{\mathsf{T}}\mathbf{u} = \mathbf{u}^{\mathsf{T}}\mathbf{K}_e(\rho)\mathbf{u} \\ \text{subject to} \quad & \frac{V(\rho)}{V_0} = \alpha \\ & \mathbf{K}_e(\rho)\mathbf{u} = \mathbf{f} \\ & 0 < \rho_{\min} \leq \rho \leq 1 \end{aligned}$$

In other words, our goal is to minimize the objective function $c(\rho)$ subjected to the constraint of the volume fraction $\alpha = \frac{V(\rho)}{V_0}$, where $V(\rho)$ is the volume occupied by the material and V_0 is the total volume of the design domain. The element stiffness matrix is denoted as $\mathbf{K}_e(\rho)$. The displacement and the force vector are denoted with respectively $\mathbf{u}$ and $\mathbf{f}$. Apart from the constraints on the volume fraction and the range of density values, the displacement vector, that is used to calculate the objective function, has to be physically

correct. That means that for every iteration of the optimization process, a Finite Element Analysis (FEA) problem should be solved to find $\mathbf{u}$. This step is the most computationally expensive in the whole TO process[28].

In order to update the density values on every iteration of the optimization process, the Optimality Criteria method is used[6, 33]. The update rule read as follows:

$$\rho_e^{new} = \begin{cases} \max(\rho_{min}, \rho_e - \delta_\rho), & \text{if } \rho_e B_e^\eta \leq \max(\rho_{min}, \rho_e - \delta_\rho) \\ \min(1, \rho_e + \delta_\rho), & \text{if } \min(1, \rho_e + \delta_\rho) \leq \rho_e B_e^\eta \\ \rho_e B_e^\eta, & \text{if } \max(\rho_{min}, \rho_e - \delta_\rho) < \rho_e B_e^\eta < \min(1, \rho_e + \delta_\rho) \end{cases} \tag{2.1}$$

where δ_ρ is a non-negative increment of the design variable and $\eta = \frac{1}{2}$ is a numerical dumping exponent coefficient. The sensitivity value B_e is updated by the optimality condition:

$$B_e^\eta = \frac{-\partial c / \partial \rho_e}{\lambda \partial V / \partial \rho_e} \tag{2.2}$$

Here we shall obtain the Lagrangian multiplier λ by a bisection algorithm. Finally, the sensitivity of the object is computed as

$$\frac{\partial c}{\partial \rho_e} = -p(\rho_e)^{p-1} \mathbf{u}_e^T \mathbf{K}_0 \mathbf{u}_e$$
$$\frac{\partial V}{\partial \rho_e} = 1$$

and used in the following update step.

A graphical overview of the algorithm is given in Figure 2.2. Such an algorithm is able to define the optimal material layout, based on the inputs describing BC, loads and the desired volume of the element, as shown in Figure 2.1.

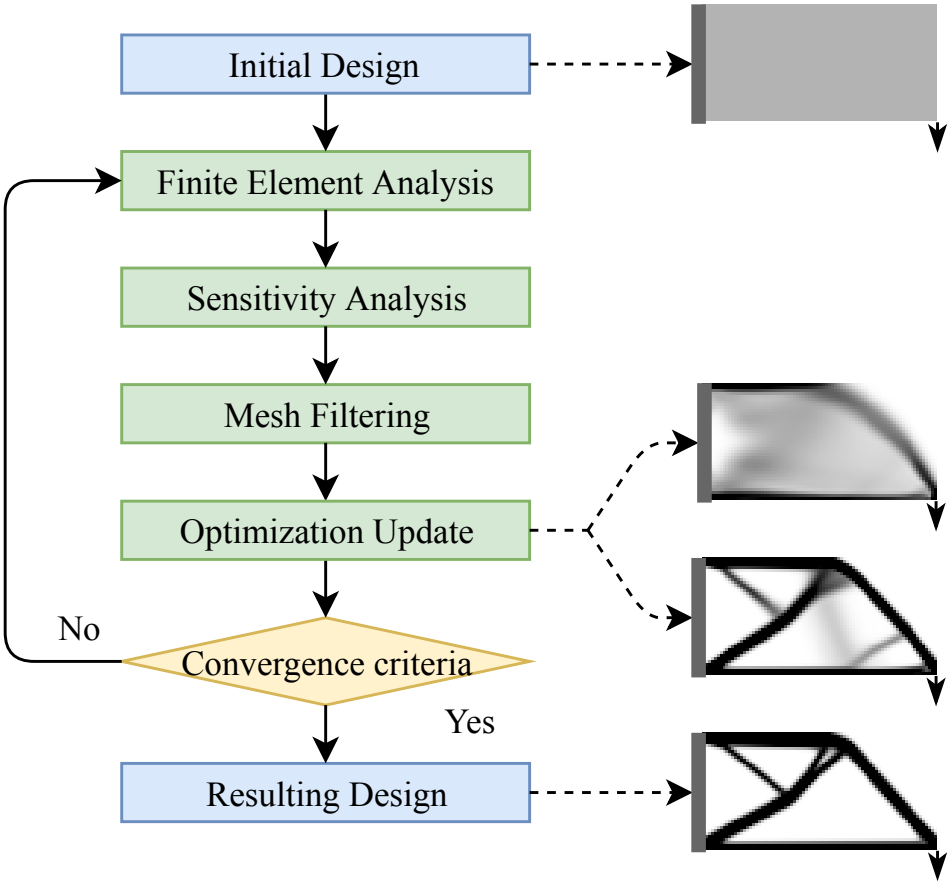


Figure 2.2.: The flowchart of the TO process.

3. Artificial Neural Networks

Artificial neural networks (ANN) are a type of algorithms in Machine Learning (ML), a branch of computer science and applied mathematics that utilizes statistical properties of data to analyze real-world phenomena and decision-making processes, that are hard to model explicitly. A goal of any ML algorithm is to use some known data related to a phenomenon in order to build a model producing insights about the phenomenon based on new some new input data, that was unknown previously. Every such a model has several structural and numerical parameters which values have to be optimized during the process of *training* that uses some known data produced by the studied problem.

The ML algorithms are usually categorized into several groups based on the data they use to train and how they treat them:

- *Supervised learning* uses a set of known input-outputs pairs to train a model able to treat new inputs.
- *Unsupervised learning* deals with data without labels and is used to find patterns within the data samples.
- *Reinforcement learning* learns how to make decisions based on the response from a certain environment.

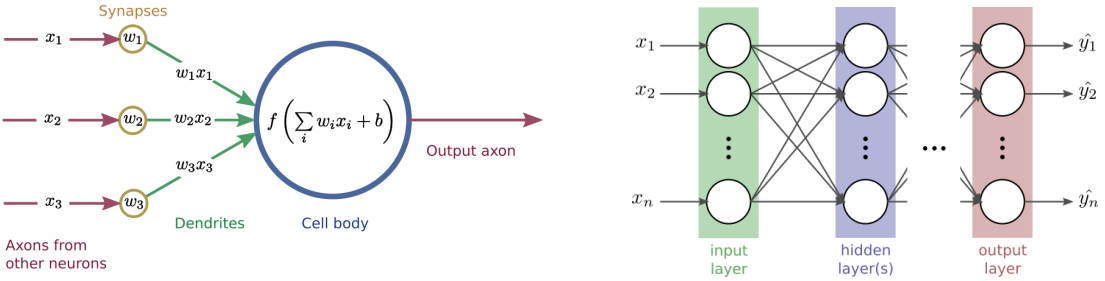
For the problem of this work, we consider algorithms that belong to supervised learning. This fits the goal of the works as we want to find a relation between some initial conditions and a resulting layout.

Most of ANN algorithms work in a supervised manner and are shown to be a powerful approach for various problems[14]. For the class of ANNs, a studied dependency is expressed using the notation of artificial *neurons*. A case of a simple dependency with a scalar output is similar to the behavior of a biological neuron of a brain that is able to generate electrical impulse based on signals received from several other neurons. Such an analogy has historically inspired the development of artificial neural network approach by Frank Rosenblatt[25]. The output of an artificial neuron y_j that takes input from n neurons $x_{i=1,..,n}$ with weights $w_{i=1,..,n}$ and some activation function f and is defined as:

$$y_j = f(\sum_{i=1}^{n} x_i w_i) \tag{3.1}$$

Usually, the values of the parameters w_i of every operation have to be found in the process of *training* in order to make the model describe the underlying dependence in an accurate way.

ANNs can also deal with more complex outputs i.e. matrices and tensors. Neurons that treat elements of the output separately could be combined into *layers*. Every layer could be considered as an independent transformation and can accept an output of the previous layer as an input for itself. Domains and codomains of these operations are mostly spaces of matrices or tensors over the field of real numbers. In the result, a complex dependency can be represented as a compositions of layers comprising separate neurons that form a network. A graphical representation of a single-neuron operation and the relations between neurons in a model are shown in Figure 3.1.



(a) A model of neuron describing data transformation.

(b) A model of an ANN with three layers shown.

Figure 3.1.: A structure of an ANN[37]. Layers which are not input or output ones are called *hidden*. Here, neurons of every next layer take inputs from every neuron of the previous layer; such a data transformation is called *fully connected layer*.

3.0.1. Optimization

The goal of the ANN training is to achieve a model that would predict the accurate results based on the various inputs. During the process of training, the optimal values of parameters that describe ANN transformations should be found. For that, we need to estimate how wrong are the predictions $\hat{Y} = \{\hat{y}(x),\ x \in X\}$ based on known ground-truth pairs of inputs and outputs $(x, y) \in X \times Y$. We express the difference between the predicted results and the ground-truth labels with the *loss function*, which is also called *cost* or *objective* function. This function indicates how generally wrong our model is. Because the outputs y_i are usually taken from a finite-dimensional vector space, one of the common ways to build a loss function is to sum the distances between all pairs of y_i and $\hat{y}_i$. For example,

we can take squared L_2 norms of the $y_i - \hat{y}_i$ errors and compute total Euclidean loss as

$$L(\hat{Y}|Y) = \frac{1}{2}\sum_{\hat{y}\in\hat{Y}}(y - \hat{y})^2 \tag{3.2}$$

The goal of the *training* process is to find such weights, or values of model parameters, that minimize the value of the loss function.

It is usually virtually impossible to analyze the dependency of loss function on parameters analytically. It is also not possible to check all the combination of weights values because the number of parameters is extremely large and the dependencies are non-trivial. Due to that, the training is implemented through a numerical optimization, usually based on the *gradient descent (GD)* algorithm. The idea of the method is based on finding the numerical gradient of the loss function with respect to parameters. With that, we iteratively update the weights until some convergence criterion is reached so that the loss function value is considered minimal for the set of found parameters values.

We can express the GD algorithm as an iterative process, with every iteration consisting of several steps.

1. Finding the derivative of the loss function at the current point of parametric space $\nabla L(W_t)$.
2. Updating the model weights in form of $W_{t+1} = W_t - \alpha\nabla L(W_t)$, where α is the *learning rate*.

On next iteration the gradient is found for the updated point, and so on. The gradient indicates the direction of the fastest growth of the loss function. Using its value we can define a new point in the parametric space. The process terminates when one of the user-defined criteria is met, for example, the change in the loss function is too small or the number of iteration exceeds the maximal amount.

Modifications of GD algorithm

The classical GD assumes that we could easily estimate the gradient of loss function at every point of the parametric space. However, in the case of ML, the value of gradient would depend on multiple data samples. Essentially, in order to find a single gradient value for a fixed set of parameters in full form, it would require calculating and averaging gradient found for every sample in the dataset, which is practically infeasible. To avoid that, the *stochastic gradient descent (SGD)* is used, for which we use only one or a few samples chosen randomly, called a *minibatch*, to calculate the gradient. Besides being much faster, this simplification shows good convergence and is very popular as an ANN optimizer[26].

The GD method has also multiple disadvantages. The algorithm gives a guarantee to converge only for the convex objective functions and generally converges slowly for ill-posed problems, with very slow evolution along the gradient. For the non-convex shapes

of the function, there is a chance that the algorithm will converge to a local minimum, without any possibility to escape and discover other regions of the domain[26]. Nevertheless, this approach is still being the most common way to optimize the parameters, especially if several modifications are introduced as a remedy for the disadvantages.

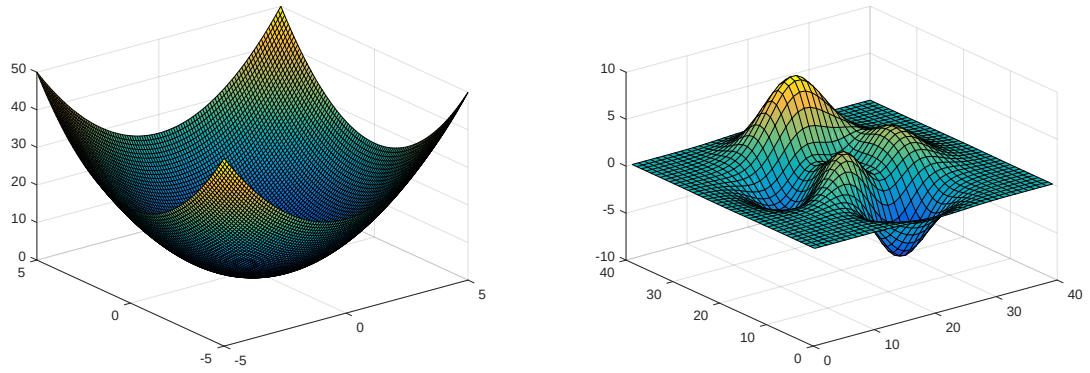


Figure 3.2.: Loss function forms. The left one is convex and has a single extremum. The right one is non-convex and is ill-posed for GD.

One of the simplest solutions to the slow convergence issue is introducing a notion of a numerical *momentum*. It is inspired by the mechanical dynamics and brings the optimization process closer to a simulation of a particle motion with the goal function playing a role of the potential field. The update rule has an additional velocity term V_t which adds certain inertia to the way parameters change, pushing the virtual particle more along the gradient. One of the GD-based methods that use the idea of momentum is *Nesterov Accelerated gradient (NAG)* and it shows better convergence properties compared to simple gradient descend. The algorithm is interpreted as follows:

1. Take an intermediate point in the direction of the momentum and calculate the gradient for it.
2. Use the gradient value to update both the parameter values and the momentum.

The update rule for NAG reads as follows:

$$\begin{aligned} V_t &= \mu V_t - \alpha \nabla L(W_t + \mu V_t) \\ W_{t+1} &= W_{t+1} + V_{t+1} \end{aligned} \tag{3.3}$$

Apart from the inability to find the region of the global minimum, another issue is fine-tuning of the optimal parameter value. With a large learning rate, the step of the optimization stays large, which leads to an oscillating behavior of the optimizer that constantly

oversteps the minimum. To improve the accuracy during the later iterations, the learning rate should be decreased with time, which is often done manually with some predefined scheme. However, a more general way would be to find learning rate adaptively, based on the trajectory made by the optimizer in the parameter space. The *Adaptive Gradient Algorithm (AdaGrad)* uses the sum of squared, thus non-negative, gradients over time to define the learning rate. In this way, the learning rate is large only while the historical gradient is small, and growing small while the gradient accumulates its value:

$$\begin{aligned} G_{t+1} &= G_t + \nabla L(W_t)^2 \\ W_{t+1} &= W_t - \mu \frac{\nabla L(W_t)}{\sqrt{G_{t+1}} + \epsilon} \end{aligned} \tag{3.4}$$

Furthermore, here every parameter has, in the end, its own learning rate, which allows keeping total precision and convergence speed at a good rate. Finally, the case of division by zero should be also prevented with an ϵ in denominator.

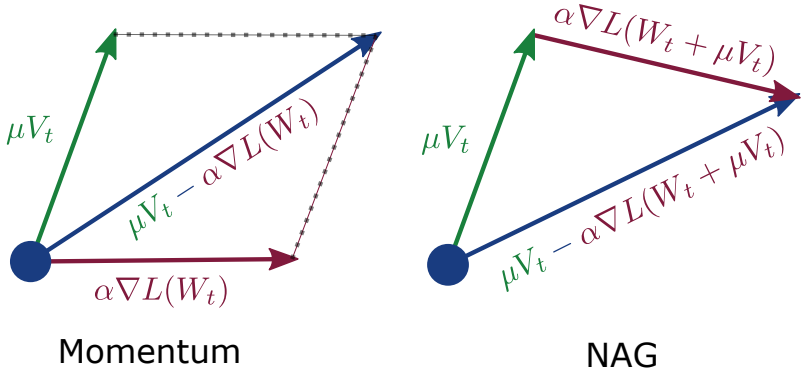


Figure 3.3.: The difference between Nesterov and regular optimization momentum[37].

One of the most popular optimization algorithms that treat well both finding the global minimum region and fine-tuning is *Adaptive Moment Estimation (Adam)* with the following update rule:

$$\begin{aligned} M_{t+1} &= \beta_1 M_t + (1 - \beta_1) \nabla L(W_t) \\ V_{t+1} &= \beta_2 V_t + (1 - \beta_2) \nabla L(W_t)^2 \\ W_{t+1} &= W_t - \mu \frac{\sqrt{1 - \beta_2^{t+1}}}{1 - \beta_1^{t+1}} \frac{M_{t+1}}{\sqrt{V_{t+1}} + \epsilon} \end{aligned} \tag{3.5}$$

Adam combines all aforementioned approaches and generalizes AdaGrad method. It is one of the most used and the most favorite optimization algorithm in ANNs nowadays[26]. It uses the second-order momentum to adapt learning rates, the first-order moment as an inertia term and bias correction multiplicative constant to prevent from update steps being close to zero.

