SCHOOL OF COMPUTATION, INFORMATICS AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Thesis in Computational Science and Engineering

Evaluating Convolutional Neural Networks in Multi-Fidelity Modeling

Philipp Kutz

SCHOOL OF COMPUTATION, INFORMATICS AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Thesis in Computational Science and Engineering

Evaluating Convolutional Neural Networks in Multi-Fidelity Modeling

Author:	Philipp Kutz
Supervisor:	Prof. Dr. Felix Dietrich
Advisor:	M.Sc. Vladyslav Fediukov
Submission Date:	30.10.2024

I confirm that this master thesis in computational science and engineering is my own work and I have documented all sources and material used.

Munich, 30.10.2024 Philipp Kutz

Acknowledgments

I would like to thank my supervisor Prof. Dr. Felix Dietrich and my advisor M. Sc. Vladyslav Fediukov for their support throughout my master thesis.

Abstract

The first goal is to investigate how well Convolutional Neural Networks (CNNs) are suited for multi-fidelity (MF) modelling. The second objective is to analyse which architectures and approaches perform better and which perform worse and why there are differences between the various methods. Neural Networks (NN) are capable of learning arbitrary discontinuous functions and can handle high dimensional data better than other regression methods. CNNs are NN, which are well-suited for image processing. MF modelling is an important component in surrogate modelling. There are two main learning styles: NN-based MF models learn the low-fidelity (LF) / high-fidelity (LF) relation either implicit or explicit. Terramechanical data with bi-fidelity tabular data and images were provided by the German Aerospace Center (DLR) for the investigations. Two main architecture types were examined: the Multi-Fidelity Data-Fusion (MF-DF) and the Transfer Learning Neural Network (TLNN) architecture type. The MF-DF architecture type was represented with the explicitly learning MDACNN architecture. The MDACNN architecture processes tabularized data. The TLNN architecture type was represented with the implicitly learning MFCNN-TL architecture and with the implicitly and explicitly learning MF-TLNN architecture. Both the MFCNN-TL and the MF-TLNN architecture process images. It can be concluded from the investigations that all main architectures (MF-DF and TLNN architecture types) and all learning types (explicit, implicit and mixed learning) are suitable for MF modelling. CNNs can be used effectively in MF models. The MF-TLNN architecture with its mixed-learning approach outperformed the other two architectures. This leads to the conclusion that the combination of explicit and implicit learning styles in a network increases learning performance. In future work, more implicit and explicit learning architectures and their combinations in the MF-TLNN architecture need to be investigated to further optimise the performance of the TLNN architecture.

Contents

1. Introduction

Mathematical models have always been developed to solve problems. A big disadvantage of mathematical problems is that they are theoretical and idealised which makes them hard to compute in real-world computers which have to take into account parameters like computational cost. A solution for this problem is the creation of numerical models which are surrogate models created by approximating the operations in the mathematical model. Out of a single mathematical model can be derived, several numerical models which differ in accuracy and computational cost. On the one hand, there are numerical models which approximate better and which have a higher accuracy and computational cost. These models are called High-Fidelity models (HF models). Their results are reliable but due to the high cost, they are less applicable in daily life computations. On the other hand, there are numerical models which approximate worse and which have a lower accuracy and a computational cost. These models are called Low-Fidelity models (LF models). Due to the lower accuracy, the results of these models are less reliable, but the models are very practical for calculations in everyday life due to their low cost. The holy grail in many model design tasks is to create a numerical model which has an accuracy as high as possible and a computational cost as low as possible. To get a model which delivers reliable results and is easily applicable in daily-life computations. Multi-Fidelity modelling (MF modelling) is a method which tries to generate exactly that kind of model. Typical MF models take as input either just the input sample X if they are implicit learning or they take the input sample X and its LF value f_{LF} if they are explicit learning models. Correspondingly, an implicit learning MF model can be described as $f_{MF} : X \rightarrow f_{HF}$ and an explicit learning MF model as $f_{MF} : (X, f_{LF}) \rightarrow f_{HF}$. Independently, whether the MF model is implicit or explicit learning - all MF models need to learn the LF/HF relation - the relationship between the LF and HF function. Therefore, each MF model is trained with datasets of different fidelity values, e.g., an LF and HF dataset. During training, the MF models learn to make the transition from the LF value f_{LF} to the HF value f_{HF} of the input sample X. This approach increases the accuracy of the MF model and keeps the computational cost low compared to a single-fidelity model.
MF modelling can be implemented using different methods. Two of the most popular methods are Gaussian processes (GP) like by Dietrich and Fediukov et al. [1] and neural networks (NN) like by Chen et al. [2]. Both methods have their general advantages and disadvantages but according to [2] is the main advantage of NN against GP that 1) NN are generally better at learning arbitrary discontinuous functions and 2) NN are better at processing high dimensional data than GP. The master thesis aims to create an MF model for a given terramechanical dataset, which is provided by DLR [3][4]. The terramechanical data describes a wheel of a rover moving through the sand. The LF/HF relation in the provided dataset is complex and the features are 12-dimensional - all these are indicators that a NN-

based MF model is better suited for the task than a GP-based MF model. Therefore, the Master thesis aims to create a NN-based MF model. A typical NN-based MF model architecture is the Multi-Fidelity Data-Fusion architecture (MF-DF architecture) type. The original MF-DF architecture is a single perceptron (Figure 2.8) which learned the LF/HF relation implicitly. Later, Guo et al. [5] demonstrated that the original and implicit learning MF-DF architecture is less powerful than multi-level MF-DF architectures, which are composite networks where each sub-network explicitly learns a single relationship. Explicit learning multi-level MF-DF architectures like the 3-level MF-DF architecture by Meng et al [6] (Figure 2.9) and the 2-level MF-DF architecture by Liu et al. [7] and Motamed [8] (Figure 2.10) became state of the art.
Now, it needs to be considered that the original MF-DF architecture and the 3-level and 2-level MF-DF architectures are perceptrons. But the provided terramechanical data has not only a complex LF/HF relation and 12-dimensional samples but the terramechanical data can be described in general as an aggregation of two different datasets, where one dataset contains tabularized data and the other dataset images. Both data sets refer to the wheel moving through the sand; they just provide a different type of data. Both the tabularized data and the image data can be more effectively processed if using a Convolutional Neural Network (CNN) instead of a perceptron. Therefore, the investigations regarding the Master thesis focus on MF modelling using CNNs.
Two main architecture types are used for MF modelling using CNNs: the already known MF-DF architecture type and the Transfer Learning Neural Networks architecture (TLNN architecture) type. In this master's thesis, the MF-DF and the TLNN architecture type are represented by the following networks: the MF-DF architecture type gets represented by the MDACNN architecture by Chen et al. [2] (Figure 3.3) and the TLNN architecture type gets represented by the MFCNN-TL architecture by Liao et al. [9] (Figure 3.6) and the MF-TLNN architecture by Zhang et al. [10] (Figure 3.10). All three networks learn the LF/HF relationship in different ways; the MDACNN architecture learns the LF/HF relation explicitly, the MFCNN-TL architecture learns it implicitly and the MF-TLNN architecture combines both approaches and learns explicitly and implicitly. To summarise, two essential framework conditions were defined for the investigations regarding the Master's thesis: 1) MF modelling is implemented based on the given terramechanical data with the aid of CNNs, and 2) three different learning strategies are implemented, which are used to train MF models. Each learning strategy is represented by its own target architecture: MDACNN, MFCNN-TL, and MF-TLNN architecture. Out of these two frame conditions, two hypotheses can be derived:

1. **The MF models using CNNs will outperform the MF models using perceptrons in terms of the terramechanical data provided.** The reason for this is that CNNs learn, e.g., image processing much more efficiently than perceptrons, as they generalise to local patterns and reuse their weights using moving kernels.

2. **The MF-TLNN architecture will have a higher learning performance than the two other architectures.** The MF-TLNN architecture combines both the explicit and the implicit learning strategy in one network and therefore learns in the theoretically most efficient way.

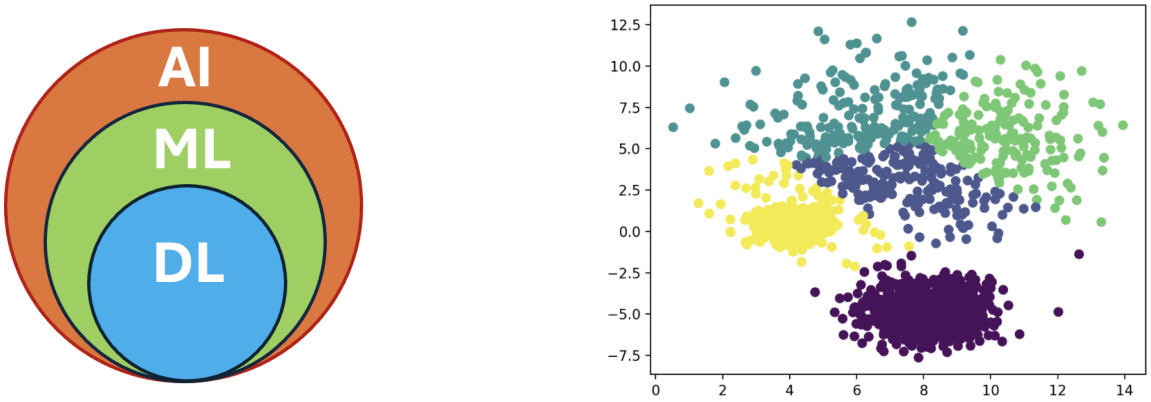
2. State of the Art

2.1. Neural Networks

Neural Networks (NN) are a part of Artificial Intelligence (AI), see Figure 2.1a. But AI includes many more methods than just NNs. This Master Thesis will use NN to process high-dimensional data. The table will be tabularized or images. To understand, why out of all things NNs are used instead of any other AI method to process the data, the AI method environment and the role of NN in the AI environment needs to be understood. AI describes a broad field of ongoing research intending to create methods and techniques which can imitate human intelligence. To understand the area of AI better books like "Artificial Intelligence: A Modern Approach" from Stuart Russel and Peter Norvig [11] are very important. Stuart Russell is a Professor at the University of California, Berkeley and Peter Norvig is the Director of Researcher at Google. This section was written with the knowledge taken from this book. Not all AI is the same. There are multiple levels like shown in Fig. 2.1a, where the most general version of AI is marked with orange colour. Methods which are part of this type of AI are systems which operate entirely with knowledge pre-given to them by the human. The entire knowledge in which these systems operate was given to them initially and the systems do not elaborate new and more knowledge during the process. A typical example for AI is First Order Logic (FOL) with which it is possible to encode knowledge in a logical manner and make it accessible to machines. A typical state in FOL would is shown in Eq. 2.1:

$$\forall x \; King(x) \rightarrow Person(x). \tag{2.1}$$

The x in the FOL is defined as a set of values e.g., here the values can be names. The knowledge encoded by Eq. 2.1 is (1) if x is the king then x is a person, (2) it is not possible that x is the king but not a person, (3) x can be a person but not the king and (4) if x is not a person then x cannot be the king. Methods like FOL are very efficient if the features like *King* or *Person* and their border values for different categorization classes are already known for the given dataset. If the features and their border values are unknown for the data, then are pure AI methods like FOL highly ineffective. For these cases methods of Machine Learning (ML) were developed. ML is a subfield of AI- like visualized in Fig. 2.1a. All methods of ML are AI but not all methods of AI are ML. ML is all about feature extraction, pattern recognition and categorization of data based on the new found features and their border values. It fills the task gap which the pure AI like FOL or robotic agents like lawn mowers are not able to resolve. ML differs from AI by trying to find unknown pattern in the input data and create new knowledge about the dataset which was prior not there. Typical examples are Support Vector Machines (SVM) and most of the unsupervised clustering algorithms like



(a) Sets of AI, ML and DL.

(b) KMeans [12].

Figure 2.1.: Left: relation of AI, ML and DL to each other. Right: KMeans [12], a ML method.

K-Means. K-Means - depicted in Fig. 2.1b - is one of the most commonly used clustering algorithm. It finds clusters of input samples based on the distance of the input samples to each other in the input space. The basic idea is that samples with similar values possesses similar features and belong therefore to the same category. The hyper-parameter k denotes the k nearest input samples around a (virtual) center point in the input space. Each cluster needs k samples to be valid. K-Means is not able to correct itself and reduce its loss because it is not able to change its parameter like hyper-parameter k on its own to find the most optimal cluster distribution. The loss in K-Means can be e.g., the sum of all distances of all cluster points to their common center point. The goal is to make this distance as small as possible and therefore make the clusters as dense as possible like visualized in Fig. 2.1b. A human needs to supervise the algorithm and optimize the hyper-parameters manually to reduce the error. Generally speaking, ML is very effective if using it on low dimensional data for linear and logistic regression. But there are also ML methods which are capable of processing non-linear functions, like Gaussian Processes and SVM using the kernel trick. Otherwise are the most ML methods less effective if it comes to the processing of high dimensional data and for non-linear regression. Typical ML methods like K-Means on their own are not able to use the knowledge from prior feature extractions to find optimal clusters in a new unknown dataset. The downsides of ML equalize Deep Learning (DL). DL is a subfield of ML like depicted in Fig. 2.1a. All methods of DL are part of ML (and AI), but not all methods of ML are part of DL. Typical tasks for ML and DL methods are classification and regression. Default ML methods like SVM and GP, but also DL methods perform well on these tasks for high dimensional data and are capable of non-linear regression. DL is also able to minimise its error by updating its parameters during a training process and can learn new features and how to detect them in the data by generalizing during training. The generalization after training enables an NN to recognize later on the learned features in unknown data which differs DL from ML as well. In other words: DL performs much better than other ML algorithms when it comes to very high-dimensional data with a large number of training data.

The classic examples of this are images or text data. Typical DL methods are NNs like Fully Connected (FC) Networks, Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). This work focuses on the usage of CNNs.

Due to different types of input data like images, texts or speech many different networks and network architectures were developed. Roughly there are two basic types of Neural Networks (NN): the cyclic **recurrent networks** and the acyclic **feedforward networks**[11]. CNNs are acyclic feedforward networks like depicted in Figure 2.2. The Master Thesis focuses on CNNs. Therefore, CNNs will be described further. Convolutions are used in CNNs; the most common application for CNNs is image detection.

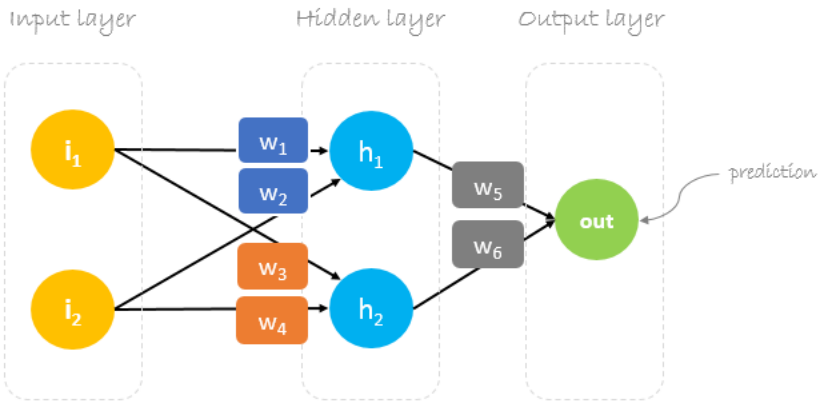


Figure 2.2.: Typical feedforward network [13]. Each node is characterized with trainable parameters.