3.0.2. Backward Propagation

The optimization process requires knowing values of the gradient of the loss function with respect to parameters space. In the general case, the form of the loss function is in unknown and defined on a high-dimensional parameter space. Furthermore, the loss function depends on training data samples and in principle infeasible to analyze. Thus, it is only possible to find values for a finite number of points that are of interest based on some numerical considerations.

One of the ideas that made ANNs fast and easy to implement is *automatic differentiation*. Since we describe the dependency between input and output of the network as a sequence of simple data transformations, like convolutions or activations, called layers, it is possible to determine the gradient of every single operation for every input and save it. Having the total transformation and applying the chain rule, we can easily express the total gradient using the atomic derivatives of every transformation, and the process of finding it is called automatic differentiation.

In its turn, the process of plugging in the values on every iteration of optimization in order to find loss function gradient value and update the parameters is called *backward propagation* or *backpropagation*. It is an essential principle of ANNs and many modern software frameworks offer means for automatic differentiation and backpropagation, allowing an easy model set-up and bringing good performance of its training process[14].

3.0.3. Activation Function

In the general case, not every dependency could be described and approximated as a composition of linear operations. A simple model that uses only linear operations could easily miss complex patterns. In order to build a more general model, non-linear *activation functions* applied to the output of the layers should be introduced.

One of the most famous activation function is the *sigmoid* function. This is a deterministic function $\sigma : \mathbb{R} \to [0;\, 1]$ and reads as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.6}$$

One of the issues of the sigmoid function is that it is not zero-centered. If the input values of the neuron always have the same sign it will lead to issues with the weight updates during the back-propagation process, since the values of the gradient will also always have the same sign.

The other popular non-linear activation function is the hyperbolic tangent which is a mapping $\tanh : \mathbb{R} \to [-1;\, 1]$ and equal to:

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{3.7}$$

It is very similar to the sigmoid, but due to the symmetricity of the range, it is zero-centered.

As it is seen from the graphs of functions in Figure 3.4, for the tails of both sigmoid and hyperbolic tangent, the value of the function is very close to either 0 or 1. This is called the *saturation problem* and it is expressed in the fact that for the most of the domain the gradient of the function is very close to zero. The gradient of the activation function influences the gradient of the cost function, which leads to a very slow convergence rate of the training process. This phenomenon, that is also called the *gradient vanishing*, is one of the reasons why sigm is rarely used as an activation function nowadays, while $\tanh$ still has some use.

The third family of activation functions are *Rectified Linear Units* (ReLU) which are based on thresholding of the input value at zero $f(x) = \max(0, x)$. It is the most widely used activation function in the last years and it shows a better convergence of SGD than previously mentioned ones and it is computationally very simple. The main disadvantage is that there is a chance that a large bias term will be calculated which will lead to so-called *"dying ReLU"*. That means that the gradient of the operation stays zero and the training process is stopped without any possibilities to recover.

The common solution to this problem is to replace putting to zero at the negative part of the domain with the function with a small negative slope so that gradient will never be zero. This type of activation function is called *parametric ReLU (PReLU)* and is defined as:

$$f_{PReLU}(x) = \begin{cases} \alpha x & : x < 0 \\ x & : x \geq 0 \end{cases} \tag{3.8}$$

One of the further modifications is an *Exponential Linear Unit (ELU)*. Like the ReLU activation function modifications, it avoids gradient vanishing, enhances normalization and shows a speedup for training, however, is slightly slower during propagation[8].

$$f_{PReLU}(x) = \begin{cases} x & : x < 0 \\ \alpha(e^x - 1) & : x \geq 0 \end{cases} \tag{3.9}$$

In our work, we apply another two advanced techniques used alongside activation functions. The first one is called *concatenated activation* which we used together with the ELU. It is inspired by the idea that the negative values propagated through network could be as important as positive ones, and, because they are treated by activation asymmetrically, they should be considered separately[32]. That means that at the activation step we should map every channel into two channels, the identical one and the negated one, and apply the activation function to them separately, thus considering a more general picture. In the end, the *Concatenated ReLU (CReLU)* activation reads as follows:

$$\rho_c : \mathbb{R} \to \mathbb{R}^2, \text{ defined } \forall x \in \mathbb{R} \text{ as } \rho_c(x) = ([x]_+, [-x]_+), \text{ where } [\cdot]_+ = \max(\cdot, 0) \tag{3.10}$$

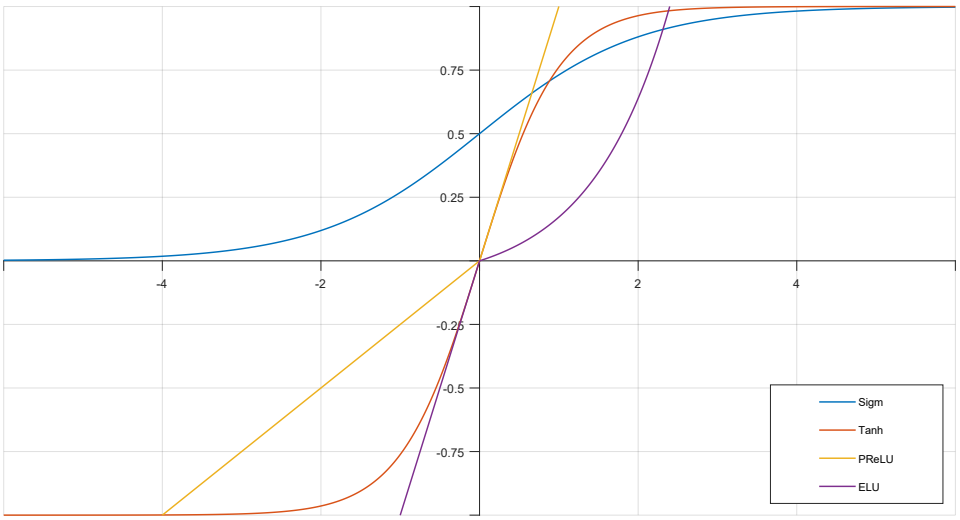


Figure 3.4.: Comparison of different activation functions.

The other activation technique is *gating* and it is based on doubling the number of channels at the convolution and then treating the groups of channels separately. We apply a sigmoid function only for a half of channels and then multiple the results element-wisely with the leftover channels. In this way, the convolution has two times more parameters comparing with a regular convolution, leading to the same size of the output tensors. Doubling the number of channels prevents the gradient vanishing since during the back-propagation the gradient will have both direct and sigmoid components[10].

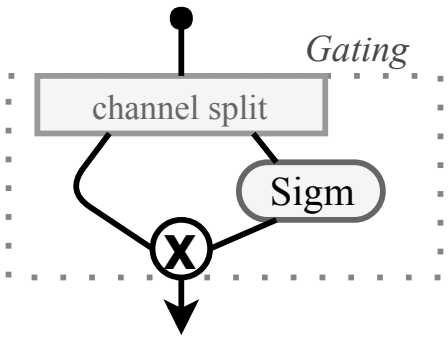


Figure 3.5.: A scheme of gating activation.

3.0.4. Regularization

One of the common issues for the ML models is *over-fitting*, or the low ability to generalize. This issue occurs when the model performs very well on the training set but shows bad results with the unknown input data while evaluation.

There are several ways to facilitate the ability of the model to generalize, and one of them is to put some restrictions on its parameters. One can decrease the number of parameters, which would lead to a simpler, and thus more general, model. The other is adding a penalty term, which depends on the parameters' values, to the loss function, as $L(w|x) = L_0(w|x) + R(w)$, which would tend to convergence to smaller values of the wights after the optimization. The large values of parameters typically mean that the model fits well the known data samples, which in its turn means that model learned the effects of the noise in the training dataset. Lower wights make the outputs change less with the small disturbance of the inputs, hence making the model more robust.

3.0.5. Common Regularization Techniques

Some of the most common types of the regularization are L_2 and L_1 regularizations. For the former one, the regularization term is L_2 norm of the model's parameters and could be expressed as $R = \frac{1}{2}\lambda w^2$, where λ is a regularization parameter. For the latter, it is respectively an L_1 norm of parameters $R = \lambda|w|$. In the result of the optimization, the L_2 one will lead to weights shrinking proportionally to their value, as $\frac{\partial \frac{1}{2}\lambda w^2}{\partial w} = \lambda w$, whereas L_1 shrinks them as constant, $\frac{\partial \lambda|w|}{\partial w} = \lambda \operatorname{sgn}(w)$. For the latter one, the weight tends to decrease faster and become sparse as many of them are put to zero value in the end, which allows explicit feature selection. However, if there are no special restrictions on the number of parameters, typically L_2 regularization is used as it giver better results than L_1[14].

The other method to deal with over-fitting is the *dropout* and it can be used alongside the L_1 and L_2 regularizations. It is based on omitting random neurons on each step of the training of the model. Typically, an approach of choosing any neuron with the probability of 50% is used, so at every step, we have a different combination of neurons. In the end, we change the structure of the network during the learning process and the resulting network is equal to an average of several models, which reduce the effect of over-fitting on every single network.

Finally, one of the most common approach to avoid over-fitting is *early stopping*. The optimization process happens in iterations and usually ends when the maximum number of iterations is reached. However, it is possible to choose a different convergence criterion, for example, reaching the optimum of the validation error. During the optimization, the model tries to fit the training data samples as close as possible and can end up showing bad results for new data. To avoid getting an over-fitted model, one can validate the model during the process of training, inferring results on data-samples that were not used for

training and analyzing the validation error. It could be performed once in several iterations of optimization and the start of validation error growth usually indicates over-fitting. A similar approach would be to obtain several models which underwent a different number of training iterations and choose one with the lowest validation error.

3.1. Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a type of ANN that is specifically designed to work with images and image-like spatial data. They primarily make use of *convolution* operations and are based on certain assumptions about images. The most important ones are:

- *Locality*, meaning that only elements in the vicinity could influence each other.
- *Transitional equivariance*, meaning that local features do not depend on the position of the image.

CNNs represent a dependency between models' input and output as a composition of multiple operations that typically include *convolutions*, *activations*, *poolings* and sometimes usual *fully connected layers*. The number of such operations and layers in most of the CNN models could reach several dozens, thus such many-layered models are called *deep* and CNNs are referred to *Deep Learning*[20]. This approach fits for our problem, since the final layout of TO is often composed of distinctive elements e.g. beams of different shapes, and depends on the relative position of the boundaries and the loaded points.

A CNN model expresses the dependency between inputs and outputs as a composition of operations like convolution, pooling, and transposed convolution, each having some parameters, like weights or elements of kernels.

3.1.1. Convolution Operation

The *convolution* operation could be defined as obtaining a value of every element of the output image as a weighted sum of the values of input image elements of a region in the vicinity of this output element, as shown in Figure 3.6. In general form, we say that a convolution is an operation on two n–dimensional arrays A and B which gives an array C of the same dimensionality as a result, which is defined as:

$$\mathbf{C} = \mathbf{A} * \mathbf{B} \text{ , where}$$
$$c_{j_1,\dots,j_n} = \sum_{i_1=1,\dots,i_n=1}^{K_1,\dots,K_n} a_{j_1+i_1-\lfloor\frac{K_1}{2}\rfloor,\dots,j_n+i_n-\lfloor\frac{K_n}{2}\rfloor} \cdot b_{i_1,\dots,i_n}$$
$$\text{and } \mathbf{A} = (a_{i_1,\dots,i_n}) \in \mathbb{R}^{N_1\times\dots\times N_n},\ \mathbf{B} = (b_{k_1,\dots,k_n}) \in \mathbb{R}^{K_1\times\dots\times K_n},$$
$$\mathbf{C} = (c_{j_1,\dots,j_n}) \in \mathbb{R}^{(N_1-K_1)\times\dots\times(N_n-K_n)} \tag{3.11}$$

In the case of CNNs, the first operand A is the input of the layer and the second operand B is the array of weights. The second array is called the *filter* or the *kernel* and its elements are usually the parameters that should be found during the training. When dealing with images, every input and output is a 3D tensor of data, with height and width corresponding to the dimensions of the 2D image. The depth of the tensor is a number of *channels*, which could be different representations of the image, like color channels for the RGB format.

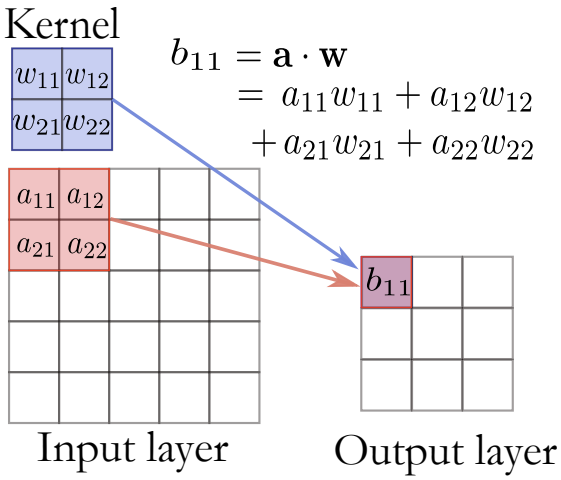


Figure 3.6.: A visualization of a convolution operation.

Such a definition of the convolution assures the property of *locality*. That means that the output neuron is influenced only by a vicinity of neurons in the input. The size of the input window is called the *local receptive field* of the neuron. In CNNs, every element of the next layer is obtained by sliding the local receptive field along the previous layer and applying a convolution operation as shown in Figure 3.7.

The other important property of CNNs implied by the definition of convolution is *parameter sharing*. It means that the kernel which is applied to every local receptive field is the same for the whole layer. The property of such operation uniformity with respect to

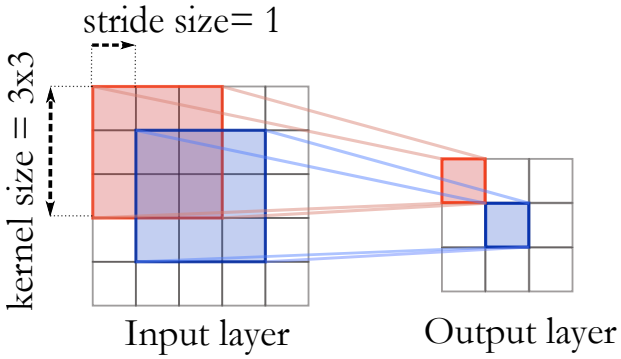


Figure 3.7.: Receptive fields of the neurons of the output.

shifts is also referred to as *translation equivariance*. This property corresponds to the nature of most of images, which could be described as consisting of different features combined in a larger entity. In the process of optimization, the optimal kernels are found, and they describe the features that recur within the training dataset. An intermediate output of a convolutional network shows the representation of the previous image in terms of these features. Furthermore, the size of the kernel, which is applied to every block of pixels with some step, is usually much smaller than the size of the image. This means that parameter sharing drastically reduces the number of model's parameters compared to the fully connected networks and convolutions are beneficial to the speed of the model's training.

3.1.2. Additional CNN Operations

In the general case, the output of the layer depends on several channels of the input. By summing values across the layers and applying a different number of kernels, we can have an arbitrary number of channels at every layer. It is possible to reduce the number of channels at the next layer by applying 1×1 *convolutions*. They are, essentially, a weighted sum of input elements along channels only and without using spatial kernels. Thus, they change the number of channels but keep their size.

One of the properties of convolution operation is that the size of the output is smaller than the size of the input, because during the sliding the filter will fit fewer times than the number of elements along the side. If it is important to have the inputs and outputs of the layer of the same size, it is possible to apply *padding*. That means that for the output elements close to the edge we span the receptive field outside of the size of the input, using virtual input elements with some value. One of the most common way to pad the receptive field is *zero-padding* which is shown in Figure 3.8.

The distance between two receptive fields, or a size of the sliding step, is called *stride* and could vary for different layers. A padded convolution with a stride larger than 1 reduces the size of the output image and the stride defines the ratio of the input and output

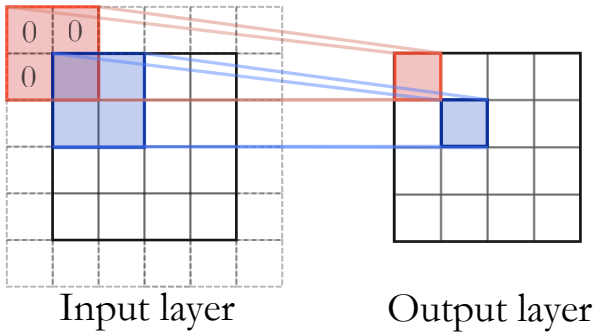


Figure 3.8.: Convolutions with padding preserve the size of the output.

linear sizes. Such an operation could be considered as a *dimensionality reduction* and every element of its output encapsulates more unique influence of the input.