Almost all NN are build out of neurons. All these neurons of a layer are connected with all neurons of the prior layer. The neurons of the same layer are not connected. Due to the connection among the layer this type of feedforward network is called a Fully Connected network (FC network) with Fully Connected layers (FC layers). CNNs are build out of a block of convolutional layers followed by a block of block of FC layers. The convolutional layers extract features, and the FC layers are used to recognise correlations between the features found. The block of FC layers also outputs the result of the regression or classification task. All neurons in all FC layers possess the same ground structure and do therefore the same basic computations in exact the same order. The goal of each neuron is to either find new feature or find new (non-) linear correlations between already found features. First each neuron does a linear transformation and then the non-linear activation like described in Eq. 2.2. A neuron can be described by the equation:

$$h_{\mathbf{W}}(\mathbf{x_j}) = \sigma(w_0 + \sum_{n=i} w_i * x_{j,i}) \tag{2.2}$$

where the $w_0 + \sum_{n=i}^{w} i * x_{j,i}$ denotes the linear transformation and $\sigma(.)$ the non-linear activation. $h_{\mathbf{w}}(\mathbf{x_j})$ denotes the output (activation) of the neuron h with the weights $\mathbf{w}$ for input $\mathbf{x_j}$. The

weights are important for the neuron to weigh the incoming inputs according to their importance for the feature detection and extraction happening in the node itself. A practical example for the application of Equation (2.2) can be made by using the neuron h_1 from Figure 2.2. Neuron h_1 gets the input vector $\mathbf{x_0} = [I_1, I_2]^T$. For each of the two input values possesses the neuron h_1 a trained weight with which the weighted inputs $w_1 * I_1$ and $w_2 * I_2$ get computed. The weighted inputs get summed together before adding the bias $b = w_0$ to the sum. Until this point, neuron h_1 performed a linear transformation. The result of the linear transformation can be additionally passed through a non-linear activation function σ to add non-linearity. The total equation for neuron h_1 based on Eq. 2.2 can be described by:

$$(h_1)_{\mathbf{W}}(\mathbf{x_0}) = \sigma(w_0 + w_1 * I_1 + w_2 * I_2)$$

The bias $b = w_0$ is an important parameter which prevents the results of the linear transformation from being zero if the weighted sum should be zero. This can happen e.g. if all inputs $\mathbf{x_j}$ into the node h_j are zero. The training process for neuron parameters optimizes only parameters which are directly associated with incoming input values $\mathbf{x_j}$ like the weights $\mathbf{w}$. The bias b itself does not interfere directly with input values, it just gets added to the weighted sum. Therefore, bias $b = w_0$ gets defined as weight by using a dummy input $x_0 = 1$. The dummy input x_0 does not exist actually and possesses a constant value which never changes. In this way, the bias gets trained by the same training procedure that trains the weights. In Equation (2.2) a sigmoid function σ gets used as an activation function. Following non-linear activation functions are the most common to be used in NN:

- The sigmoid function is popular for logistic regression due to its values $y \in [0, 1]$

$$\sigma(x) = 1/(1 + e^{-1})$$

- The rectified linear unit (ReLU) leads in many cases to faster convergence during the training than other activation functions.

$$ReLU(x) = max(0, 1)$$

- The softplus function is a variant of the ReLU function with a smoother transition from $y = 0$ to $y = x$ at $x = 0$. The derivative of the softplus is the sigmoid function.

$$softplus(x) = log(1 + e^x)$$

- The tanh function is a shifted and scaled variant of the sigmoid function with $y \in [-1, 1]$

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

The ability of an NN to perform non-linear regression does not only depend on whether the NN possesses neurons with non-linear activation functions. The **Universal Approximation Theorem**[11] defines the minimum requirement which each NN needs to fulfil to be able to

learn arbitrary functions. According to the theorem the smallest possible NN to do this needs only two (propagating) layers - a hidden layer and an output layer - where the first layer is non-linear and the second layer is linear. To describe this using an example: According to the Universal Approximation Theorem, the network shown in Figure 2.2 is the smallest possible NN for learning arbitrary functions if the nodes h1 and h2 have non-linear activation functions and the node *out* in the output layer a linear activation function like, e.g., $y = x$.

In most cases the input gets forwarded through many neurons and layers of neurons. Therefore, the Equation (2.2) needs to be put into context of the propagation serially through many layers. The Equation (2.3) describes a full propagation of the input $\mathbf{x}$ through the network deriving the predicted output $h_{\mathbf{W}}(\mathbf{x})$. All neurons of the network execute the operations described by Equation (2.2) during the propagation:

$$h_{\mathbf{W}}(\mathbf{x}) = \mathbf{g}^{(2)}(\mathbf{W}^{(2)}\mathbf{g}^{(1)}(\mathbf{W}^{(1)}\mathbf{x})). \tag{2.3}$$

Eq. 2.3 describes the forward propagation through a two-layer network with an input layer ($\mathbf{x}$), hidden layer (parameters $\mathbf{W}^{(1)}$ and $\mathbf{g}^{(1)}$) and an output layer (parameters $\mathbf{W}^{(2)}$ and $\mathbf{g}^{(2)}$). The two-layer network could, e.g., the two-layer NN visualised in 2.2. Each weighted sum in $\mathbf{W}^{(1)}\mathbf{x}$ gets passed through its corresponding activation function in $\mathbf{g}^{(1)}$. The result is a vector of (non-)linear activation's $\mathbf{g}^{(1)}(\mathbf{W}^{(1)}\mathbf{x})$. The second layer does the same as the first layer: it computes with the weight matrix $\mathbf{W}^{(2)}$ the weighted sum for each neuron and computes then with the activation vector $\mathbf{g}^{(2)}$ the corresponding (non-)linear activations. Referring to the example network in Figure 2.2, there the output layer possesses only one neuron. In this case, the output $h_{\mathbf{W}}(\mathbf{x}) = \hat{y}$ is the prediction of the network. Using Equation (2.3) it is possible to describe the complete forward propagation through the network in Figure 2.2 by the following equations:

$$\begin{aligned}
\hat{y} &= g^{(out)}(in_{out}) && (2.4)\\
&= g^{(out)}(w_{0,out} + w_5 * h_1 + w_6 * h_2) && (2.5)\\
&= g^{(out)}(w_{0,out} + w_5 * (g^{(h_1)})(w_{0,h_1} + w_1 * i_1 + w_2 * i_2) && (2.6)\\
&\quad + w_6 * (g^{(h_2)})(w_{0,h_2} + w_3 * i_1 + w_4 * i_2)) && (2.7)
\end{aligned}$$

Where in_{out} is the weighted sum of the output node which gets computed out of its bias $w_{0,out}$ and weighted inputs $w_5 * h_1 + w_6 * h_2$. $g^{(out)}$ is the (non-)linear activation function of the output neuron though which the weighted sum in_{out} gets passed.

Like already proved, the weight parameter have together with the activation functions a key role during the forward propagation. The combinations of all weights over all layers expresses the function learned by the NN. The weights of the NN are initialized arbitrarily and we optimize them to learn a target function y, i.e. learn the weights. To optimize the weight parameters NNs gets trained using a special training set which contains samples $\mathbf{x}$ and corresponding ground truths $\mathbf{y}$. Ground truths $\mathbf{y}$ are the correct class or desired regression value for a certain input sample x and gets used to evaluate the model prediction $\hat{y}$ which can deviate from the ground truth especially before and and during the training. The most

common method to update and optimize the weight parameters during the training process is called Gradient Descent:

$$w \leftarrow w - \alpha \nabla_{\mathbf{W}} L(\mathbf{w}) \tag{2.8}$$

The gradient in Gradient Descent is $\nabla_{\mathbf{W}} L(\mathbf{w})$ and can be rewritten as $\delta L(\mathbf{w})/\delta \mathbf{w}$, which describes a derivative of the loss function $L(\mathbf{w})$ w.r.t. a weight parameter. Backpropagation is the process, where the NN gets optimized by updating its weights. Backpropagation starts at the output of the output neuron with the Loss function $Loss(h_{\mathbf{w}})$ and computes from the back to the front the gradients of all weight parameters in the system. The Loss function defines the prediction error between the prediction y and the ground truth $\hat{y}$, which the NN had during a training cycle. There are different methods to compute the loss (prediction error) during the training. All losses of a training cycle computed using the same method and put together create the loss function $Loss(h_{\mathbf{w}})$. One example would be the L2-Loss:

$$Loss(h_{\mathbf{w}}) = L_2(y, h_{\mathbf{w}}(\mathbf{x})) = ||y - h_{\mathbf{w}}(\mathbf{x})||^2 = (y - \hat{y})^2. \tag{2.9}$$

A different approach to compute the loss is to use the negative log-likelihood as loss function:

$$Loss(h_{\mathbf{w}}) = -logP(D|W) \tag{2.10}$$

Negative log-likelihood describes, how well an NN defined by the weights W can can define the observed data D. The decrease of the negative log-likelihood via optimizing the weights to maximize the likelihood is called Maximum Likelihood Estimate (MLE) and is a popular method to derive parameters for NN.
The forward propagation starts at the input layer and ends at the output layer and computes each prediction $\hat{y}$ for and input x. The backpropagation starts at the output layer and ends at the input layer, takes as input the loss function $L(\mathbf{w})$ and computes on its way through the network all derivatives $\nabla_{\mathbf{W}} L(\mathbf{w})$ of the loss function according to all weights in the network.

$$\frac{\delta g(f(x))}{\delta x} = \frac{\delta g(f(x))}{\delta f(x)} * \frac{\delta f(x)}{\delta x} \tag{2.11}$$

Backpropagation gets implemented using the chain rule. The chain rule makes the derivations throughout the whole network easier because it enables the possibility to reuse already computed derivations over and over again as shown in Equation (2.11) which saves computational power. An application of the backpropagation on the feedforward network described in Fig. 2.2 for the weight w_5 would be:

$$\frac{\delta}{\delta w_5} L(h_{\mathbf{W}}) = \frac{\delta}{\delta w_5}(y - \hat{y})^2 = -2 * (y - \hat{y}) \frac{\delta \hat{y}}{\delta w_5} \tag{2.12}$$

$$= -2(y - \hat{y}) \frac{\delta}{\delta w_5} g_{out}(in_{out}) = -2(y - \hat{y}) g'_{out}(in_{out}) \frac{\delta}{\delta w_5} in_{out} \tag{2.13}$$

$$= -2(y - \hat{y}) g'_{out}(in_{out}) \frac{\delta}{\delta w_5}(w_{0,out} + w_5 h_1 + w_6 h_2) \tag{2.14}$$

$$= -2(y - \hat{y}) g'_{out}(in_{out}) h_1. \tag{2.15}$$

$L(h_{\mathbf{W}})$ describes the Loss function based on the weights $\mathbf{w}$ of the network $h_{\mathbf{W}}$. In the backpropagation above the Loss function gets computed using the L2-Loss $(y - \hat{y})^2$ where for each input sample x the ground truth is y and the prediction is $\hat{y}$. $w_{0,out}$ is the bias of the output node and h_1 and h_2 are the activations of the corresponding neurons which get weighted with the weights w_5 and w_6 by the out neuron. The weighted sum of the out neuron gets denoted by in_{out}.

Backpropagation computes the gradients for all parameters like $\delta L(\mathbf{w})/\delta \mathbf{w}$ and functions like $\delta L(\mathbf{w})/\delta \mathbf{a}$ inside the NN. Parameter w_5 from the Figure 2.2 updated by subtracting the gradient $\frac{\delta}{\delta w_5} L(h_{\mathbf{W}})$ retrieved in the Equation (2.16).

$$w_5 \leftarrow w_5 - \alpha \frac{\delta}{\delta w_5} L(h_{\mathbf{W}}). \tag{2.16}$$

It is very expensive to fit the entire dataset, as they could be extremely large. However, the gradient could be seen as an expectation, and the expectation can be approximated using smaller data samples. Therefore, we randomly (stochastically) split the original dataset into smaller mini-batches and update the weights using backpropagation on this smaller set. We iteratively update the weights based on what we have learned from each minibatch. When we went through the entire dataset, its called an epoch. We repeat this iterative process until we converge to the optimal weight values. There are different approaches to determine the size of mini-batches. The goal is to reduce outliners and define a clear trend that stabilises the convergence. Another problem during training are vanishing gradients. Vanishing gradients are a problem during training because they block the optimisation process of the NN parameters during training. The network does not update its parameters, the weights remain equal, and the predictions do not improve over training iterations. Vanishing gradients describe the phenomenon that the network makes false predictions (Loss function is not zero) but the gradients $\delta L(\mathbf{w})/\delta \mathbf{w}$ are close to zero or zero due to rounding errors or Underflow and do not update the weight parameters using Gradient Descent like described in Eq. 2.8. An effective and commonly used approach to counter the degradation of information in the forward propagation and vanishing gradients in the backpropagation are residual connections. These residual connections connect inside the network layers (and their activations) with layers behind those and transport the activation over several layers unprocessed over several layers back into the network. By cutting out some processing steps and adding the activations later on again into the main information flow residual connections fight information loss. Another method to counter vanishing gradients is to use Batchnormalization. Batchnormalization gets described as follows:

$$\hat{a}_i = \gamma \frac{a_i - \mu}{\sqrt{\epsilon - \sigma^2}} + \beta. \tag{2.17}$$

a_i is the activation of a neuron $h_{\mathbf{w}}$ for the i-th input sample out of x_1, ... x_N samples in the batch and $\hat{a}_i$ is the normalized activation passing the Batchnormalization layer. Mean μ and the variance σ^2 get computed by forward propagating all batch samples through the layer, storing their corresponding unnormalized activations and compute out of the resulting

set of activations **a** the mean μ and the variance σ^2. The term ϵ prevents the denumerator from being zero if the variance σ^2 should be zero. γ and β are two learnable parameters to optimize the Batchnormalization.
The most important feature of an NN is its ability to generalize. Generalization is the ability of a NN two learn knowledge during its training and to apply the learned knowledge later on unknown data correctly. In other words: the network learns the patterns in the train dataset and not the train dataset value-by-value. There are three different methods to make sure that the NN generalizes as well as possible during the training: **network architecture search**, **dropout**, **early stopping** etc. The network architecture is important for the generalization because the architecture must fit the input data type, e.g., speech or text needs a different network type (recurrent neural networks) than images (feedforward network including convolutional layers). Then there is the empirical findings that the deeper the network, the better does the network generalize - independently from the amount of parameters which the network possess. The more layers, the better the network generalizes. Normally the optimal network architecture for a problem gets found with experience and good evaluation. But there are also algorithms which get used to construct an network architecture which provides the most optimal generalization like the Evolution Algorithm. The Evolution algorithm possesses a heuristic leading it to minimize the loss function and maximize the generalization by search for the optimal architecture. The algorithm merges two already created architectures together and creates this way a new architecture. Also new mutations can be added to the networks. Mutations are stochastically generated features which get implemented into the network. Another way to improve the generalization is weight decay via applying regularization:

$$L(h_{\mathbf{W}})_{regularized} = L(h_{\mathbf{W}}) + \lambda \sum_{i,j} W_{i,j}^2. \tag{2.18}$$

$L(h_{\mathbf{W}})$ is the Loss function of network $h_{\mathbf{W}}$ computed with a chosen loss metric. Penalty $\lambda \sum_{i,j} W_{i,j}^2$ penalizes big weights by increasing the loss function. Factor λ weights the penalty which gets added to the Loss function $L(h_{\mathbf{W}})$. This leads to bigger gradients during backpropagation and bigger update steps of the weight parameters during the Gradient Descent. The goal is to minimize the weight parameters because big weight parameters lead to numerical instability, overfitting and disturbed convergence. The last method is dropout. Dropout possesses as probability p which defines for each neuron how big its probability for each batch is to be activated. An activated gets used during the forwardpropagation and backpropagation of batch samples and deactivated neuron does not get used. E.g., if $p = 70\%$ than the probability for each neuron is 70% that the neuron will be activated. A possible explanation for the effectiveness of dropout is, that dropout forces the individual neurons to find the most effective feature to look for in the input data and to learn the most effective way to pass the found feature to many different other neurons in the following layer in a form in which the following neurons can use the found feature.

2.2. Convolution Neural Networks (CNNs)

There are different tasks which are connected to the processing of images using Neural Networks (NN). One of the most common ones is to classify the shown object on the image. There are different approaches to how the classification can be realized.
On the one hand, a feasible approach would be to use a Neural Network which is built out of Fully Connected Layers - a Fully Connected Network (FCN). The original image is organized as a 2D to 3D matrix. An FCN cannot handle data organised in matrix-like shapes. Therefore, the image gets transformed into a 1D vector by flattening before it gets processed by the FCN. Flattening converts the image (matrix) into a 1D vector by serially concatenating all rows back after back. One of the biggest downsides of this approach is, that there will be a huge amount of training data necessary to train such an NN due to the large amount of parameters the NN will have. The problem with the sheer amount of parameters is illustrated later on in this section.
On the other hand, a Convolutional Neural Network (CNN) could perform the classification task just as well. Advantages of the CNN before an FCN are:

1. CNN reuse their parameters and need therefore less training data. The concept is known as shared weights and will be explained later on. Due to shared weights, CNNs are more invariant to input data transformation than FCNs are. Data transformations are, e.g., distortion, translation, rotation, scaling, wrapping, etc.

2. The CNN uses an important key feature of images: directly adjacent pixels are more correlated to each other than distant pixels. Unlike the FCN are the CNNs able to use the knowledge of correlating pixels for their classification.

The CNN enables these advantages over the FCN by incorporating the following features into its architecture: (i) **local receptive fields**, (ii) **weight sharing** (iii) (down-) **sampling**.