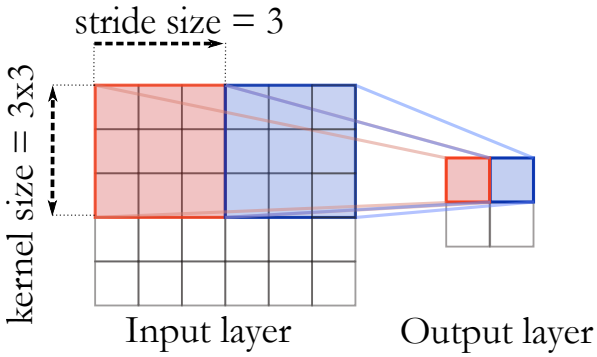


Figure 3.9.: Stride size defines the distance between two nearest receptive fields of convolution.

Another operation that is used in CNNs is *pooling*. Essentially, it is a reduction operation that uses values of elements in a receptive field to produce a single value. This reduces the size of the image at the next layer and builds a representation in a lower dimensional space. Typically, average or max operations are used and the receptive fields at which operation is applied do not overlap. This operation could be generalized with a convolution operation with a stride equal to the size of the kernel. For example, the average pooling will be identical to the kernel values all equal to the inverse of the number of its elements. The difference is that kernels are predefined in the case of pooling. This reduces the number of parameters to be trained and positively influences the speed of model's training.

Furthermore, there is an operation that allows increasing the size of the next layer, called *deconvolution* or *transposed convolution*. Essentially, it is an interpolation of a previous lower-dimensional layer onto a higher resolution. For a transposed convolution, every element of the input is element-wisely multiplied by a kernel and projected to the output.

Thus, every element of the output is equal to the sum of the elements of input multiplied by a kernel of size n in the form:

$$b_{ij} = \sum_{k=i-n+1,l=j-n+1}^{i,j} a_{kl} \cdot w_{i-k+1,j-l+1} \tag{3.12}$$

which is visually represented in Figure 3.10. For this formula, the dependency of the input

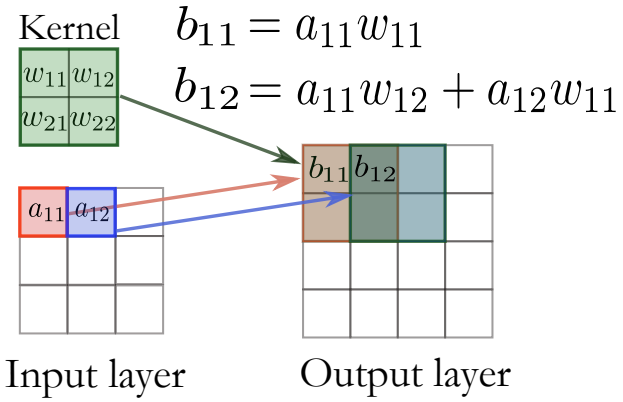


Figure 3.10.: Visualization of a transposed convolution operation.

on the output with given weights will have a form of a standard convolution[11]. By combining a transposed convolution with a stride of a variable length, it is possible to interpolate a previous layer on a layer of an arbitrary larger size. In our case, we apply transposed convolution with a stride of size 2. In such a way, we reconstruct the result based on the low-dimensional representation of inputs using features found in the space of results.

3.1.3. CNN Architectures

Using the aforementioned operations, it is possible to compose a complex function that would model a dependency described by the ANN. The structure of such a function and the way the layers are designed and combined is called the *architecture* of the network. There are many classes of architectures of CNNs that are being researched and applied these days. The main two architecture types that influenced this work are Residual networks and Encoder-decoder networks.

A *Residual Network* is a type of the network, where an atomic part of the transformation is defined in the form $F(x) = H(x) + x$, which means that the gradient would describe the change of the residual between the original and transformed data[16]. This allows increasing accuracy while having smaller amount of layers and to speed up the training process.

The *encoder-decoder* architecture consists of two groups of data transformations: encoding and decoding[19]. The former one, encoding, goals to find a representation of the input in form of prominent features. It consists of the sequential application of size-preserving convolutions and down-sampling convolutions with increasing amount of features. After the application of several blocks of such operations, inputs transform from a large image with few channels into a many-channel image of a very small size, which we call the low-dimensional representation. The latter part of the network, decoding, consists of several blocks of operations of normal convolutions alternating with *transposed convolutions*. Every transposed convolution up-samples the data propagating through the network, increasing the size of the channels and decreasing the number of channels, which corresponds to the reconstruction of the image using features in the space of outputs. This architecture is widely used for image segmentation task, in particular in the field of the medical computer vision, and there are many of the architecture variations e.g. U-net and V-net[24, 19].

Part III.

Methodology

4. Data Preparation

This chapter describes the data that was used to train and evaluate the ANN system. We discuss the parameters used to define the TO problems, their variability, and properties. The other aspect to be covered is the ways to represent these data in a way suitable for the ANN system input. Lastly, the chapter discusses the aspects of choosing the proper training dataset for the ANN system.

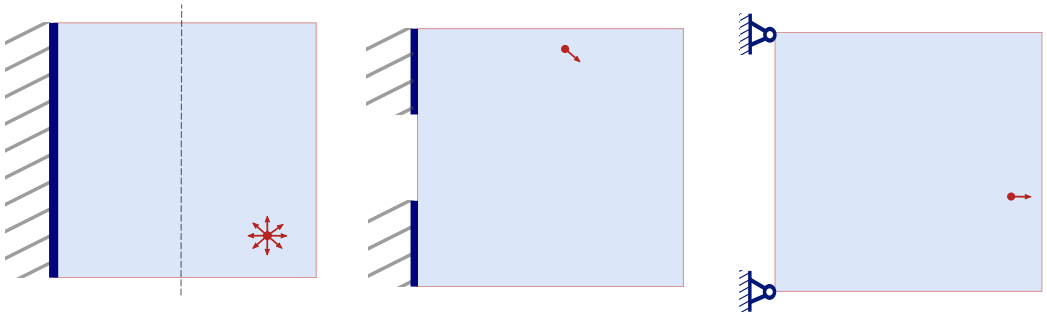
4.1. Topology Optimization Problem Cases

The physical problem, for which the TO is performed, is defined by the boundary conditions (BC) and the force applied to the element. The former has a view of Dirichlet BC $\mathbf{u}\big|_{\partial\Gamma} = 0$ and defines the parts of the element that experience no displacement. In the general form, the latter is a field of forces $\mathbf{F}(x, y) = (f_x, f_y)$ applied to points of the element. In order to perform the Finite Element Analysis (FEA), which is used during the TO, one needs to translate this conditions into a discretized form. In such a form, the BC defines the fixed degrees of freedom (DoFs) and the latter defines the non-zero elements of the right-hand-side vector of loads. To obtain a set of TO results that could be used as the training and testing samples for the ANN, we performed TO for various combination of values of the boundary conditions and load cases.

During the first phase of the work, only 2D cases of TO were considered. Creation of the model for 2D domains and the analysis of its performance was an important step in proofing of the concept of such an ANN approach to TO field. The decision to start the work by performing 2D analysis was also dictated by data availability and speed of dataset preparation, as 2D TO solutions take significantly less time than 3D ones. For a 2D case, we could generate large training datasets with more variable parameters and experiment with different representations. The dimensionality of the data also influenced the number of model parameters, which means that training a single model took much less time and it was possible to introduce changes or fine-tune the model quickly.

4.1.1. 2D Cases of TO

During this work, among all possible BCs, we considered only fixing one of the edges, or walls, of the domain. For every chosen side, we decided to consider one of the three types of BC, as shown in Figure 4.1:



(a) Solid wall (b) A gap in the wall (c) Corner points fixed

Figure 4.1.: Different BC considered for the 2D TO cases denoted with blue. Red denotes a force applied to one of the points.

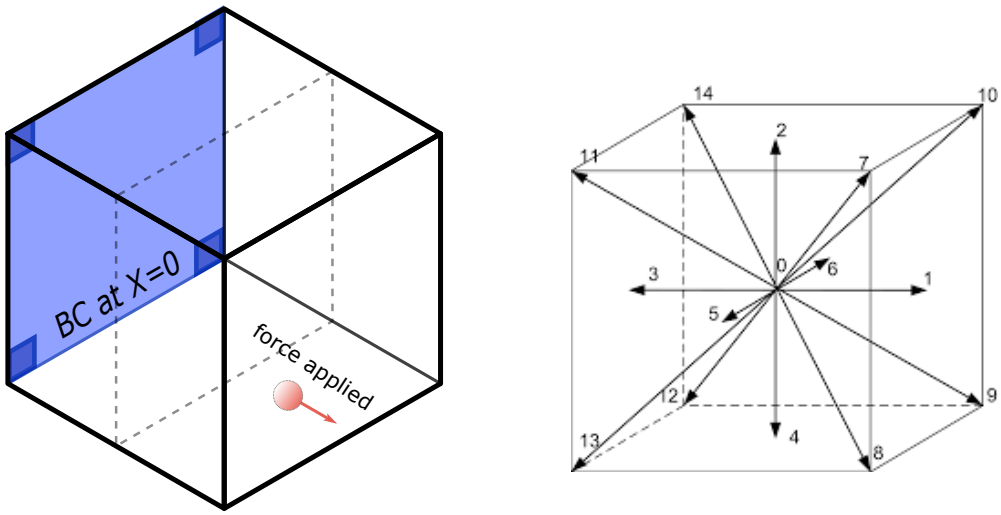
- fixing the entire wall
- leaving the inner $\frac{1}{3}$ of the wall unfixed
- fixing only the extreme points of the wall

As a simple load case, we considered only situations where a force is applied to a single point of the element. The variable parameters here are:

- the coordinates of the force application (x, y)
- the direction of the force (f_x, f_y)

The magnitude of the force in every direction is kept constant and unit, and the direction of the force is chosen among one of the eight possible ones, as shown in Figure 4.1: along one of the axes in one of the directions, or a combination of those. We considered cases when the force is applied to a point in the domain's half which is the furthermost from the fixed wall. The last parameter that serves as an input for TO, is the value of the desired volume fraction V_f occupied by the material, for which we choose among values: $0.1, 0.2, 0.4$. For the most cases, we considered the domain of size 32×32 square elements. In the end, we obtain the number of input samples available for a training dataset equal to $4\,(\text{fixed side}) \times 3\,(\text{type of BC}) \times 3\,(\text{value of volume fraction}) \times 8\,(\text{force direction}) \times 32\,(\text{Y coordinate}) \times 16\,(\text{X coordinate}) = 147,446$ layouts.

Because of the symmetry of the square problem domain, for every case where a single wall is fixed as a BC, there are three cases equivalent up to a rotation on $90°, 180°, 270°$ angles. That means that it is sufficient to generate layouts only for a single domain edge fixed and then apply *data augmentation* to obtain all the other results. Data augmentation is a simple transformation applied to every sample of the training dataset, a rotation in our



(a) TO problem domain and BC (b) 14 possible force directions[21]

Figure 4.2.: TO problem for the 3D case.

case, that allows having a richer set with easier means. In our case, we reduced the time of dataset generation by almost 4 times since we had four possible edges to fix.

4.1.2. 3D Cases of TO

For the second series of experiments, we required 3D models of optimized elements. Because of the larger design space and larger problem size, which leads to much larger time spent to obtain a single result, we decided to restrict data variability. For the 3D case, we left only a single volume fraction $V_f = 0.2$, as well as single BC case, where four corner elements of the $X = 0$ face of the domain are fixed. The problem domain we considered has the size of $32 \times 32 \times 32$ hexahedral elements, and we performed TO for cases when a force was applied to a single element. We generated TO results not for every possible point of load application, but for points with the step of 4 in every dimension. As in the 2D case, we considered only the furthermost half of the domain in the X direction. For every point of load application, we considered forces of unit magnitude directed to each face of the hexahedral element, as well as towards each vertex, which corresponds to $D3Q15$ model of hexahedral directions[21]. In total, for the entire dataset, we generated $14\,(\text{force direction}) \times 5\,(\text{Y coordinate}) \times 9\,(\text{X coordinate}) \times 9\,(\text{Y coordinate}) = 5670$ models.

4.2. Dataset generation

For the purpose of generating the dataset of known solved TO problems for the 2D case, we used the MATLAB code provided in the paper "A 99 line topology optimization code written in Matlab"[33]. Its goal is to minimize the compliance for 2D material layout under some load. The code uses square elements for finite element discretization and uses the power-law model to represent the compliance as the goal function, as described in Chapter 2.

The code was modified in order to have several arbitrary points where the load is applied so that a more general case is considered. A solution for a single TO problem is obtained by calling a MATLAB function, where all TO inputs are passed as parameters. The parameters include the problem dimension, volume fraction, power penalty, convergence criterion, coordinates of force application points and the force vector. The script saves the resulting material layout as a gray-scale PNG image where every pixel value denotes the virtual material density of the corresponding square finite element. For the typical problem size of 32×32, a single TO problem solution took less than $1\mathrm{s}$ using a single CPU. Totally there were ~ 150 thousand layouts created.

To generate results of compliance minimization for 3D models we used a program called IDeAs, that was created at Siemens CT[13]. It is a program for fast GPU-based parallel solutions of TO problems in solid mechanics. It offers high-performing Multigrid[5] CUDA-based solver for linear elasticity equations used for the TO process. The parameters of the TO problem are defined in a JSON[4] file, in which one should state STEP[3] files describing the geometrical domain, fixed regions for BC, and surfaces at which the force is applied. The BC type, force vector, and volume fraction value are also stated, as well as parameters of physics, discretization, and optimization analysis. Using 32 threads on a single Nvidia GeForce GTX 1080 Ti GPU[9], we generated ~ 5 thousands of $32 \times 32 \times 32$ voxels resulting models, each taking about 10 seconds.

4.3. Inputs of the ANN Model

In our case, for every fixed domain, a TO problem is defined by BC, the field of applied forces $\mathbf{F}$ and the volume fraction V_f occupied by the material. These are the three parameters that vary from case to case of TO problem when others stay the same, therefore the model should be able to infer a proper layout using only this varying information. One of the problems during the design of the model was to define how does the model should account for this information.

The BC is defined by a geometrical locus on the domain for which values of some function is predefined. In our case, it is the displacement vector $\mathbf{u}$ which is equal to zero for these points. After the discretization, the BC is expressed in having some DoFs of the domain fixed. Our domain has a correspondence with an image where every element of a

TO problem maps to an element of an array. With that, we can represent the BC with an array, which elements with indices corresponding to coordinates of the fixed DoFs have a value of 1, and everything else is 0.

In a similar manner, we represent the force field applied to the domain element. We have two different images for X and Y components of the force. For every force application point, we fill the array element with the $+1$ or -1 value depending on the force direction, leaving everything else 0.

The volume fraction V_f is essentially a scalar value and it is accounted in the system by introducing a new channel, tiled with a single value, at the ANN's layer of the lowest dimensionality representation as described in Section 5.1.

The images of BC, force in X direction and force in Y direction form the three input channels of the model for which the convolutions are applied. The issue with such a representation of BC and force fields is that there are only a few non-zero elements in the channel, that leads very little information influx in the model and makes it very hard for the model to train. For a better model performance, a great change of the outputs should be produced by a great change of the inputs[14]. As a remedy for this problem, we populate the input channels with more non-zero elements representing meaningful information by finding a *signed distance field (SDF)* of every input channel[7].

For every channel, we consider only non-zero elements as important and put a virtual surface around them in order to find the signed distance to them for every pixel of the image. The ambiguity for the force direction is solved by defining the inner part of the zero-surface as either inner or outer while computing the SDF, depending on the direction of the force.

4.3.1. Signed Distance Field

One of the ways to describe a set of geometrical points Ω within a domain is defining the distance to the boundary $\partial\Omega$ of Ω for every point of the domain. The values of such a function are typically negative if the point is located within Ω and positive it is outside. In the end, we define the SDF as a metric space X with metrics f, also called *signed distance function* and defined as:

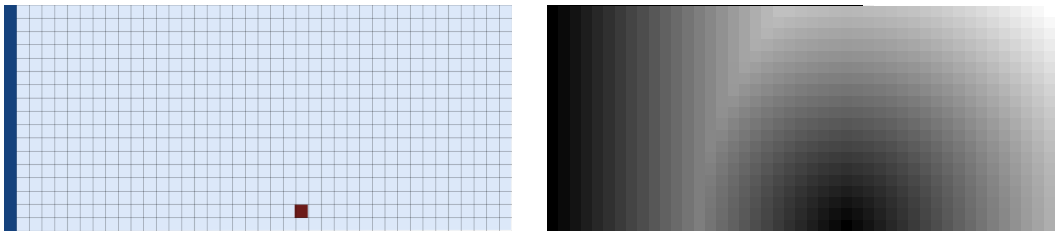
$$f(x) = \begin{cases} d(x, \partial\Omega) & \text{if } x \in \Omega \\ -d(x, \partial\Omega) & \text{if } x \in \Omega^{\complement} \end{cases}, \quad \text{where } d(x, \partial\Omega) := \inf_{y \in \partial\Omega} d(x, y) \tag{4.1}$$

If Ω is a subset of Euclidean space and has smooth boundaries, then the gradient of the metrics is constant at every point and satisfies the *Eikonal equation*:

$$|\nabla f(x)| = \mathbf{1}, \quad \forall x \in X \tag{4.2}$$

This equation describes the propagation of the wavefront started from the boundary of the Ω and its solution describes the travel time of the wavefront to every point of the domain in case of the constant speed of propagation[31]. Such travel time function is equal to the signed distance function.

There is a fast way to solve this equation for the points located on the regular grid called *Fast Marching Method (FMM)*. It is a graph traversing algorithm based on the Dijkstra's algorithm[31] and its complexity is $O(M \log M)$ where M is the number of grid points at the domain. There are plenty of FMM implementations available, and we chose to use python *ski-fmm* package[29], a part of SciKit framework that allows obtaining SFD as Numpy array[1] by calling only one function and passing an array describing a zero-boundary. In the end, we have a fast and easy method to describe our input images having all their pixels populated with meaningful values.



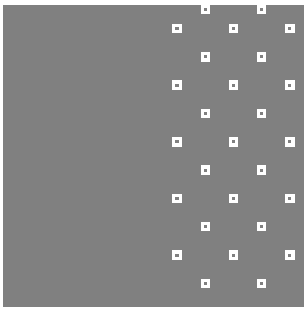
(a) Default input channels. (b) Input channels after SDF.

Figure 4.3.: An input of the ANN model with and without pre-processing. The latter has more non-zero pixels that change for different TO cases.

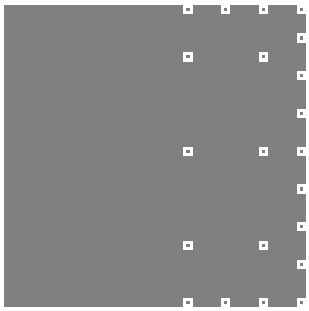
4.3.2. Training Dataset

One of the main goals of the work was to estimate the capability of such an approach to produce the model with high accuracy and trained using as small as possible training dataset. For the 3D case, preparing every data sample was taking more than 10 seconds for a relatively coarse resolution. This means that the time spent on preparing a large dataset, combined with training of the network, could diminish the speedup gained by applying the ANN system and educated guesses.

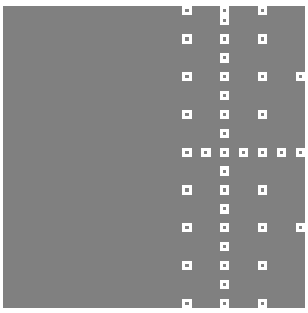
To define the influence of the choice of the training dataset S_{tr} on the resulting model and its performance, we prepared several of those. Every training dataset S_{tr} had a different number of samples and diffident distribution of parameter values within itself. Each of the training datasets consisted of samples $s = (x_{BC}, x_{F_x}, x_{F_y}, y) \in S_{tr}$ that correspond to a particular TO problem and its solution. Each sample includes images describing problem's boundary condition x_{BC}, force field in X and Y directions x_{F_x} and x_{F_y}, and the resulting material layout y. One of the main questions was how to chose the samples with respect to the load application points, so we created datasets each having different patterns for the



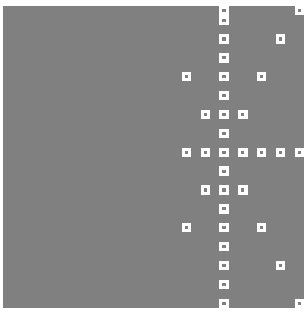
(a) Checkered pattern



(b) More loads at the edges



(c) Sparse grid pattern



(d) Loads along cross lines

Figure 4.4.: Comparison of different sampling patterns for the training dataset. Every pixel denotes a sample, coordinates of pixel groups denote a coordinate of force application point, every pixel around denotes a direction of the force. Figures show the images used to diagnose the real datasets.

coordinate of force application point. For every pattern, we trained models to see which one gives a better model performance. Among the patterns, we have chosen the next ones: checkered pattern, a sparse grid, and the force applied at the edges.

5. Method and Model

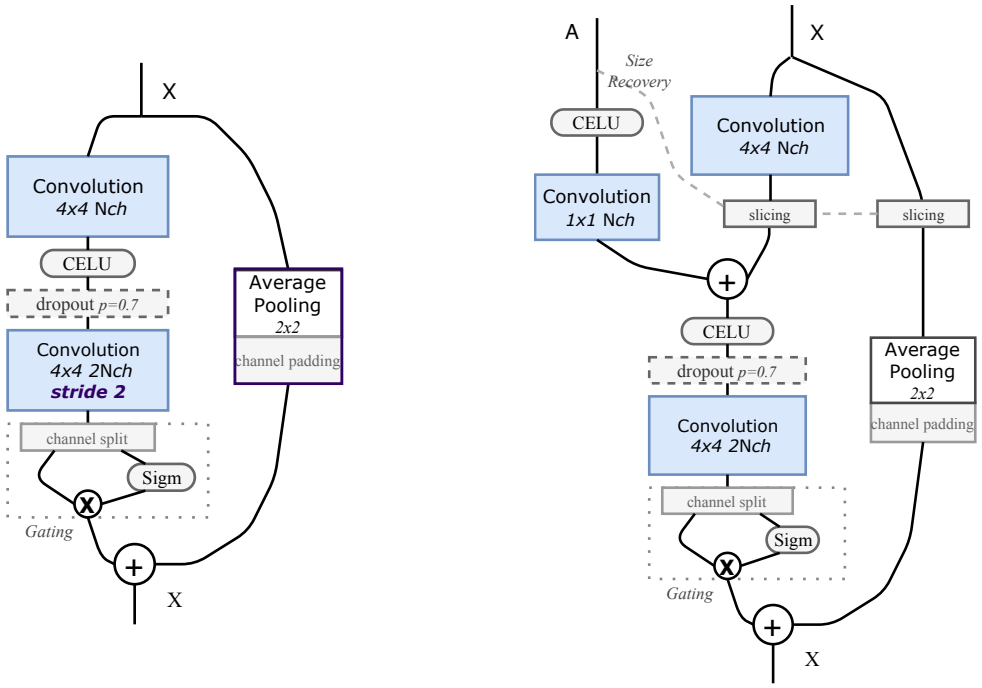
This chapter describes the implementation aspects of the work. The chapter gives the description of the ANN architecture and its reasoning, the general pipeline of computations and data manipulations applied to set up the system creating data that improves the TO performance. The ways for the system analysis and the software used for its creation are described as well.

5.1. ANN Architecture

The model used to describe the dependency between inputs and the resulting material layout is a CNN which structure relates to the class of *encoder-decoder* architectures. This network describes a generative model that fits for the regression tasks when input and outputs are of the same size. The overall architecture of this model is inspired by the PixelCNN++[27] architecture and its application for engineering problems[15, 17].

In our case, an image of four channels transforms into 128 different representations during the encoding. The *encoder* consists of nine blocks of convolutions, each describing its output in a *residual* form. In every block, we apply two convolutions for every channel. After the first one we apply the activation function and *dropout* on the phase of training for the sake of regularization, but not in the final model. The activation function applied after the first convolution is *Concatenated Exponential LU (CELU)*. After the second convolution, we apply the *gating* activation with sigmoid. Every second block has a stride for its second convolution equal to two, which decrease the size of the output in two times in every direction. Here we also apply average pooling and pad empty channels at the residual channel to fit the new size of the image. In total, the residual block of the encoder is represented visually in Figure 5.1a.

The decoder part has the same amount of blocks, each increasing the size of the channel. Every block of the decoder is similar to the one used for the encoder, apart from using *shortcut connections* from the previous layers. That means that every block has an additional input, namely the result of the encoder layer that has the same output size as the current decoder layer. After bringing layers to similar dimensionality, they are summed. This enhances the preservation of general geometrical properties of the input and facilitates the training process[18]. To reduce the number of channels at the shortcuts we apply 1×1 *convolutions*. Also, we need to restore the size of the image after the up-sampling, because of the striding technique leading only to sizes multiple of sizes of lower layers.



(a) Residual block of the encoder.

(b) Residual block of the decoder.

Figure 5.1.: A flowchart for residual blocks used in the network architecture. Dashed denotes dropout applied only during the training phase. Purple denotes down-sampling operations applied every second block only. Blue denotes convolution layers with weights to be trained. Light dashed means meta-information propagation.

The size of the shortcut image is passed as meta-information to restore the size of every shortcut. The encoder residual block could be easier understood using Figure 5.1b.

One of the tweaks we required to build a model for our case allowed the network having an additional scalar input. Since we want to enforce the output having the desired non-empty volume fraction V_f, we used its value as an input as well. We tiled a matrix of the size of the lowest-dimensional representation with values of the volume fraction and added one more channel at the bottleneck of the network, as shown in Figure 5.2.

One of the issues of the transposed convolution is a checkered error pattern occurring in the output. This is caused by the fact that for kernel size being non-multiple of stride, different output elements have a different number of input elements influencing them. In the result, this gives different magnitudes for some output coordinates of certain multiplicity. To mitigate this effect we applied kernels of size 4×4 since the stride we used during up-sampling is 2.

After the sequence of four transposed convolutions, each followed by a residual block with a shortcut, we apply a single convolution layer with one kernel to decrease the final channels' dimensionality. In the end, based on the 128 channels of low-dimensional representation, we obtain a final image of a single channel of size equal to the TO problem domain. The total architecture of our model is visually represented in Figure 5.2.

The architecture for the 3D case is very similar to the 2D one. For the 3D case, every convolution and transposed convolution is replaced with a corresponding *3D convolution*, where channels and kernels are cubes and summation happens across three dimensions. The number of blocks, their structure, and all numeral parameters are the same for the 3D case architecture. The number of channels is different for the 3D case since there is a channel describing forces along the Z axis.

Since this architecture belongs to encoder-decoder architectures, this work also considers the comparison of the performance with other similar architectures. For this purpose we consider two other encoder-decoder architectures, U-net[24] and V-net[19], that are used in medical image segmentation. This architectures use a different structure of blocks, with a different number of convolutions, sizes of kernels, activation functions and do not consider residual shortcuts. They also suggest a different overall structure with a different number of convolution blocks.

5.2. Loss function

The optimization, performed during the training, goals the minimization of the loss function. Thus, one of the most important steps during the design is defining a proper loss function which will precisely encapsulate our requirements on the results inferred by the model.

Our idea of the result produced by the model is an image describing material density layout being as close as possible to the result of conventional TO process. In this way, we have to minimize the difference between the known optimized material design for given input and the result inferred by the model. However, the difference between the two arrays of values is not easy to describe with a single scalar. Usually, the influence of a single element of output is low, which would make the optimization process extremely slow.

In order to tackle all these issues, we formulate our problem as a classification problem for every single element of the output channel for given input. We treat our ground-truth image of the layout as an array for which every element can belong to one of two classes, namely occupied by material (1) or not (0), for which we should threshold our results of known TO problems. In such a formulation, the goal of our system is to classify every element of output array based on the inputs and give the confidence for the inferred classes. Thus, for a single input, our model performs binary classification for multiple out-

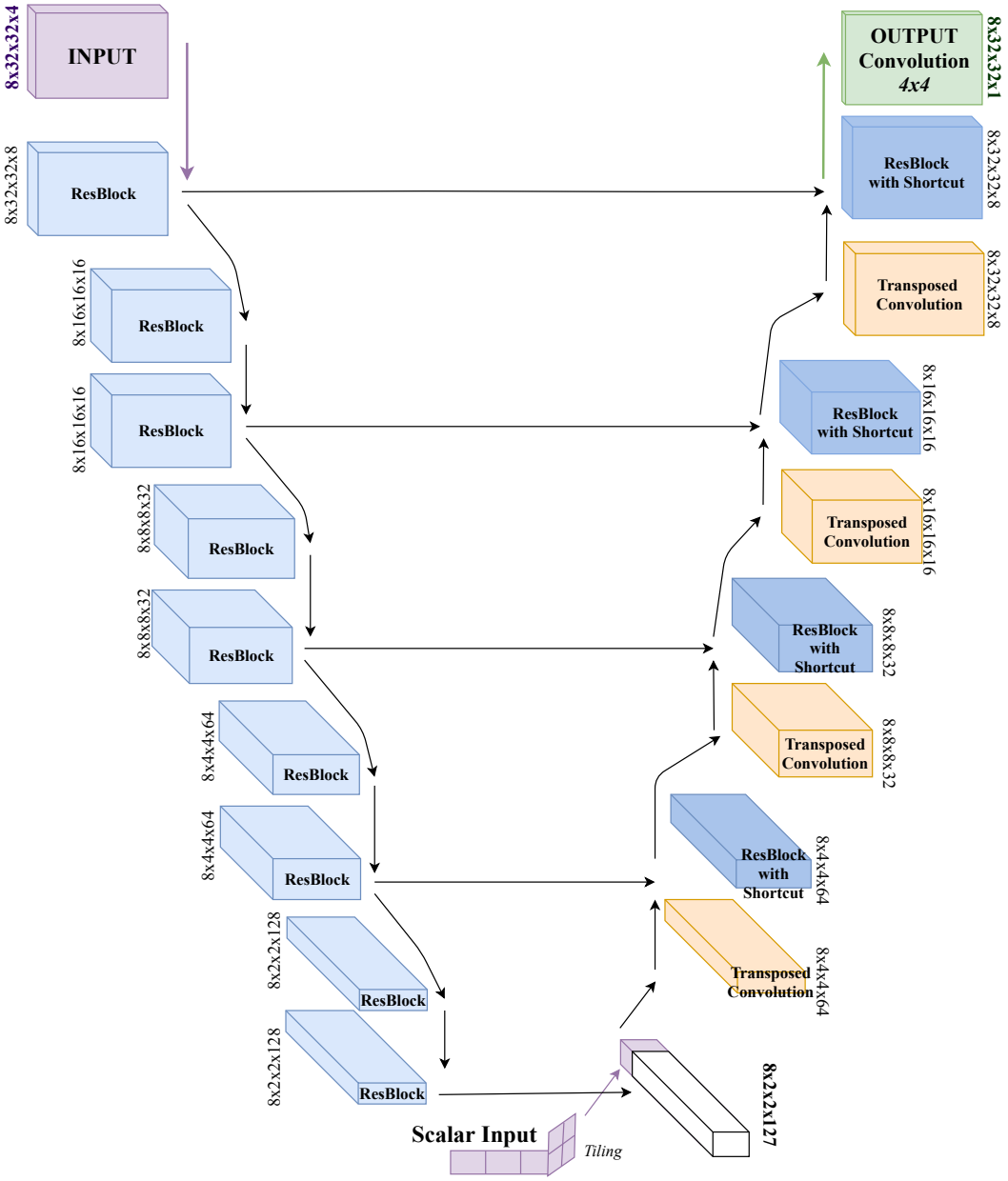


Figure 5.2.: The data flowchart of the model, describing the CNN's architecture. Here the sizes are shown for the 32×32 domain resolution, most commonly used in the work.

puts. This means that by discretizing the problem we reduce the information flux over the model and significantly increase the speed of the model training.

In this way, we define our loss function as a sum of logits of binary cross-entropy between the output of the model and reference image calculated for every output pixel. The loss function, as a dependency on model's parameters w for the given set of reference outputs Y and the set of model's outputs $\hat{Y}$, has the next form:

$$L_{(0)} = L_{CE}(w|Y,\hat{Y}) = -\frac{1}{|Y|}\sum_{y\in Y}[y\log\hat{y^*} + (1-y)\log(1-\hat{y^*})], \text{ where } \hat{y^*} = \frac{1}{1+e^{-\hat{y}(w)}} \quad (5.1)$$

Furthermore, in order to prevent over-fitting, we incorporate a regularization term in the form of an L_2 norm of all trained parameters, multiplied by a small constant α_{reg}:

$$L_{(1)}(w) = L_{CE}(w) + \alpha_{reg}R_{L_2}(w), \text{ where } R_{L_2}(w) = (\sum_i |w_i|^2)^{\frac{1}{2}} \quad (5.2)$$

One of the clear requirements defined by the TO problem itself is the resulting layout having the desired volume. Since the closeness to the reference data does not implicitly put any restriction on this criterion, we decided to use several ways to facilitate the inferred layout having the desired volume. One of the ways used in the system is adding a volume fraction penalty term to the loss function. For this, we calculate the volume of the inferred result as a count of the pixels for which the probability to be occupied is higher than a threshold. We took its ratio with the volume stated for the reference problem and then calculated the absolute difference with the desired volume fraction. The sum of these absolute differences is added to the loss function with some small multiplicative constant β:

$$L_{(2)}(w) = L_{CE}(w) + \alpha_{reg}R_{L_2}(w) + \beta L_{V_f}(w), \text{ where } L_{V_f}(w) = \frac{1}{|\hat{Y}|}\sum_{\hat{y}\in\hat{Y}}|V_f(w|\hat{y}) - V_f^*| \quad (5.3)$$

5.3. Hyperparameters

The hyperparameters are the parameters of the model that cannot be optimized during the conventional training process. They differ from the weights of the layers of the model, because of their special nature, and because it is not possible to find a gradient of the loss function with respect to them. However, choosing them is important to find the best model, and in order to find them *hyperparameter optimization* is used.

The model had a number of hyperparameters, including:

- batch size

- volume fraction penalty constant
- regularization parameter
- dropout probability
- learning rate

The number of training iteration was also treated as a hyperparameter, which allowed us applying early stopping by choosing the best model after the validation. This also relates to the dataset chosen to train the model, especially its size, and a particular form of the loss function.