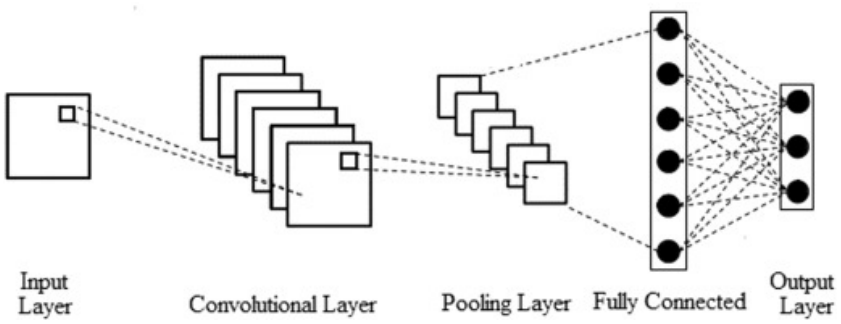


Figure 2.3.: Typical structure of an CNN [14]. The Convolutional Layer tries to find local features using kernels to search in image patches. The Pooling Layer downsamples the convolution results. This makes the CNN more invariant regarding data transformations like translation or distortion. The Fully Connected Layer section at the end finds the correlation among the found features - merges the local features together to global features - and classifies the image.

All these three features get visualized in Fig. 2.3. In the first step the Fig. 2.3 shows the Input Layer. The input layers contains the image which depicts the objects which must be classified. The input image in consists out of only one plane which carries all pixels. In this case - by default - does the image depict its content in different scales of gray. The single plane represents the distribution of the feature brightness over the area of the image, therefore the plane is called a **feature map**. The next layer in Fig. 2.3 is a Convolutional Layer. An important key feature of the CNN is to be **spatial invariant**. The CNN should be able to detect the same feature independently where the feature is located on the image. To search after a certain feature in the whole image feature detection must be applied to different locations of the image. For that purpose, the image gets divided into smaller sub-pieces - so-called patches. The goal is to check in each patch individually whether the feature is contained in the corresponding area of the image or not. All patches possess the same size, and patches can overlap each other - the patch size itself and whether the patches overlap or not depend on the parameters of the used feature detectors. The feature detectors are also called **kernels**. Each feature possesses for detection its own kernel. The feature detection using a kernel outputs a feature map showing the overall distribution of the feature over the input image. In practice, a default Convolutional Layer possesses several kernels to create an efficient Neural Network (NN). In Fig. 2.3 possesses the output of the Convolutional Layer several feature maps - the input image was investigated after several features (each feature got a kernel) and each feature detection produced one output feature map. All kernels of the same Convolutional Layer possess the same hyper-parameters like a fixed defined size and stride. The stride is the step size of a kernel between two feature detection operations. The size of the kernel equals the size of the patches in the input image. The Convolutional Layer applies between each patch and the kernel a convolution explained in Eq. 2.19. Therefore, each patch of the image marks a local receptive field. Each kernel is characterized by adjustable weight parameters, a bias value and an associated (non-linear) activation function. The convolution computed by a kernel is more a cross-correlation - it gets checked how similar is the kernel compared to the currently checked image patch. A convolutional operation can be expressed by

$$z = \sigma((\sum_{n=1}^{N} w_i * x_i) + b). \tag{2.19}$$

First all kernel weights w get element-wise multiplied with the corresponding image patch values x. N describes the number of values which the same-sized kernel and image patch contain. All $w_i * x_i$ products get summed together. Then the scalar bias value b gets added to the sum $\sum$. In the last step an activation function σ gets applied to the resulting scalar value - the activation z is the output of the (non-linear) activation function σ and the result of the convolution. A typical activation function could be e.g., Rectified Linear Unit (ReLU).
Short summary for the convolutional layer: the convolutional layer must be spatial invariant. Therefore, the input image gets split up into patches to provide local feature detection using a kernel at different locations of the image. The kernels use the key feature of correlation among adjacent pixels to find features in the input image. Each patch area in the image corresponds to a local receptive field. A single kernel gets reused over and over again

for different patches to detect the same feature at different locations. Therefore, the CNN possesses fewer parameters than a comparable FNC and can be trained faster with less data than the FCN.
A simple example to illustrate the advantage of CNNs over FCNs: the total amount of parameters for a Fully Connected layer (FC layer) z_{FC} gets computed by:

$$z_{FC} = w * n + b$$

where w is the number of input values, n is the number of neurons which the FC layers possess and b describes the amount of bias values in the FC layer. Each neuron in an FC layer possesses by default one bias value. Therefore, is $n = b$. The total amount of parameters for a kernel in a Convolutional Layer z_{CONV} is:

$$z_{CONV} = h * w + b$$

where h describes the height, and w the width of the kernel and b describes the scalar bias value of the kernel. A kernel possesses only one bias value b - therefore, is $b = 1$.
Lets consider for the example only the first layer of an image processing NN. Case 1: If the image has n pixels and the first layer is a FC layer with the same amount of neurons as there are image pixels, then the total amount of parameters for the one layer is $n^2 + n \approx n^2$. For a typical megapixel RGB image this would be 9 Trillion weights. Case 2: If the first layer is not a FC layer like in Case 1 but a Convolutional Layer like depicted in Fig. 2.3 with a kernel of shape rxc where $r << n$ is the amount of rows and $c << n$ is the amount of columns a kernel covers. If the Convolutional Layer possesses a single kernel and $d = r * c + 1$ is the total amount of parameters for a single kernel, than $d << n^2$ is true. Even if the Convolutional Layer has $l << n$ different kernels, than the total amount of parameters is $d * l << n^2$.
No matter how many kernels the Convolutional Layers possesses, under the given circumstances the Convolutional Layer possesses with one kernel $d << n^2$ and with several kernels $d * l << n^2$ significantly less parameters than a comparable FC Layer (n^2 parameters). Another shown advantage is, that with $d \perp n$ and $(d * l) \perp n$ the number of parameters of the Convolutional Layer unlike the FC Layer is independent of the size of the input image.

A big problem for feature detection using Convolutional Layers are data transformations of the input data. E.g., translations, distortions and other shifts. These data transformations make the feature detection less unambiguous. But the more unambiguous the results of feature detection are, the more effective the NN. It is common to pair a Convolutional Layer with a subsequent Pooling Layer visualised in Figure 2.3 to reduce the effects of small shifts in the input data, to make the feature maps of the Convolutional Layer less ambiguous and therefore make the NN generally more effective. The effect of a Convolution-Pooling pair on a convolution result is illustrated in Figure 2.4. The Pooling Layer achieves this result by downsampling the spatial resolution of the feature maps. The Pooling Layer removes units in the feature maps but does not removes feature maps. There are as many feature maps after pooling as there where before pooling. The Pooling Layer defines a sliding pooling window. The pooling window is like a kernel in the Convolutional Layer which is moving over the input and operates at different locations of the feature maps. But unlike the

kernels, the pooling window does not possess any parameters. A typical Pooling Layer is a pooling window of size $2x2$ which covers a feature map patch of $2x2$. Due to downsampling, commonly, the patches created by the Pooling Layer do not overlap each other. If the pooling window has a size of $2x2$, then the window possesses a stride $s = 2$. This way the Pooling Layer has halved the number of units in the feature maps. Each patch (feature map) consists out of $2 * 2 = 4$ values and gets reduced by the pooling window to one scalar value. The computation of the scalar pooling results differs, depending whether using Average Pooling or Max Pooling. Average Pooling (bottom right in Fig. 2.4) computes out of all patch values in the current pooling window the average value. In Max Pooling (top right in Fig. 2.4) the pooling window does not compute the average but extracts the biggest value (maximum) out of the current patch.

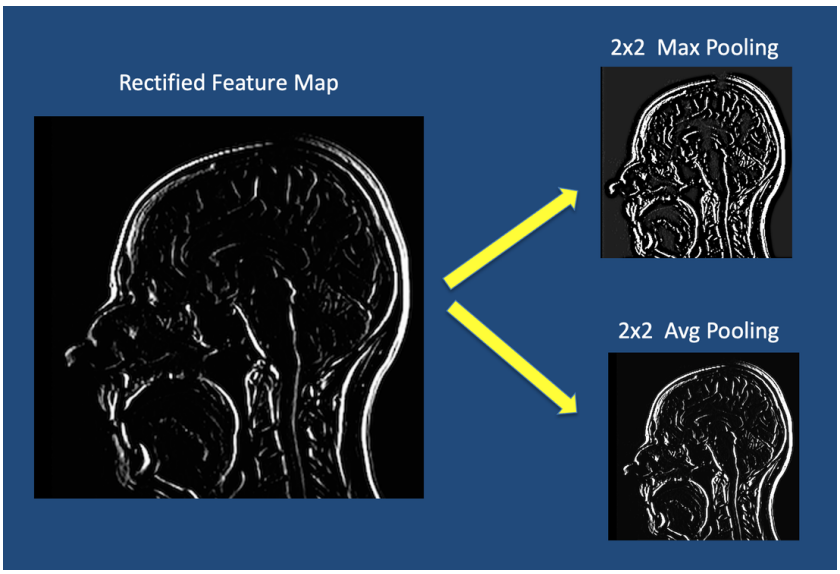


Figure 2.4.: The left image the original image before pooling [15]. On the right side is the image after pooling. Both - average and max pooling - increase the contrast of the image visibly and highlight this way the edge routing even more than the original image does. This way the result of the Convolutional Layer gets less unambiguous and the NN gets more efficient.