We decided to apply the *grid search* approach for hyperparameter optimization[14]. For that, we chose key values for every hyperparameter and trained model for every combination of those values and checked the evaluation parameter to find the best model.

We were choosing batch size values of $1, 4, 8$. For the dropout probability, we tried $0.5, 0.7, 0.9$. Learning rate was taken with value 10^{-4} since changing the value did not significantly change the outcome because of the adaptive nature of the optimizer. For the L_2 regularization, we tried several values of the multiplicative constant but stopped at single a value of $\alpha_{reg} = 10^{-6}$ because for other values the results were significantly worse. We also tried two variations of the loss function, with and without volume fraction penalty term, and we put this variation as a separate experiment. For this, we checked several values of the multiplicative constant β.

After some preliminary research, we chose to take training datasets with four patterns described in Section 4.3.2. For each of them, we took three sampling steps, therefore three different sizes of the training dataset. For the most of the experiments, we have models trained for a various number of optimization steps. We chose the range of $(10, 25, 50, 75, 100)$ thousand iterations. For some cases, we took the second round of optimization with respect to early stopping, taking the number of iterations with steps of 5 or 10 thousand steps.

5.4. Implementation

The system was designed using the Tensorflow framework. It is built in form of a Python program calling the Tensorflow Python API. We used the 1.8 API version of Tensorflow run on GPU using CUDA 9.0 and CuDNN 7.0. The main reason of the choice of the Tensorflow and Python was the easy and fast way of building, modifying and understanding the code. These tools also give little overhead of creating a program that can be run using a GPU.

5.4.1. Tensorflow

Tensorflow is an open source library under Apache 2.0 license for high-performance numerical computations[2]. It provides API for multiple languages, including Python, and it

is easy to deploy on multiple platforms, including Nvidia GPUs. It was initially developed by Google's AI Organization and primarily supports the development of ML systems.

Tensorflow requires a definition of the pipeline of data transformations in the code. This definition is made in a functional manner using one of the high-level languages' API and includes various operations on numerical and other types of data, as well as easy means to update and optimize variable parameters using backpropagation. While definition, the programmer can specify the device on which a specific operation will be performed and a specific variable will be stored, as well as which variable to initialize and which to read at the end of the pipeline. During the pipeline descriptions, the user specifies a placeholder for the data on which operations should be performed. These placeholders are typically described as multi-dimensional arrays of certain data-type called *tensors*. Tensorflow takes care about compiling the optimized code described using the API and the deployment of the code and the data on the specified device. Then the user can run the defined operations for a concrete data using the mapping of the real input values with the input placeholders, the queue runners or iterators of some user predefined data set. In such a manner, one is able to define loops in which parameters of the model are trained, as well as to use the model to infer results for a certain input. Tensorflow allows saving trained model parameters in an easy way, which, in the end, helps to deploy the working model on different platforms using only the script describing the pipeline and the file with model parameters values. The other benefit of the Tensorflow is the *Tensorboard* utility with a web-based interface, which allows runtime observation over the system status with which user can see current values of any variable, like the value of loss function or intermediate output results, without almost any intrusion into the code.

Such design of the framework with API in multiple languages together with intrinsic optimization for high performance at multiple platforms, as well as convenient tools for easy deployment and control over the system makes Tensorflow extremely useful for many scientific and industrial applications.

5.4.2. Scripts

In order to quickly infer results that could be easily used for further TO and to proceed various experiments on performance, a software framework was created. The requirements of the software system were to be able to take various known cases of TO problem solutions, train the best CNN inference model and create approximate solutions for new TO problems. The framework was implemented in a functional way as a sequence of scripts and has the form of following algorithms presented in the pseudocode.

The first part of the system is responsible for automatic datasets generation. The outcome of a single run of the process described in Algorithm 1 is a number of datasets populated by data samples that could be directly fed to the network for further training or evaluation. Every dataset has different force sampling patterns, cases for different BC,

Algorithm 1 Dataset generation process

for all TO problem parameters values combination $p = (BC, F, V_f, Res.) \in S_{TO}$ **do**
    Generate a reference output y i.e. resulting material layout
    Start a sample $s(p)$
end for
for all Samples s **do**
    Augment Data
    Parse the case name
    Create an input images x_{BC}, x_{F_x}, x_{F_y}
    Pre-process inputs i.e. create SDFs of x_{BC}, x_{F_x}, x_{F_y}
    Combine inputs and form network-fedible sample s^*
end for
for all Dataset sampling schemes **do**
    Form training dataset S_{tr}
end for
Create evaluation dataset S_{ev}

volume fractions V_f and resolutions. Every sample contain images describing the BC, the load and the resulting material layout.

After we obtained different training datasets in the proper format, we can start training ANN models. The process of training the models is described in Algorithm 2. The result of the training process are models' checkpoints saving parameter values.

Algorithm 2 Model training process

for all hyperparameter values combinations $h = (h_1, ..., h_{n_H}) \in$ the hyperparameter values grid $H_1 \times ... \times H_{n_H}$ e.g. (training data set, number of training steps) **do**
    Create batch queue
    Set up the architecture $\mathrm{CNN}(w_0|h)$ and the loss function $L(y_1, y_2)$
    while Current number of training iterations not exceeded **do**
        Get a batch from the queue and read the input samples (x, y)
        Propagate the input $x = (x_{BC}, x_{F_x}, x_{F_y})$ forward and infer the result $\hat{y} = \mathrm{CNN}(x)$
        Find the loss function value $L(y, \hat{y})$ and propagate gradients $\nabla_w L$ backward
        Update parameter values $w_n := w_{n+1}(\nabla_w L)$ using Adam optimizer
        Save the state of parameters and the loss function value
    end while
    Save the parameter value checkpoint $(w|h)$
end for

After that, we validate created models and obtain the MSE measures and other data for models' evaluation. We chose the best model based on the average MSE and, in our case,

use it to infer results that will be used as an input for the IDeAs in order to evaluate how we can speed up the TO solution. The evaluation process is described in Algorithm 3.

Algorithm 3 CNN model evaluation and testing processes

for all model checkpoint $(w|h)$ **do**
  Load evaluation dataset S_{ev}
  Set up the architecture $\mathrm{CNN}(w|h)$
  for all evaluation samples $(x, y) \in S_{ev}$ **do**
    Infer results $\hat{y} = \mathrm{CNN}(x)$ and find loss function $L(y, \hat{y})$
    Record data for further evaluation e.g. MSE and the inferred output
  end for
  Find evaluation measures value, e.g. $\mathrm{MSE}_{\mathrm{avg}}$ and the error heatmap
end for
Choose the best performing model as $\mathrm{CNN}^* = \mathrm{argmin}(\mathrm{MSE}_{\mathrm{avg}})$
Use the model CNN to generate results to compare with the test set S_t
if 3D case **then**
  for all chosen test subset S_t^* results **do**
    Post-process i.e. smooth and reach desired volume V_f
    Initialize the TO model
    Run the TO process
    Evaluate the problem size and time to optimize
  end for
end if

The outcome of this process is the best model chosen and results produced by it for the testing. In the result, we give an estimation of how accurate is the best model for the current experiment set-up and how does it fit to facilitate the overall TO process.

6. Output Post-processing

This chapter describes the ways to evaluate the quality of the model. It also discusses typical issues revealed by the first evaluations and the ways to analyze and mitigate the flaws of the model. In particular, the chapter proposes a post-processing method, that brings the outputs to physically feasible shapes and facilitates restoration of the desired material volume.

6.1. Evaluation Means

The evaluation metrics chosen was an average Mean Squared Error (MSE) between the inferred and ground-truth material layouts, taken in form of arrays, averaged for the testing dataset. As a testing dataset, apart from several separately designated cases, we took input-output pairs for every single load case possible and with other parameters varying in the same way as in the training dataset. The variance of MSE was also analyzed to estimate the level of generality of the model.

Separate cases were analyzed visually to define the issues of the model more precisely. For this, for every test run, a pixel-wise difference of the ground-truth and inferred images was created, as shown in Figure 6.1.

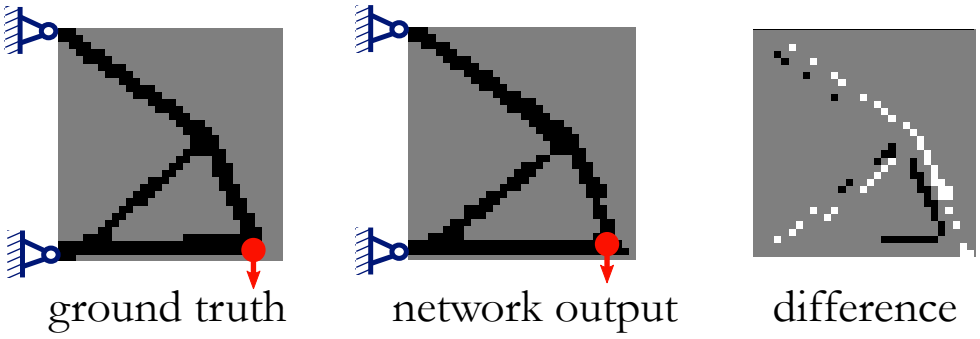


Figure 6.1.: A comparison of a ground-truth and an inferred image.

For the purpose of visual analyzes of dependency of accuracy on the location of force application in comparison to known cases, *error heatmaps* were created. They have the same size as the TO problem domain and every their pixel shows an error of inference for a test case for which force is applied at this coordinate of the domain. A more intense color

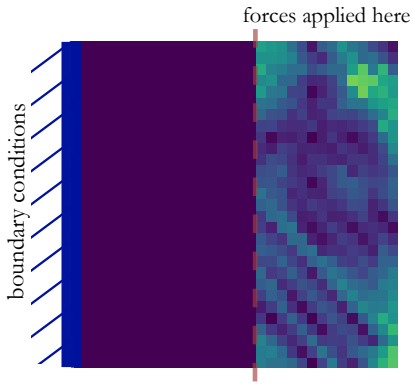


Figure 6.2.: An error heatmap generated during the evaluation. The intensity of every pixel denotes the value of error for the corresponding coordinate of force application. We perform the evaluation for TO cases for every possible force, here forces of direction $\mathbf{f}(x, y) = (-1, 1)$ are considered.

of the heatmap indicates that when the force is applied in this region of the domain, the model has a higher error, as shown in Figure 6.2. Such heatmaps were created for every force direction and BC, as well as for the average error across all the force directions.

For a more precise evaluation of the model accuracy for the 2D case, a small program was visualizing created using HTML, CSS, and JavaScript. This program, for a single evaluated model, shows error heatmaps for every force direction and BC. It also shows the concrete ground-truth, inferred image, and their difference, based on the mouse position. The GUI of the program looks like shown in Figure 6.3. This allowed observing how material layouts, obtained by both reference TO process and the ANN model, change with smooth movement of the force application coordinate, and define what are the principal groups of TO results for a similar setup.

We also chose to analyze the dependency of the mean and variance of MSE for different training datasets as well as depending on the training steps to enhance early stopping and choose the best model.

6.2. Physical TO Model Requirements

This section describes the issues concerning the outputs of the network that were observed before performing final tests of the model. These issues and their nature gave insights about the flaws of the models during the work. The outputs of the model demonstrate several recurring issues related to the material layout having incorrect physical properties. The need to mitigate them led to some decisions about additional data processing made during the design of the ANN system.

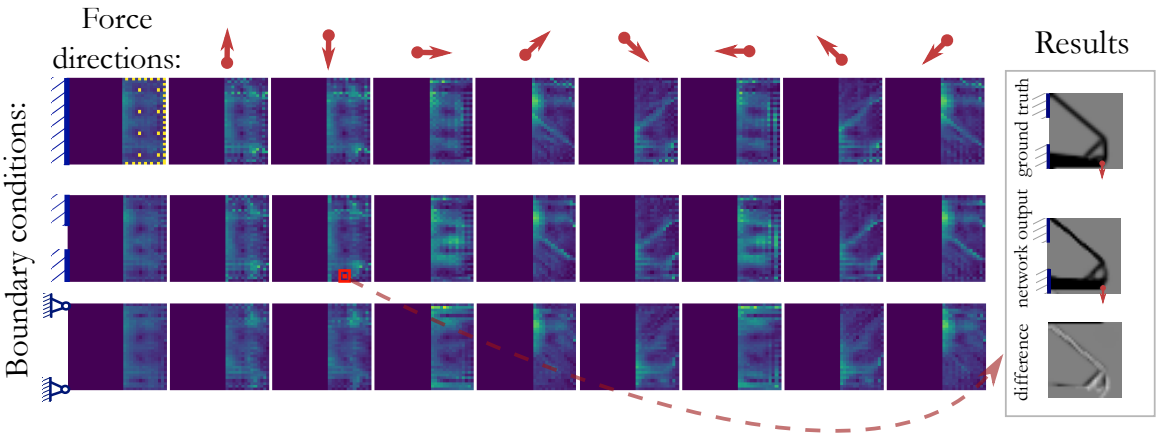


Figure 6.3.: A program for results evaluation. Here error heatmaps are shown for different BC (rows, BC shape indicated on the left) and forces directions (columns, directions indicated at the top) for a model trained for 25k steps on a set with three different BC, constant volume fraction $V_f = 0.2$, and for forces applied as indicated in the upper left heat map with yellow points. Here we can see every case for which accuracy is unsatisfactory (lighter shades of the heatmap). We chose to demonstrate the difference between the ground-truth and the inferred result for the case of BC of corner points, and a force $(1.0, 0.0)$ applied at $(23.0, 31.0)$.

One of the issues was the material layout not corresponding to some topological requirements, that would be automatically met during a conventional TO process. These issues include violating physical properties, such as :

- structural elements of the layout being not connected
- small non-contiguous domains occupied by material
- the material having the volume not equal to the desired one

Even having a physically meaningful shape, requirements on the material volume will not be automatically met and this needs an additional handling.

Despite having some issues for selected ill-posed TO problems, most of the optimized models were able to preserve the main structural requirements on the element shape for most of the cases, which means having the body of the element connecting the BC and load application point. This means that the cases with the aforementioned issues could be potentially treated with a simple and computationally cheap post-processing approach based on some local visual properties of the output.

Another issue related to structural requirements on the resulting shapes was discovered by using the interactive result visualization program. During the first trials on the 2D case,

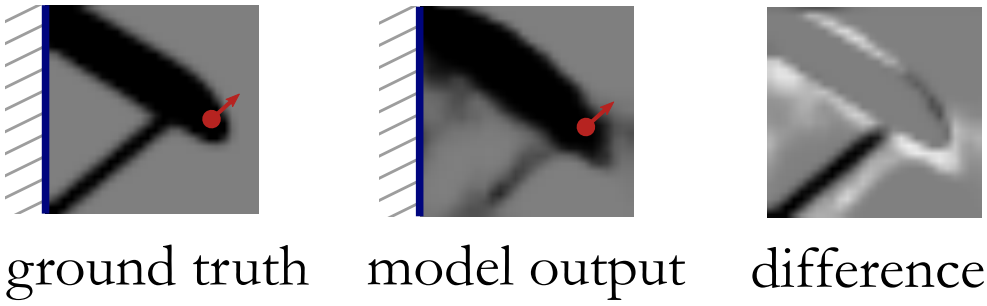
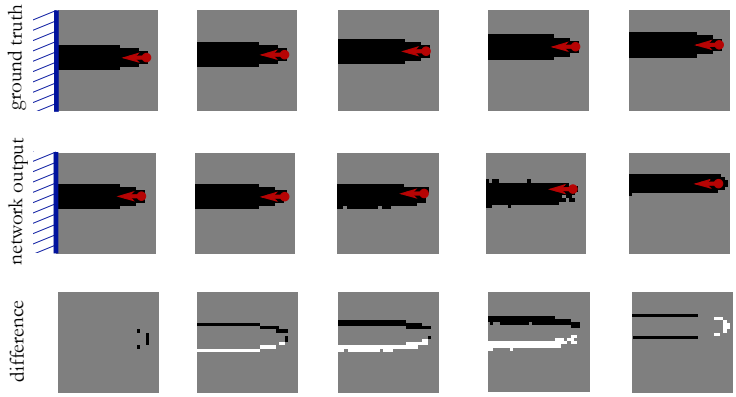
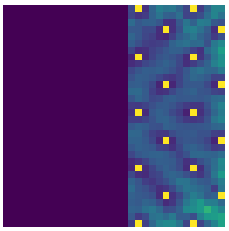


Figure 6.4.: An example of the case when the model output has a wrong volume fraction and physically incorrect parts.

a model was prepared using a training dataset for which forces were applied at points creating a checkered grid. The evaluation results did not satisfy the requirement and were not able to reconstruct the relationship between BC and load application point, as it is shown in Figure 6.5. This problem led to future research on model configurations, in particular, choice of sampling patterns for the training dataset.



(a) With a smooth change of force application coordinate, the result does not change smoothly and tend to leap between known cases



(b) The training dataset sampling pattern and the error heatmap

Figure 6.5.: A recurring problem of incorrect structure reconstruction based on load application point observed for nonoptimal training dataset sampling.

6.3. Post-processing Approaches

In order to mitigate the issues described above, we decided to perform several post-processing operations on the output of the model. We decided to resort to mathematical morphological operations in order to solve these issues.

We used two basic operations, namely *erosion* $\ominus$ and *dilation* $\oplus$ [30]. The erosion is defined as:

$$A \ominus B = \{z \in E | B_z \subseteq A\}, \text{ where } B_z = \{b + z | b \in B\}, \forall z \in E \tag{6.1}$$

Here A is the initial set of points, B is a structuring element, and E is the domain space for both. This essentially can be interpreted as cutting a layer constructed out of elements B from every contiguous shape of the image A.

The dilation is defined as:

$$A \oplus B = \{z \in E | (B^s)_z \cap A \neq \emptyset\}, \text{ where } B^s = \{x \in E | -x \in B\} \tag{6.2}$$

Similar to erosion, this could be interpreted as adding a layer constructed out of elements B to every contiguous shape of the image A.

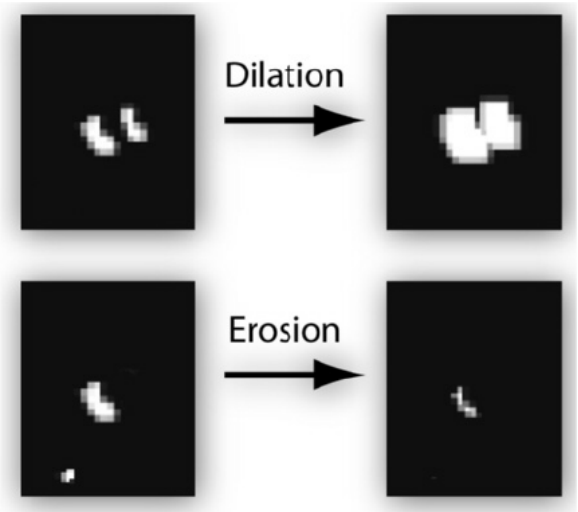


Figure 6.6.: The results of morphological dilation and erosion[23]. The former one increases the volume and may connect regions. The latter one decreases the volume and may eliminate small regions.

In the result, the post-processing sequence consists of the next steps:

- *Apply Morphological Opening*: Firstly we apply erosion to the image by small ball element B and then we apply dilation to the image by the same element. In the result, all domains of connectivity smaller than B will vanish.
- *Apply Morphological Closing*: Firstly we apply dilation to the image by small ball element B and then we apply erosion to the image by the same element. In the result, all gaps smaller than B will be closed.

In the case when the volume fraction of the result is larger than desired, which in practice happened in most of the times, we tried to eliminate the elements that have the lowest probability to be occupied with the material. For that, we represent the resulting image as a sparse matrix in form of a coordinate list, meaning that matrix is read as a list of tuples $(\mathrm{row}, \mathrm{column}, \mathrm{value})$ and could be easily sorted by value. With that, we delete the lower values until we reach a volume fraction value close to the desired one.

As a result, using these approaches we were able to bring the resulting model to a more physically feasible shape and to ensure that its volume has the desired value.

Part IV.

Results and Conclusions

7. Results

This chapter describes the tests on accuracy and performance for different ANN models inferring TO outputs. The results of the evaluation are given for cases with both 2D and 3D domains. The chapter analyzes the dependency of the model performance on the training dataset properties. The models' ability to interpolate results for different variable parameter values is tested. The chapter describes different methods to prepare the model with the best performance based on several criteria. The models are evaluated with respect to the accuracy of their results. The models' ability to improve and accelerate the TO process is also evaluated.

The models for all cases are trained using a GPU. The same GPU is also used to generate samples and measure TO speedup and for 3D cases. The graphics card that was used is a single Nvidia GeForce GTX 1080 Ti, with 3584 CUDA cores[9].

7.1. Results for the 2D case

During the first practical part of the work, only the 2D cases of TO was considered. For this case, it was possible to generate a training dataset with parameters that vary more and train models that are able to infer results for a more general case. Furthermore, the training process for 2D data takes less time comparing to the 3D case. A training set for the 2D case includes samples with varying boundary conditions, material volume fractions, domain resolutions. The test are performed to analyze the model's ability to capture these parameters.

Test I: Influence of training dataset size and sampling pattern

One of the main tests' goal was to find the influence of the training dataset properties on the model performance. The evaluations were done for models trained with datasets of different size and of different sampling patterns, described in "Data Preparation" chapter. The goal of the test was to find the dependency of the model's validation accuracy on the training dataset size as well as to choose the best training dataset sampling pattern.

After some preliminary research, the decision was to take four different sampling schemes with three different sampling intervals for each. The models for various numbers of iterations were trained and we can observe the saturation of the validation error after some point (10^4 training iterations) as it is shown in Figure 7.1. However, the case where most

of the loads are applied at the edge of the domain, as in Figure 4.4b, shows slightly better results compared to others, even with smaller amount of training samples.

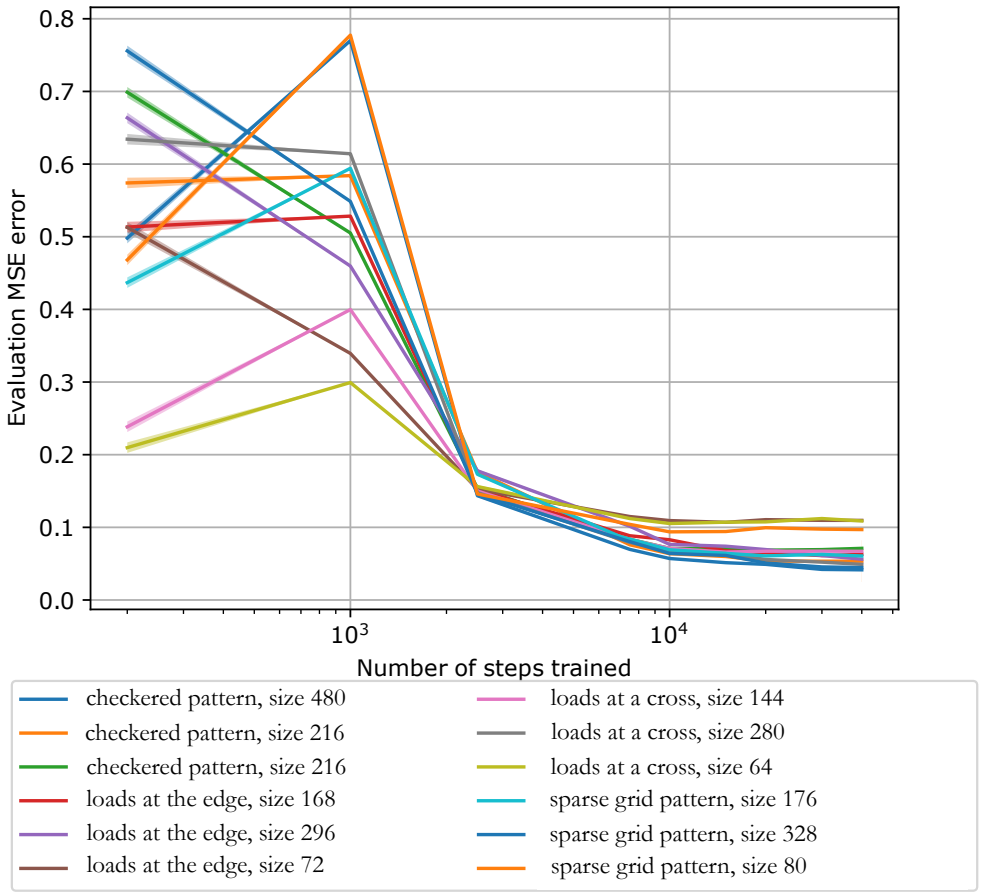
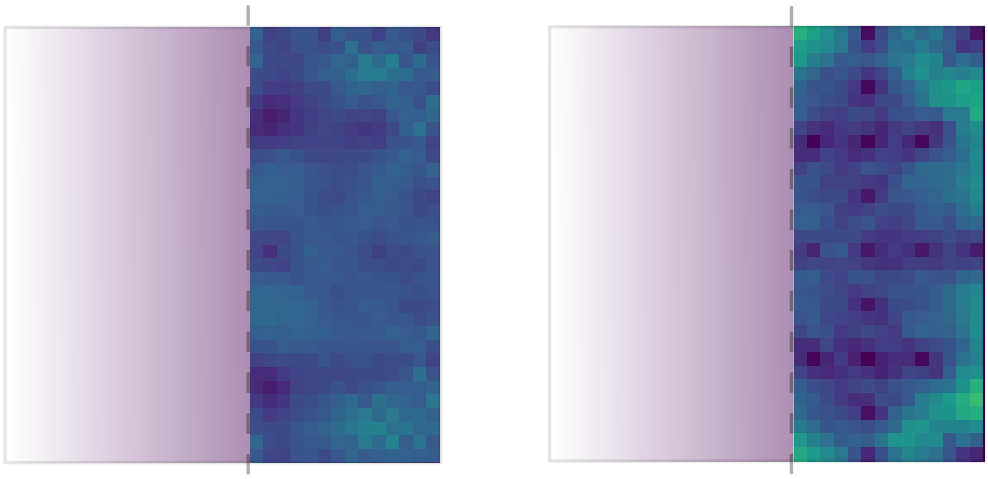


Figure 7.1.: The dependency of the model validation error on the number of training steps.

We can also observe different patterns occurring in the validation error heatmap. The longer the training was performed, the less uniform the error is with respect to load application point for the test case. For the coordinates of loaded points for known cases and their vicinity, the error value becomes very low, while it increases for other points. However, as it seen from the validation error graph in Figure 7.1, the average error stays almost the same after some number of training iterations. The validation technique requires a more general approach that will consider uniformity of the error across the test dataset.

The validation error decreases almost equally for every sampling pattern with the increase of training dataset size and it is shown in Figure 7.3. The dependency of the error on the training dataset size $|S_{tr}|$ is almost the inverse proportion $\mathrm{MSE}_{\mathrm{avg}} \sim |S_{tr}|^{-1}$.



(a) Number of iterations $15,000$ (b) Number of iterations $40,000$

Figure 7.2.: Validation error heatmaps after different amounts of training steps. With training, the error distribution based on the load application point becomes less uniform. The error becomes more dependent on how close are the known cases that were used during the training.

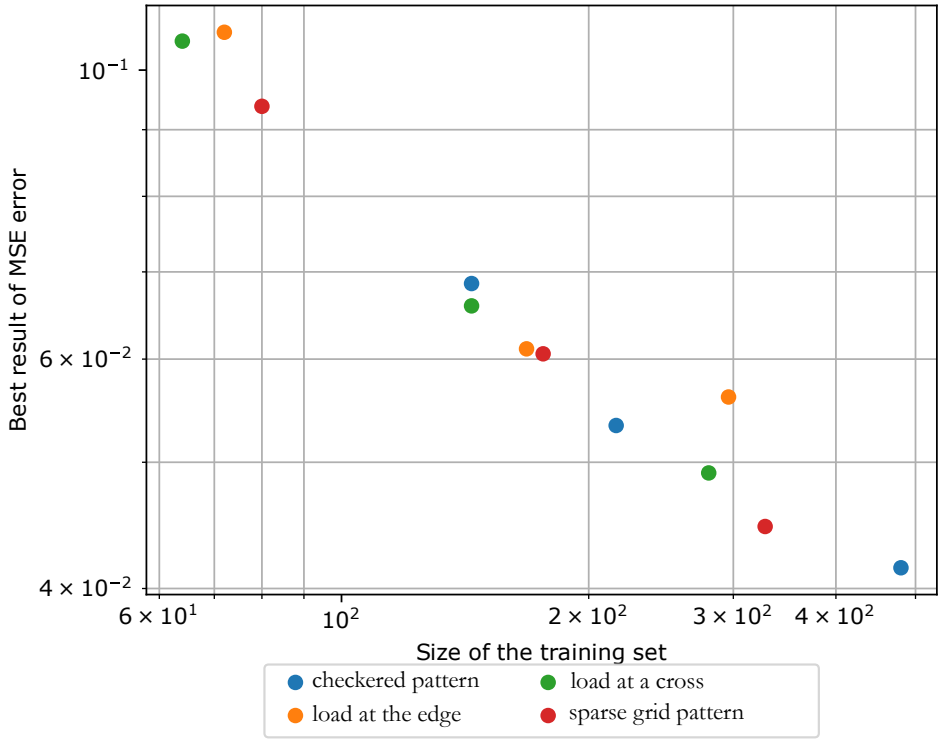


Figure 7.3.: The dependency of the model validation error on the training dataset size.

Test II: Loss function form and volume restoration

This test was performed in order to evaluate the capability of the approach to preserve the proper volume fraction of the material in the final model layout. For this, models were trained using different loss functions:

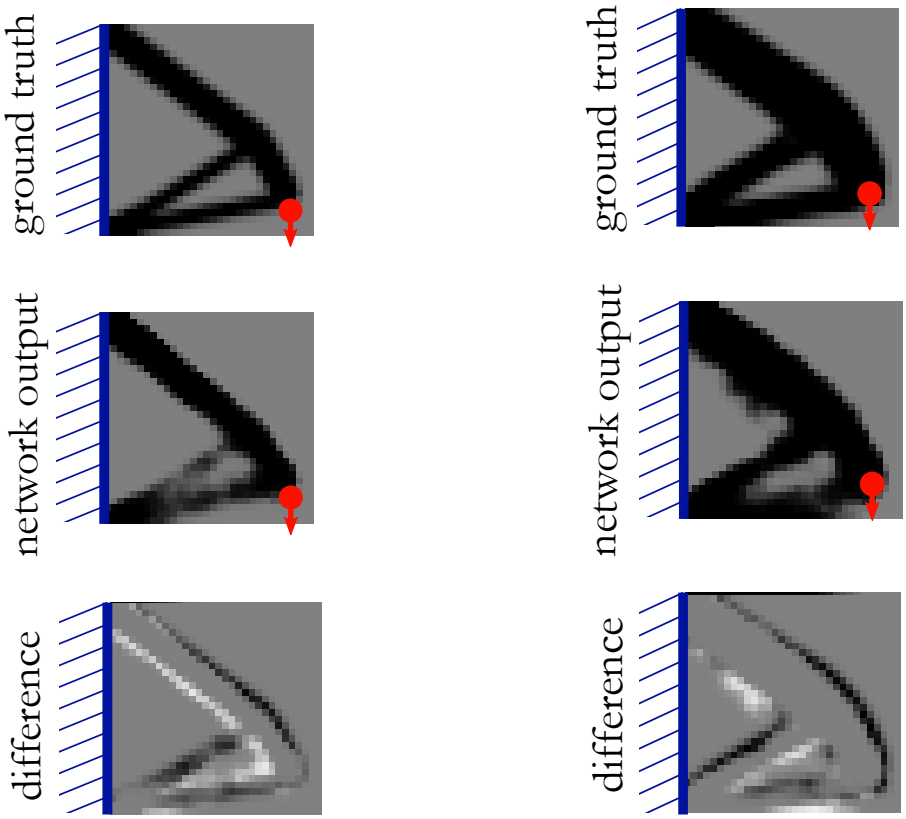
- without volume fraction treatment as $L_{(1)}$ in Equation 5.2
- with additional volume fraction penalty term as $L_{(2)} = L_{(1)} + \beta L_{V_f}$ in Equation 5.3
  - for different values of the multiplicative constant β

We took training datasets with label layouts of variable volume with three different volume fraction values. For the evaluation, we used datasets composed of both layouts with known volumes and with layouts of volume fraction values not used during the training. We measured both the general accuracy of the model and the accuracy of volume fraction restoration as a separate parameter. The results show that considering volume fraction within loss function improves the accuracy of the model, as one can see from the Tables 7.1 and 7.2.

Loss vs. MSE(Var)	$V_f = 0.2$	$V_f = 0.3$	$V_f = 0.5$	$V_f = 0.3$, res. 28×28
$L_{(1)}$	0.063(0.0001)	0.072(0.0012)	0.097(0.0013)	0.168(0.006)
$L_{(2)}, \beta = 0.01$	0.060(0.008)	0.070(0.0013)	0.10(0.001)	0.156(0.006)
$L_{(2)}, \beta = 0.2$	0.058(0.0007)	0.068(0.0009)	0.092(0.001)	0.162(0.005)
$L_{(2)}, \beta = 1.0$	0.063(0.001)	0.073(0.0009)	0.094(0.0008)	0.177(0.005)

Table 7.1.: Test results evaluating influence of considering the volume fraction at the loss function. The loss function $L_{(1)}$ is a default loss as in Equation 5.2 and the loss function $L_{(2)}$ has an additional volume fraction penalty term multiplied by β as in Equation 5.3. Every model was trained on the same training dataset. For every model, average MSE and its variance are calculated. Test datasets with a known (0.2), intermediate unknown (0.3) and larger unknown (0.5) desired resulting volume fractions V_f are used.

For the test dataset with an intermediate volume fraction value $V_f = 0.3$, the model using the loss with volume fraction penalty term L_{V_f} with a multiplicative constant $\beta = 0.2$ had an average error of $\text{MSE}_{\text{avg}} = 0.068$ and average volume error $E_{V_f} = 0.017$, comparing with 0.072 and 0.02 for the model without, respectively. Testing for unknown intermediate values of the input volume fraction shows that the model is producing accurate results.