Fig. 2.3 shows an example CNN which possesses only one Convolutional and Pooling Layer paired together (Convolution-Pooling pair). Most CNNs are build out of several Convolution-Pooling pairs which are stacked up together in a sequential manner. This way the CNN is able to search for even more complex features in the input image. The deeper the Convolution-Pooling pair lies in the CNN, the more invariant is it to data transformations of the original input image due to the prior poolings (down samplings). The more Convolution-Pooling pairs a NN possesses and the deeper inside the NN they are, the smaller but more reliable and less unambiguous are the feature maps due to down sampling. To counter the information loss through the reduction of spatial resolution over the propagation course

through all Convolution-Pooling pairs, it is common to search after more and more features (use more kernels per Convolutional Layer) the deeper inside the NN. This leads to the effect, that the deeper the layer, the smaller the feature maps but the more feature maps are contained in the output data of layers.
All CNNs can be generally split into two parts: first a feature extraction part followed by a classification part. The example CNN in Fig. 2.3 depicts this architecture as well. In the feature extraction part all features of the input image get identified, extracted and mapped in the feature maps. For this task CNNs use Convolutional and Pooling Layers paired together. The classification part consists of the FC layers and the output layer. The Convolutional Layer is good in feature extraction regarding images. The FC layer is good in finding correlations between the features found by the Convolutional Layer. The output layer - using a softmax function - computes out of the found correlations for each possible class the probability that the input image belongs to the class. The output layer computes all probabilities and outputs than only the class with the highest probability as the classification result.
The training of a CNN can be done based on error minimisation. The error could be described by the loss function. The optimisation of the network parameters can be done via backpropagation - although for CNNs there must be done smaller adjustments to the backpropagation (compared to the backpropagation for FC layers) due to the used shared weights.

2.3. Multi-Fidelity Modelling

Multi-Fidelity Modelling is part of current research and its goal is to make numerical models more efficient by decreasing their computational cost and increasing their speed while keeping their accuracy high. The paper "Survey of Multifidelity Methods in Uncertainty Propagation, Inference, and Optimization" [16] from 2018 investigated over 200 publications regarding Multi Fidelity Modelling and defined the Multi Fidelity Modelling as a concept, defined the tasks and the corresponding areas of applications where Multi Fidelity Modelling is better suited than using other methods (see Fig. 2.6), defined the typical Multi Fidelity Model architectures which get used and defined the typical methods how low fidelity models can be derived as approximations of the corresponding original high fidelity models. The highly cited publication [16] will therefore be used as default source if not marked otherwise.
Modern Computer Science, Engineering, Physics, etc. would not be working without mathematical and numerical models. Models describe systems of interest by learning the correlation (relationship) between the system's input and output.
Each numerical model possesses three main features: accuracy, computational cost and speed. The accuracy denotes how well the model learned the mapping between the input and output. The more and bigger errors the model makes, the lower its accuracy and the lower its fidelity. Models which possess a high accuracy are called high fidelity models $f_{HF} : X \to Y_{HF}$ and models which possess a low accuracy are called low fidelity models $f_{LF} : X \to Y_{LF}$. Computational cost describes the amount of needed computational operations and therefore, the amount of computational resources aka hardware which is needed to process the model.

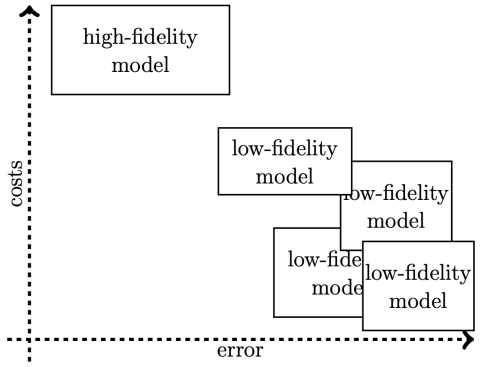
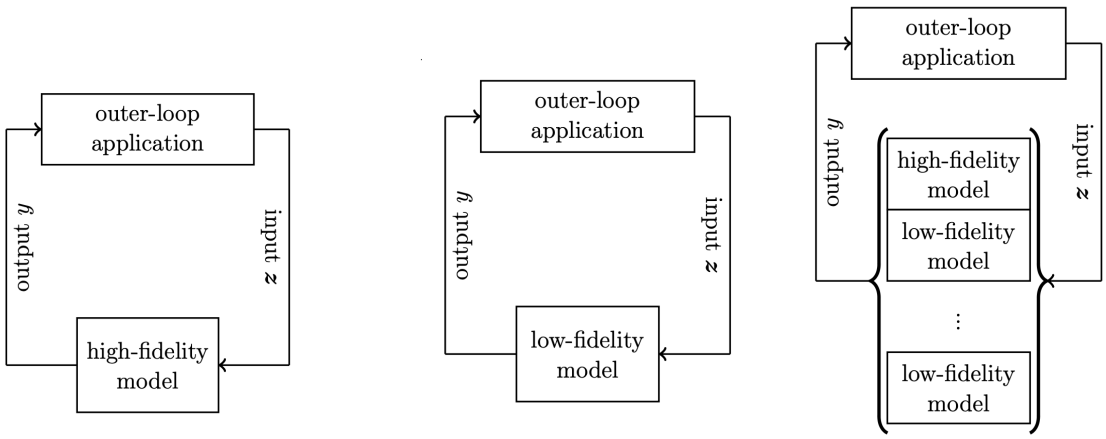


Figure 2.5.: The lower the error (the higher the accuracy) the higher the computational cost. The higher the error (lower the accuracy) the lower the computational cost [16].

Computational resources are finite and the common goal for processing models is to process models with as few computational resources as possible as fast as possible. Therefore, it is generally viewed as an important practical advantage if numerical models are computationally cheap. The speed describes how fast the model computes the output for an input and it is linked to the computational cost of the computer and the faster the better. Fig. 2.5 shows the correlation between the accuracy, computational cost and speed using a high fidelity model f_{HF} and several low fidelity models f_{LF}. Usually, one high-fidelity model f_{HF} comes with several lower-fidelity models f_{LF}, which all try to approximate the higher one with lower accuracy. Generally speaking, according to Fig. 2.5, the higher the accuracy the higher the computational cost and the lower the accuracy the lower the computational cost.

Due to the correlations between accuracy, computational cost and speed and the possibility of defining approximation of very expensive models, there are three different basic model systems which can be implemented as shown in Fig. 2.6. All model systems in Fig. 2.6 are built the same: the model is embedded into an outer-loop application where the model and its propagation of the input is the inner-loop and the outer-loop is an application which uses the knowledge about in- and output into the system to perform a task - like the optimization of a design. The first model in Fig. 2.6a shows the approach using only the high fidelity model as single fidelity in the system. An advantage over the other model types is that the accuracy is always as high as desired. The drawbacks, illustrated in Fig. 2.5 typically involve high computational costs and slower processing speeds, which result from the increased complexity of the model. A typical example would be e.g., Newton's method which is used for parameter updating during the training of neural networks. Newton's Method is a computationally more expensive alternative to the commonly used Gradient Descent and the computational cost in Newton's Method comes from the necessity to compute the Hessian matrix (second derivation). There are two other models in Fig. 2.6 which try to counter the disadvantage of the high computational cost of the high fidelity model system in Fig. 2.6a. The second model system in Fig. 2.6b shows a low-fidelity model embedded into an outer-



(a) Single-Fidelity model utilizing a High-Fidelity model [16].

(b) Single-Fidelity model utilizing a Low-Fidelity model [16].