(a) $V_f = 0.3$ (b) $V_f = 0.5$

Figure 7.4.: Results for the network trained using the loss with volume fraction penalty term for interpolating and extrapolating outcome volumes.

Results for the input volume fraction being larger than the ones used for training, however, show a worse accuracy of the models, which indicated that it is hard to extrapolate the outcome volumes. In this case, for the best model, the error was $\mathrm{MSE}_{\mathrm{avg}} = 0.092$ and the volume error was $E_{V_f} = 0.064$. This could be explained by convolutional nature of the model; with the fact that features found by the model are too subtle and do not fit to construct more massive outcome layouts. The application of the post-processing increased the accuracy and improved the volume fraction reconstruction, so that volume error E_{V_f} decreased in about 0.5 times.

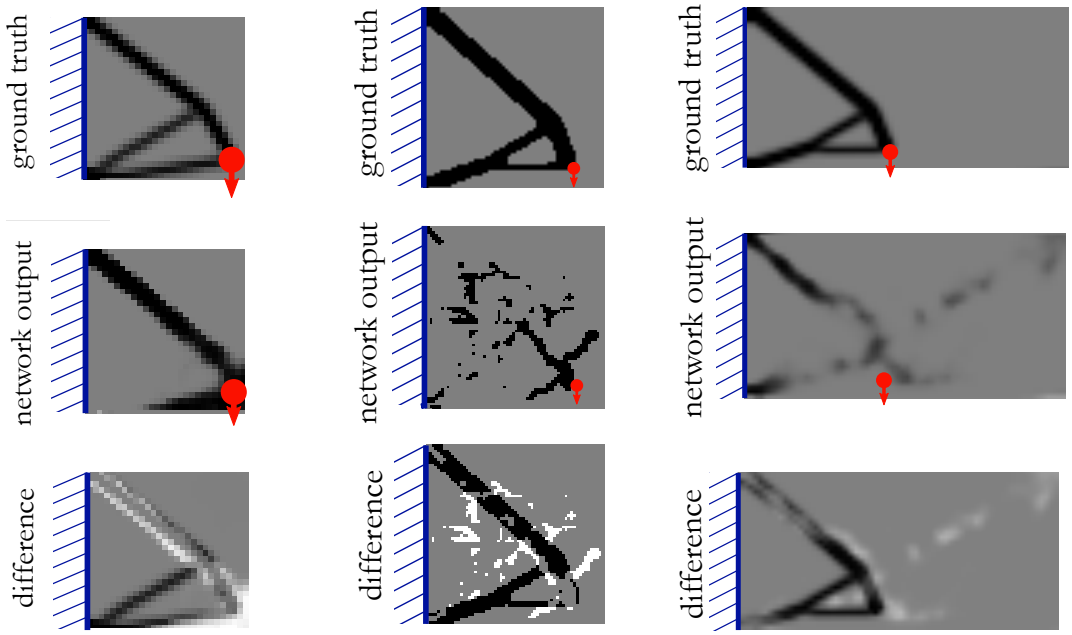
Test III: Scaling problem resolution

This test had a goal of evaluation of the capability of the network to scale the results for different resolutions. In order to achieve this, we prepared training dataset with resolutions of previously unknown sizes. For an intermediate size of the domain, the model shows a

Loss vs. E_{V_f}(Var)	$V_f = 0.2$	$V_f = 0.3$	$V_f = 0.5$	$V_f = 0.3$, res. 28×28
$\beta = 0$	0.015(0.0001)	0.020(0.0002)	0.058(0.0004)	0.054(0.001)
$\beta = 0.01$	0.0017(0.0002)	0.019(0.0002)	0.007(0.0003)	0.044(0.001)
$\beta = 0.2$	0.012(0.0001)	0.017(0.0002)	0.064(0.0003)	0.046(0.001)
$\beta = 1.0$	0.011(0.001)	0.031(0.0004)	0.067(0.0003)	0.07(0.002)

Table 7.2.: Test results evaluating the influence of the volume fraction penalty term. Here the ability to reconstruct the correct resulting volume is evaluated.

good accuracy of the interpolation. For the larger resolutions, however, the ability to extrapolate was unsatisfactory. For an additional case, we increased the size of the domain, without changing any other properties of TO problems. The result for this case was also unsatisfactory with empty part of the domain being populated with pixels having no physical meaning. The results can indicate that it is crucial for the decoding part of the model to be trained using a sample with more massive features. That also indicates that the input pre-processing require some changes to be different for varying domain resolutions and side length ratios.



(a) Resolution 28×28 (b) Resolution 64×64 (c) Resolution 32×64

Figure 7.5.: Results inferred by the model for domain resolutions not used for training. In the case of 28×28 resolution, results are meaningful, but with a low accuracy.

Test IV: Comparison with other architectures

The goal of one of the tests was to compare the performance of different network architectures. We compared our encoder-decoder residual network with other similar architectures, that also have a different number of parameters:

- *V-Net*: $9,007,601$ parameters
- *U-Net*: $2,028,385$ parameters

In comparison, our model has $4,011,585$ parameters. The U-Net network did not converge and was producing noise for any considered model configuration. The V-Net achieved slightly better results for the validation dataset composed of cases with known resolution and volume. However, for the data with new intermediate values for the size of the domain and volume fraction, the results were significantly worse, as shown in Table 7.3. Furthermore, the training of the V-net model took significantly more time, as shown in Figure 7.7.

Test Dataset	$32 \times 32, V_f = 0.2$		$28 \times 28, V_f = 0.3$	
Model	MSE(Var)	E_{V_f}(Var)	MSE(Var)	E_{V_f}(Var)
V-net	0.056(0.0008)	0.013(0.0001)	0.180(0.003)	0.085(0.0005)
U-Net	did not converge		did not converge	
Our Architecture	0.058(0.0008)	0.012(0.0001)	0.162(0.005)	0.046(0.001)
Our Architecture (w.o. SDF)	0.048(0.0009)	0.011(0.00006)	0.172(0.004)	0.072(0.001)

Table 7.3.: Test results for different models.

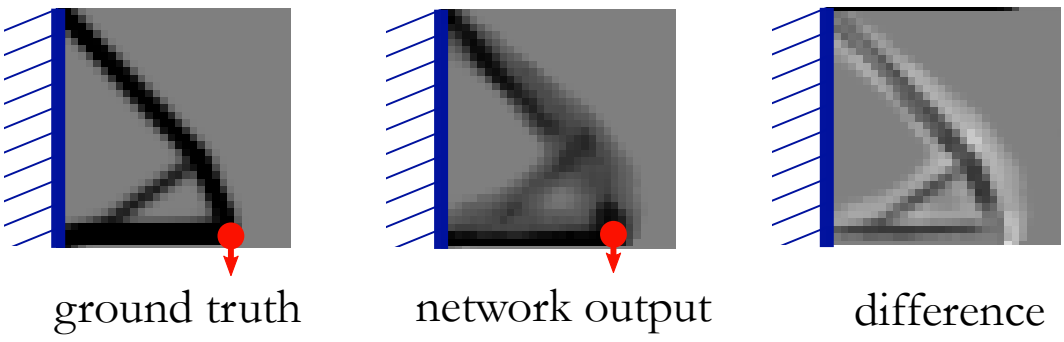


Figure 7.6.: A result produced by V-Net for a case with intermediate resolution and volume

Test V: Influence of pre-processing

This test was done to evaluate the influence of applying SDF for inputs as the pre-processing and to justify such an approach. For that, we prepared two training datasets, one with initial images describing BC and loads as inputs, and the second one with inputs after applying the pre-processing for every channel. Models for both training datasets were prepared. Then both models were evaluated for two test datasets. Test datasets were collections of TO cases for respective domain sizes and volume fractions:

- 32×32 , $V_f = 0.2$
- 28×28 , $V_f = 0.3$

The case without applying SDF shows worse results for the test dataset with data having different properties comparing to the training dataset. However, this model shows better results for problem size and volume fraction used for training, as shown in Table 7.3. The optimal model for a dataset without pre-processing also took more steps to train: 40,000 steps against 20,000 for the model with SDF application, as seen in Figure 7.7.

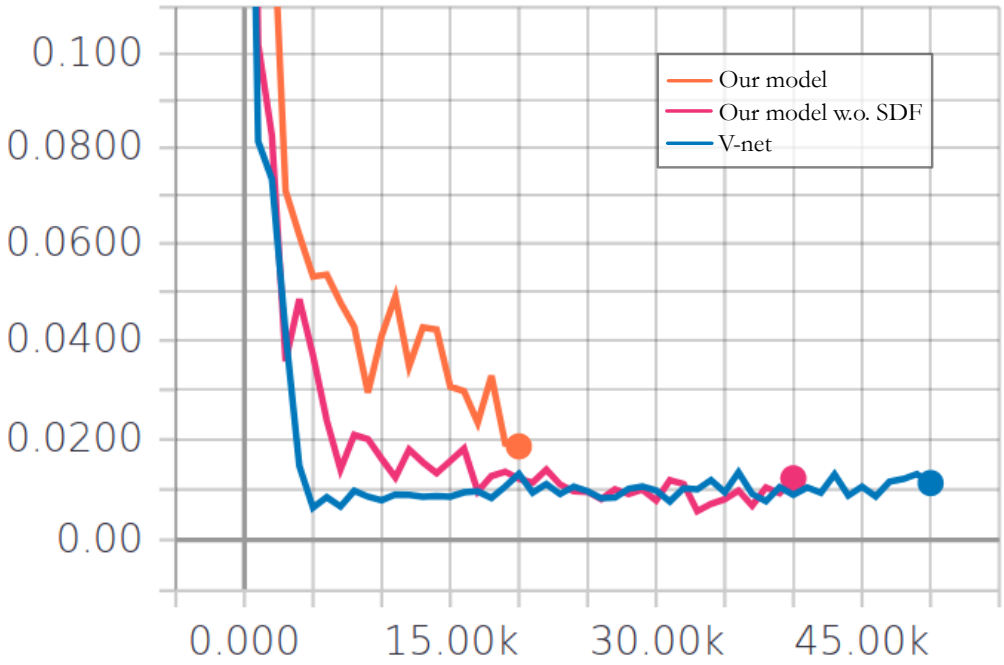


Figure 7.7.: Loss value depending on the training iteration for various models. The training graph is produced using Tensorboard.

7.2. Results for the 3D case

In case of a 3D domain, due to both smaller sample and more difficult perception of 3D data, we applied different visualization and evaluation methods. An average MSE across the test dataset was used to validate and choose the best model. The MSE variance and the error heatmap were also used to estimate the models' generality.

We also considered the influence of the inferred result on the conventional TO process performed by IDeAs. The first method of performance measurement was comparing the time required to perform TO for a single problem. As a reference value, we take the average time of a single IDeAs run for problems of a fixed domain size.

In order to use the output of the ANN for IDeAs, several adjustments had to be implemented. Firstly, we implemented the means to state a checkpoint VTK file, that describes the material layout. Secondly, we implemented a function that initializes the data structures of material density with values read from the VTK file.

For a different estimation of the TO process performance improvement, we measured how well the inferred result describe a bounding region of the ground-truth layout after an increase of the volume with dilation. We measure how many voxels of ground-truth layouts on average lie outside of the inferred bodies and the probability for ground-truth layout to be fully incorporated in the inferred result. The estimation of dilation size applied to the network output required to cover the real result up to statistical significance is given.

Test I: Finding the most accurate model for 3D data

The first experiment was to find the best model with optimized hyperparameters. We used the training dataset with sampling patterns similar to the 2D case. We pre-processed samples and the model considered the loss function that included volume fraction penalty term. After the training process, performed in the same way as the 2D case, the best model was produced by the training dataset with forces at the edge. It shows the average error of 0.14 on the evaluation dataset, as seen in Table 7.4.

Sampling vs. Number of iterations	$25k$	$50k$
Sparse grid sampling	0.048(0.003)	0.051(0.003)
Sampling at the edges	0.147(0.01)	0.140(0.003)

Table 7.4.: Validation results for models treating 3D data.

Typical results look like shown in Figures 7.8, 7.9, 7.10. The results meet the structural requirements on resulting material layouts, the requirement on volume fraction and mostly are shaped similar to ground-truth models.

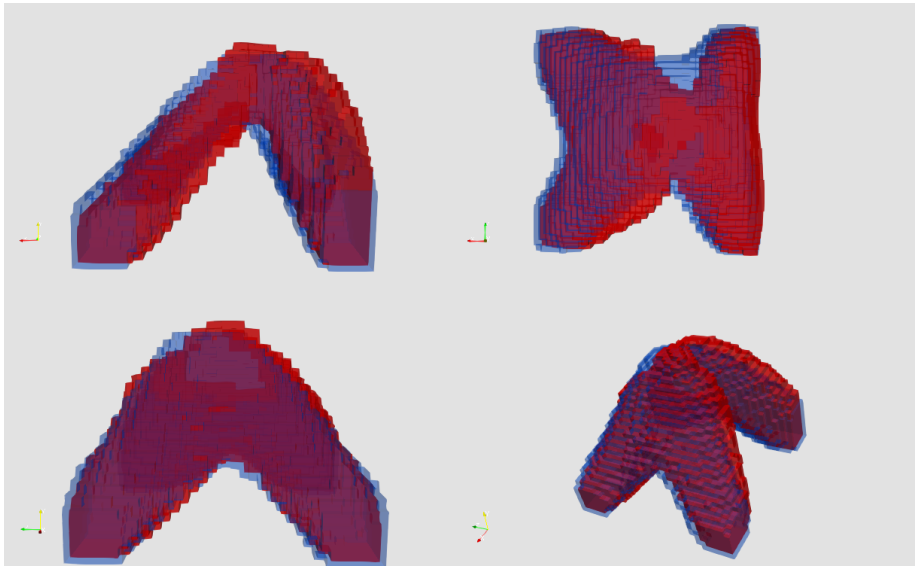


Figure 7.8.: An example output of the network (blue) compared to the reference model (red). The model correctly connects the BC (corners of the $X = 0$ wall) and the load application point, the overall shape is slightly distinct from the ground truth.

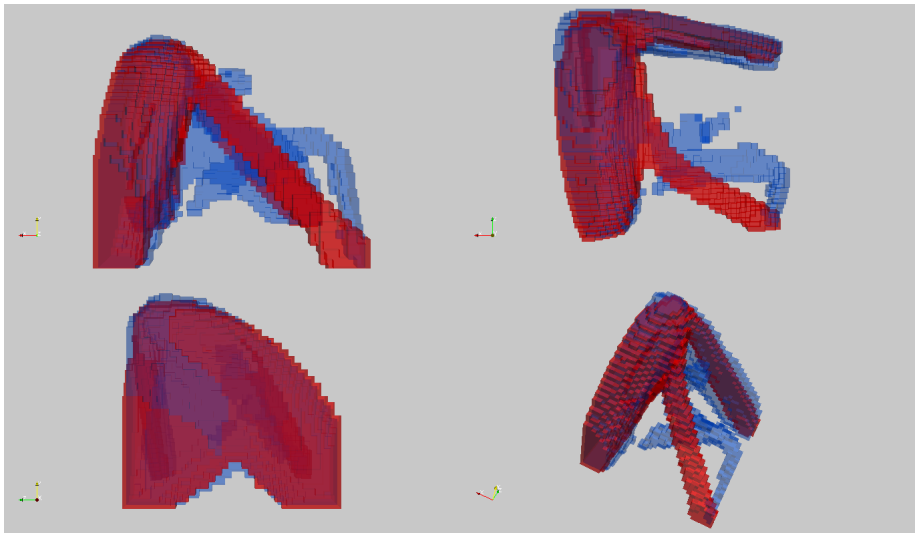
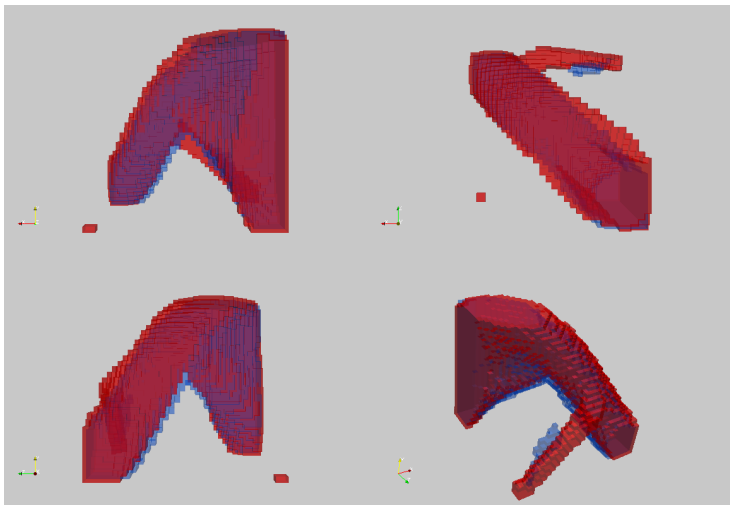
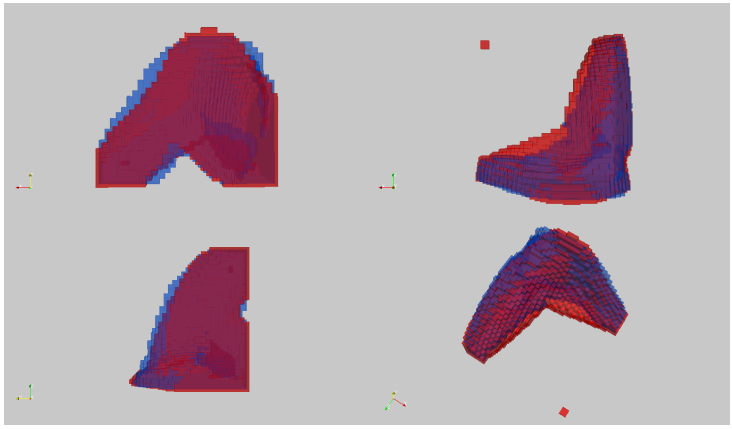


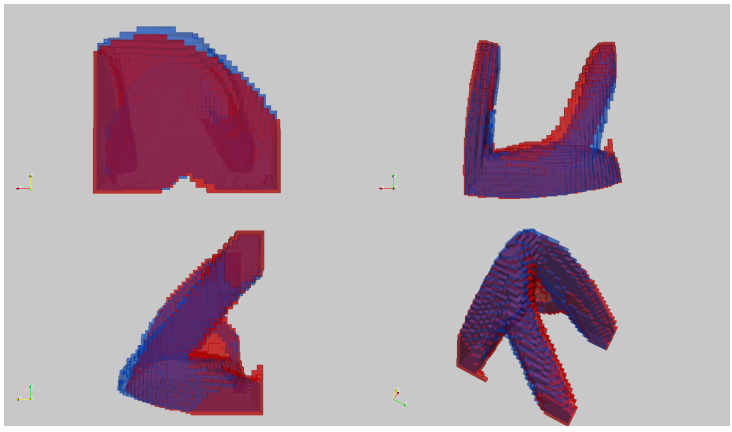
Figure 7.9.: An example output of the network (blue) compared to the reference model (red). The shape of one of the beams differ from the reference one, however, the rest of the structure is reconstructed.