(c) A Multi-Fidelity model [16].

Figure 2.6.: Left: the high fidelity model is accurate but computationally expensive and slow. Middle: the low fidelity model is less accurate but computationally cheaper and faster. Right: the multi fidelity model uses the high accuracy model to guarantee convergence and define the desired accuracy, the low fidelity models enable the computational efficient propagation through the model

loop application. The low fidelity model regarding Newton's Method would be e.g., a version of Newton's Method where the Hessian matrix (the most expensive procedure) does not get computed explicitly but gets approximated which lowers significantly the computations cost and improves the speed but lowers the accuracy. A big disadvantage of such a system is the absence of the original high-fidelity model. With the exchange of the high fidelity model with a low fidelity model the model system looses also all information about the highest desired accuracy - the accuracy of the high fidelity model - and there can be convergence problems because in contrast to high fidelity models do low fidelity models not guarantee convergence. The multi-fidelity model in Fig. 2.6c tries to combine the advantages of the high fidelity and the low fidelity model by creating a model which has a high accuracy and does guarantee converges but has also a lower computational cost and is faster than the original high fidelity model. The multi-fidelity model is built out of multiple single-fidelity models. The high-fidelity model is kept to keep the knowledge about the highest desired accuracy and to guarantee convergence. The low-fidelity models are kept for fast computations. A very important aspect of multi-fidelity models is to combine the different models effectively with each other. Depending on the executed task (described as an outer-loop application in Fig. 2.6) there where developed different model management strategies to enable multi-fidelity modelling based on the task.

There are three basic applications for which the model management strategies can be used: **Uncertainty Propagation**, **Statistical Inference** and **Optimization**.

The goal of the common Uncertainty Propagation is to compute the uncertainty distribution for a random input vector x which gets forwarded into a model. The uncertainty distribution shows the error of the model which occurs because the I/O correlation function learned by the model does not correspond perfectly to the I/O correlation of the system of interest. Due to falsely learned correlations the model is able to predict for the same input x different outputs y. The uncertainty distribution displays these output values y together with their occurrence probability. The errors are all predictions y which divert from the ground truth $\hat{y}$ of the input x. Gaussian Process Regression and Kriging compute similar uncertainty distributions for given datasets. Normally the uncertainty distribution gets computed by propagating the model many times through the given model. The cost for the computation is $n * cc_{model}$ where n is the number of propagations of input x and cc_{model} describes the computational cost of the model for one propagation. For uncertainty propagation using a high fidelity model the cost can get very high very fast with a low amount of propagations. But a high number of propagations is very important to build an expressing uncertainty distribution, therefore saving computational cost by reducing the number of propagations of x through the model is not always a preferred choice for cost reduction - at this point comes multi-fidelity modelling into play. The computational cost gets lowered by exchanging the high-fidelity model with a low-fidelity model which approximates the uncertainty distribution computationally cheaply. The low-fidelity model is an approximation of the original high-fidelity model with lower accuracy and computational cost. This Master Thesis focuses on CNNs applied to multi-fidelity modelling. CNNs are neural networks, and due to their fixed defined weights, neural networks do not possess any uncertainty distribution in their output. As long as the model uses the same set of weights the output y keeps the same for the same input x over different iterations. Therefore, the commonly used method described above cannot be applied to CNNs. [17] investigates how uncertainty propagation regarding neural networks can still be done. An answer is to not compute the uncertainty distribution for one input x but for the whole input space $\mathbf{X}$, measuring the error occurring over the input space and describing it with first and second order statistics (mean and variance). In the method for computing uncertainty described above the number of samples remains the same and the model gets approximated by a low fidelity model. [17] suggests doing exactly the opposite for neural network models, keeping the model (no approximation) and reducing the amount of actually propagated samples. As one of the simplest methods for reducing the number of samples [17] suggests to use e.g., Markov Chain Simulation where a subset $\mathbf{X_{sub}} \subseteq \mathbf{X}$ uses the law of big numbers to approximate with a smaller subset $\mathbf{X_{sub}}$ of randomly sampled inputs x from the original state space $\mathbf{x}$ the neural networks uncertainty distribution aka. the error of the learned I/O correlation function of the neural network.
Another application for multi fidelity modelling is e.g., Statistical Inference. A typical application regarding of Statistical Inference [18][19][20][21] is solving the Inverse Problem where are given a model and the output of the model and the goal is to find the input which caused the output of the system. The search after the input in the Inverse Problem is equal to finding the input with the highest posterior:

$$p(Y|X) = p(X|Y) * p(Y).$$

The posterior $p(Y|X)$ defines the probability that the model outputs for input X the output Y. The prior $p(Y)$ defines the already known output Y - the output is pre-known in the Inverse Problem. The likelihood $p(X|Y)$ compares the known output distribution Y with the output distribution X obtained by propagating input X through the model. The likelihood is a cross entropy and gets bigger the bigger the correlation between the two distributions is. The bigger the likelihood $p(X|Y)$, the bigger the posterior $p(Y|X)$, the bigger the probability that the corresponding input sample is the solution of the inverse problem. Statistical Inference can get very fast computationally very expensive if computing for each input sample in the input space the output distribution using the high fidelity model. There are two basic ways how the computational cost can be reduced. On the one hand it is possible to use Monte Carlo Sampling and sample from the input space randomly a subset of inputs which gets checked. On the other hand multi fidelity modelling can be used where it is possible to pre-filter input samples using computationally cheap low fidelity models to approximate their output distribution and to pass input samples only to the computationally expensive high fidelity model if the low fidelity model gives the input sample a pass. An input sample gets a pass if it gets accepted by the low fidelity model and the approximated output distribution lies within a desired interval. A common method to pre-check input samples using pre-conditioned Markov Chain Monte Carlo (MCMC) [22][23][24][25]. Pre-conditioned MCMC combines Monte Carlo Sampling with multi fidelity modelling. First Monte Carlo samples an input X. Then the Markov Chain uses in the first step a low fidelity model to evaluate the input sample. If the input sample gets a pass, the input sample gets forwarded in to the actual high fidelity model in the next step where the actual posterior $p(Y|X)$ gets defined. Choose the input sample X as output for the Inverse Problem which gets the highest posterior.
The last of common application types for multi fidelity modelling is optimization. Optimization gets typically done in three different ways. Instead of using the high fidelity model the search after the optimal parameters can be accelerated by using low fidelity models as well [26][27][28][29]. Another way is to use low fidelity models together with adaptive correction to minimize their error and increase the optimization process [30][31][26][32][33]. The third and last approach would be define a completely new surrogate model which takes over the optimization task. Typically surrogate models can be realized using a neural network which gets trained with data from models of different fidelities e.g., via transfer learning. The surrogate approach is interesting for this Master Thesis because the Master Thesis investigates the usage of CNNs in multi fidelity modelling.

Multi fidelity modelling is about combining models of several different fidelities with each other. [16] identified three basic methods - so called model management strategies - how models of different fidelity can be effectively combined together: **Adaptation**, **Fusion**, **Filtering**. All these basic methods get displayed in Fig. 2.7 together with the tasks for which they mainly get used. The first model management strategy represented in Fig. 2.7 is Adaptation. Using [30] as an example Adaption can be used to optimze the design of an airfoil (wing shape). The goal is to optimize the wing shape with as less computational cost as possible. The multi model in [30] consists out of a high fidelity and an low fidelity model therefore, to reduce the computational cost of the multi fidelity model it is necessary to relocate the

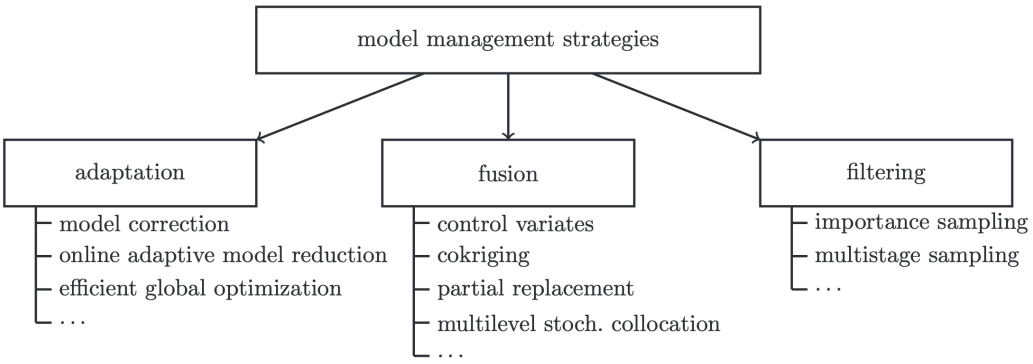


Figure 2.7.: The lower the error (the higher the accuracy) the higher the computational cost. The higher the error (lower the accuracy) the lower the computational cost [16].