(a)



(b)



(c)

Figure 7.10.: More example outputs of the network (blue) compared to the reference models (red).

Test II: TO acceleration with IDeAs

After applying layouts of the elements produced by the optimized model as the input for the IDeAs, we achieved the average duration of the TO process equal to $n^* = 10$ number of iterations. This number is almost the same as the number of iterations performed by the optimizer starting with the default initialization, which is equal to $n^* = 9$ iterations.

In the case of initialization with an approximate solution inferred by the ANN model, the objective function of the optimizer is already low during the first iterations. For the case of initialization with an educated guess, it has an average value of $L_{TO}(0) = 7.9$ on the first step, when the default initializer gives averagely $L_{TO}(0) = 250$ in absolute units, and the found minimum in both cases is averagely $L^*_{TO} = 3.0$. Starting in the vicinity of the minimum is an ill-posed case for current TO optimization approach, which means it is hard to utilize any educated initial guess to decrease the number of optimization iterations. The objective function for many TO problems might have multiple local minima with domains of a low gradient value in their vicinity, which will require more sophisticated optimization techniques for successful smoothing.

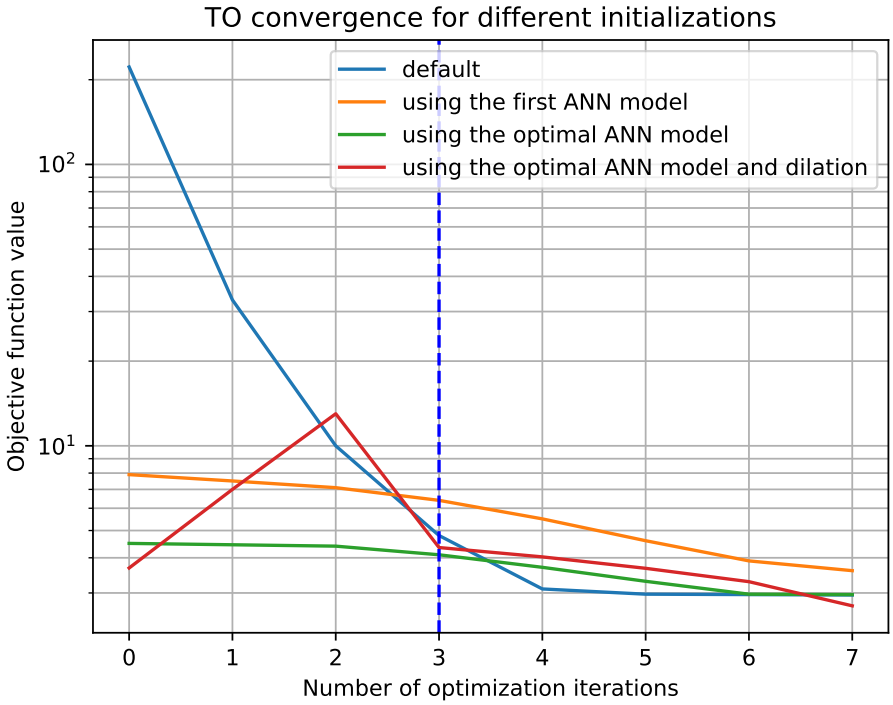


Figure 7.11.: The objective function L_{TO} of resulting material layout depending on the iteration number n of the TO process. Before the fourth iteration values are higher for the case with default initialization.

The convergence graph for TO with and without an educated guess is shown in Figure 7.11. For the default implementation, there is an initial phase during which the goal function L_{TO} magnitude decreases in two orders. Then the phase of fine-tuning follows, which takes approximately two times longer. For the case when an educated guess is used, the convergence enter the smoothing phase after the first iteration. However, due to the shape of goal function and a simple optimizer rule, this phase takes about 33% longer than for default case.

In the case when the optimization time is restricted, however, usage of the educated guess is beneficial. We restricted the maximum amount of optimization iterations to $n_r = 3$ and using our method led to a smaller value of the loss function for the resulting layout, as it is seen in Figure 7.15.

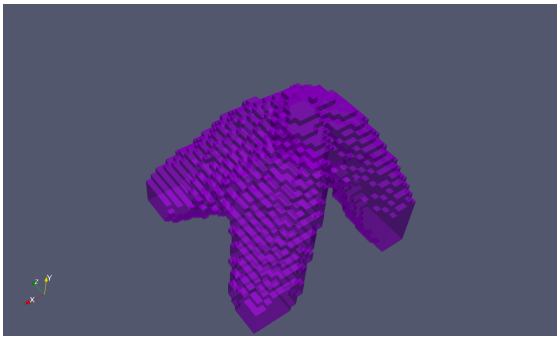


Figure 7.12.: With an educated guess, $L_{TO} = 4.37$

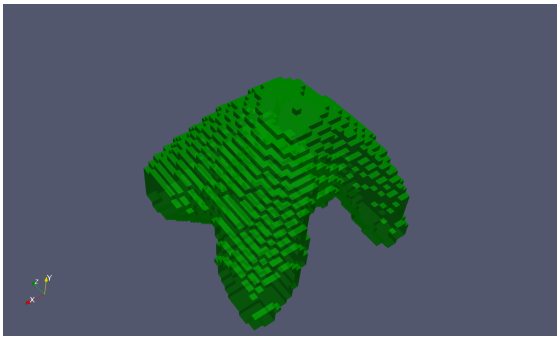


Figure 7.13.: With an educated guess after dilation, $L_{TO} = 4.35$

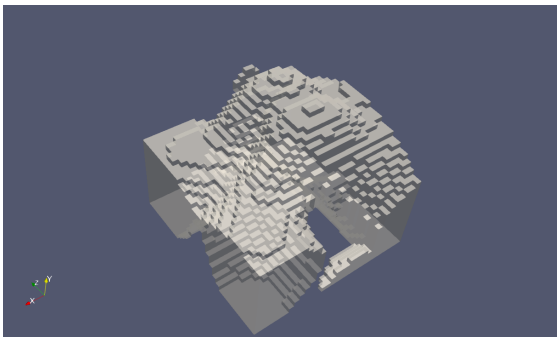


Figure 7.14.: Without an educated guess, $L_{TO} = 9.96$

Figure 7.15.: Results of the TO process after $n_r = 3$ iterations for different initializations. The images show the outcome material layout and the corresponding value of TO objective function L_{TO}.

Test III: Decreasing problem size

For a final test, we considered the possibility to make the ANN model's outputs cover the corresponding ground-truth layout. That is required to find elements that are for sure will be free of material in the final material layout. Using that, we can reduce the problem size by not taking the domain as a bounding box, but leaving out the excluded elements. We considered the cases with desired volume fraction $V_f = 0.2$. For such cases, we saw that there about 10% of ground-truth layout voxels sticking outside corresponding ANN models' output.

After averagely $n_d = 4$ dilations, we managed to make the network output cover the material layout of corresponding ground-truth model with the probability of 95%. With such a dilation, the output averagely increased the amount of its voxels from 22% to 49% percent of the whole domain, as shown in Figure 7.16. That means that it is potentially possible to decrease the TO problem size by 2 times without statistically significant errors because of domain restriction. This could be done by leaving out elements of FEM and TO problem during data structures definition, for which the ANN system predicts that they will be free of the material.

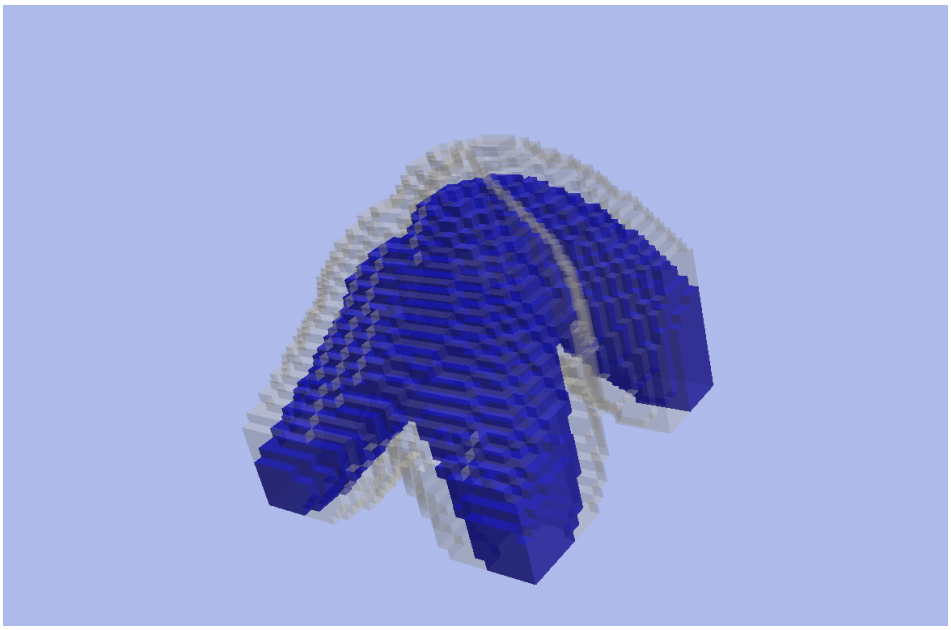


Figure 7.16.: A ground-truth layout (blue) covered by a dilated layout inferred by the ANN model (gray).

8. Conclusions and Future Work

In this work, we presented a Convolutional Neural Network model to infer approximate results of the Topology Optimization process. We proposed a CNN architecture and its implementation in Python using Tensorflow, together with the infrastructure for models' training, optimization, validation, and data preparation. The approaches for model's input pre-processing and output post-processing are proposed and their influence is analyzed. The model's dependency on the training dataset is analyzed and the comparison of the performance with other architectures is given. Furthermore, we analyzed the model's ability to treat problems with varying parameters, including material volume fraction and domain resolution. Finally, the model's result influence on the TO process performance is evaluated.

The presented approach is novel for such a problem and it is the first data-driven model suiting the 3D TO simulation approximation. The method shows a good accuracy of the inferred results of the TO. The process of producing a single solution by the ANN model takes significantly less time than in the default case. Being highly computationally expensive, TO is able to use the inferred results as an educated guess for the initial material layout for further simulations. Applying a model's output as an initial material distribution, it is possible to obtain an accurate resulting layout after only a couple of smoothing TO iterations. The program designed by us suits well for producing accurate models that could prepare preliminary data as input for the TO software in an automated manner. Using our model, we were able to significantly decrease the size of the Finite Element Analysis problem solved at every iteration of the TO, which potentially can bring a great speedup for the algorithm.

In the future work, we would like to obtain models trained on a dataset with more varying data, possibly being able to treat arbitrary boundary conditions, loads applied and domain sizes. This model should be optimized and validated with a more rigorous approach, considering more hyperparameters and the error dependency on the concrete TO case. Finally, using outputs of a better and more general model, a more advanced test on TO acceleration should be done. This would require introducing means for TO to perform FEA for arbitrary domains defined in runtime and considering more complex cases of TO problems.

Bibliography

[1] NumPy. http://www.numpy.org, version 1.14.5.

[2] Tensorflow. https://www.tensorflow.org, version 1.8.0.

[3] Industrial automation systems and integration – Product data representation and exchange – Application protocol: Managed model-based 3D engineering. Technical report, International Organization for Standardization, Geneva, CH, 2014.

[4] ECMA-404 – the JSON data interchange syntax. Standart, 2017.

[5] Nikolai Bakhvalov. On the convergence of a relaxation method with natural constraints on the elliptic operator, 1966.

[6] Martin Philip Bendsoe and Ole Sigmund. Topology optimization: theory, methods and applications, 2003.

[7] T. Chan and W. Zhu. Level set based shape prior segmentation. San Diego, USA, June 2005. IEEE.

[8] Djork-Arne Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by Exponential Linear Units (elus), 2015.

[9] NVIDIA Corporation. GeForce Gtx 1080 Ti specifications. https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/. accessed 19 July 2018.

[10] Yann N. Dauphin and David Grangier. Predicting distributions with linearizing belief networks, 2016.

[11] A. Dosovitskiy, J. T. Springenberg, and T. Brox. Learning to generate chairs with convolutional neural networks. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1538–1546, 2015.

[12] J. Gausemeier et al. Thinking ahead the future of additive manufacturing analysis of promising industries, 2011.

[13] Stefan Gavranovic. Topology optimization using GPGPU. Master's thesis, Technische Universität München, 2015.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[15] Xiaoxiao Guo, Wei Li, and Francesco Iorio. Convolutional Neural Networks for steady flow approximation. 2016.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. Las Vegas, USA, 2016.

[17] Oliver Hennigh. A Tensorflow re-implementation of the paper Convolutional Neural Networks for steady flow approximation. https://github.com/loliverhennigh/Steady-State-Flow-With-Neural-Nets. accessed 11 July 2018.

[18] Rupesh Kuma, Srivastava Klaus, and Greff Jurgen Schmidhuber. Training very deep networks, 2015.

[19] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. V-net: Fully Convolutional Neural Networks for volumetric medical image segmentation. 2016.

[20] Michael A Nielsen. Neural networks and Deep Learning. http://neuralnetworksanddeeplearning.com/, 2015. accessed 19 July 2018.

[21] C. Nita, L.M. Itu, and C. Suciu. GPU accelerated fluid flow computations using the Lattice Boltzmann Method. *Bulletin of the Transilvania University of Brasov, Series I: Engineering Science*, 6(55), 2013.

[22] Glaucio H. Paulino. Where are we in topology optimization? Orlando, Florida, 2013.

[23] Khairi Reda, Victor Mateevitsi, and Catherine Offord. A human-computer collaborative workflow for the acquisition and analysis of terrestrial insect movement in behavioral field studies. *EURASIP Journal on Image and Video Processing*, 48, 2013.

[24] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional Networks for biomedical image segmentation. *Medical Image Computing and Computer-Assisted Intervention*, 9351:234–241, 2015.

[25] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[26] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[27] Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P. Kingma. PixelCNN++: Improving the PixelCNN with discretized logistic mixture likelihood and other modifications. *International Conference on Learning Representations*, 2017.

[28] S. Schmidt and V. Schulz. A 2589 line topology optimization code written for the graphics card. *Computing and Visualization in Science*, 14(6):249–256, 2011.

[29] SciKit. Scikit-fmm site. https://pythonhosted.org/scikit-fmm, version 0.0.9.

[30] Jean Serra. *Image Analysis and Mathematical Morphology*. Academic Press, Inc, 1983.

[31] J.A. Sethian. A Fast Marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93:1591–1595, 1995.

[32] Wenling Shang et al. Understanding and improving Convolutional Neural Networks via concatenated Rectified Linear Units. volume 48, pages 2217–2225, New York, USA.

[33] Ole Sigmund. A 99 line topology optimization code written in Matlab. *Structural and Multidisciplinary Optimization*, 21:120–127, 2001.

[34] Ayan Sinha et al. SurfNet: Generating 3D shape surfaces using deep residual networks, 2017.

[35] Erva Ulu, Rusheng Zhang, and Levent Burak Kara. A data-driven investigation and estimation of optimal topologies under variable loading configurations. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization*, 4(2):61–72, 2016.

[36] Weiyue Wang et al. Shape inpainting using 3D generative adversarial network and recurrent convolutional networks, 2017.

[37] Sergey Zakharov. 3D object instance recognition and pose estimation in cluttered scenes using CNNs. Master's thesis, Technische Universität München, 2016.