iterative optimization computation from high fidelity model to the low fidelity model. [30] implemented Adaption using following routing: at the beginning the high fidelity model gets an airfoil blank, evaluates it (is the blank already optimally shaped) and if it needs to be optimized, the high fidelity model forwards the blank to the low fidelity model. After the iterative optimization process does the low fidelity model forward the airfoil result back to the high fidelity model were it gets examined again. If the airfoil is still not perfectly shaped after the criteria of the high fidelity model, does the high fidelity model forward the airfoil back to the low fidelity model. Other publications which were investigated by [16] and which researched by themselves multi fidelity modelling using adaptation are: [34][35][36]. The next model management strategy presented in Fig. 2.7 is Fusion. [37] is using Fusion to compute aerodynamic parameters (e.g., flow) of a given airfoil falling back on computational flow dynamics (CFD) and kriging. CFD gets used to compute numerical flow simulations and kriging is used for interpolation. [37] uses a high fidelity CFD and a low fidelity CFD model. In the first step an input space gets created where the individual input samples represent locations. In the second step two input sets get defined: a huge low fidelity set and a small high fidelity set. In the third step both input sets get propagated through the corresponding CFD models. The samples from the low fidelity set get propagated through the low fidelity CFD model, the samples from the high fidelity input set get propagated through the high fidelity CFD model. After the applying the low fidelity CFD model, kriging gets applied to the low fidelity results to interpolate the flow parameters for the samples from the input space which were not propagated through the low fidelity CFD model. In the fourth step the interpolated low fidelity output set and the high fidelity output set get merged together. The goal is to correct the interpolated low fidelity output set with the high fidelity output set with a kriging bridge function. The last model management system represented in Fig. 2.7 is Filtering. Using as example the research made in [22], Filtering can be described as follows: [22] built a Metropolis–Hastings Markov Chain Monte Carlo which as the goal to compute the next state of the Markov Chain as computationally efficient as possible. The solution is to no apply all possible states of the state space to the high fidelity model but first

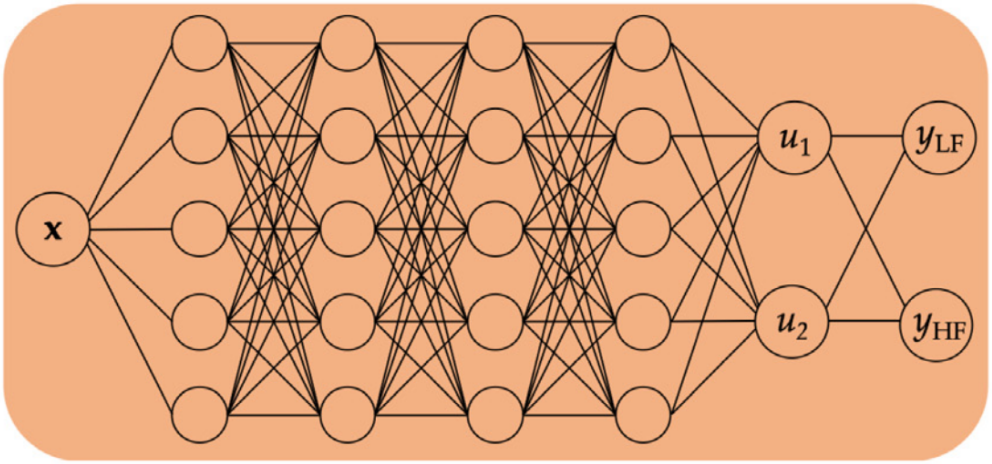


Figure 2.8.: Original All-In-One architecture of a MF-DF network. [5].

apply those to low fidelity model where the individual states get evaluated and promoted if they are likely to be a successor state of the current state. Apply the high fidelity model then only on those states which got promoted by the low fidelity models. This low fidelity based pre-selection of the states lowers the computational cost dramatically.

This Masterthesis works with neural networks. Multi fidelity related neural networks can be related to the model management strategy Fusion. Unlike the Fusion model in [37] does the neural network not consists out of two separate models, but through the training of the neural netowork with multi fidelity data (e.g., transfer learning) does the neural network automatically merge the knowledge of the different fidelity models together during each propagation - like the kriging bridge function merges the models of different fidelity together in [37].

2.4. Multi-Fidelity and CNNs

Many NNs utilised for MF modelling are fusion models (model management strategy: fusion), where each prediction gets made by aggregating the LF and HF predictions. Due to the research achievements, fusion models and their architectures are popular nowadays for MF modelling (surrogate modelling). Several strategies were developed to make fusion models as efficient as possible. The brand-new paper by Zhang et al.[10] fetches the investigation and implementation history of fusion models used regarding MF modelling from the current state-of-the-art perspective. It disaggregates the development history, describes the current state-of-the-art mainstream approaches, and tries to beat them by the proposal of a novel approach that is derived from the current mainstream methods. Therefore, [10] will be used as the main source, and all information will come from there if not noted otherwise.

There are two mainstream architectures which dominate the design of fusion models [10]: **Multi-Fidelity Data-Fusion** (MF-DF) and **Transfer-Learning Neural Networks** (TLNN).

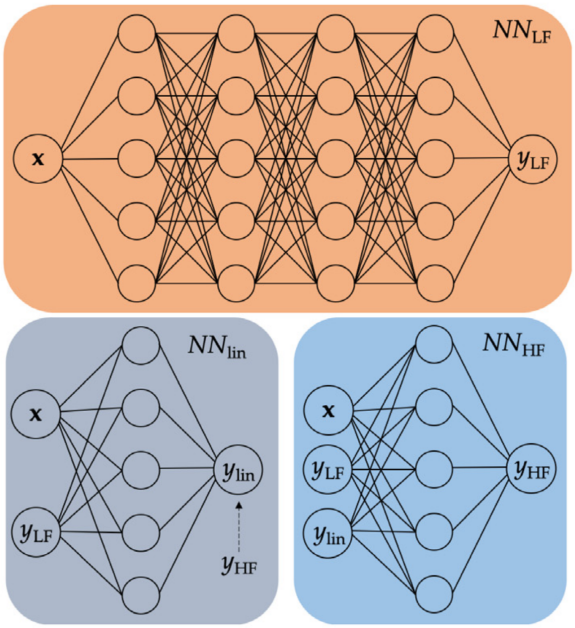


Figure 2.9.: 3-level MF-DF architecture [5]. A first representative of multi-level MD-DF architecture whose goal is to improve the All-In-One network architecture.

The original MF-DF architecture is the All-in-One network displayed in Fig. 2.8. The All-In-One architecture has the advantage that it learns the relation between the LF and HF function implicitly. Learning the LF/HF relation implicitly means that the model gets as input a sample X and derives internally first its LF value and, in a second stage, the HF value. The disadvantage of this structure is, that the network learns two different relations (X/LF and LF/HF) at the same time. Problems occur if these two relations are of different magnitudes, then the architecture underperforms heavily compared to comparable networks of similar size. To solve the problem of learning multiple relations of different magnitudes in the same model architecture, the multi-level MF-DF architecture was developed. The main idea behind those multi-level MF-DF architectures is to distribute the relations over several networks where each model learns one relation (a "level") and then assembles all networks to a composite NN. Guo et al. [5] verified in their research that the multi-level MF-DF architecture leads to better performance than the original All-In-One MF-DF architecture. Meng et al. [6] was one of the first to develop a 3-level MF-DF depicted in Fig. 2.9. In a 3-level MF-DF architecture, the All-In-One MF-DF network is split into three separate networks, and each of these three networks is trained separately with (partially) different datasets. E.g., in Fig. 2.9 the orange network NN_{LF} learns the relation $f_{LF} : X \rightarrow Y_{LF}$ between the input feature X and the LF predictions Y_{LF}, the grey network NN_{lin} learns the between linear relation $f_{HF_{lin}} : (X, Y_{LF}) \rightarrow Y_{HF_{lin}}$ between the input features X, the LF predictions Y_{LF} and the (linear) HF predictions Y_{HF}. The blue and last network learns the non-linear relationship $f_{HF} : (X, Y_{LF}, Y_{HF_{lin}}) \rightarrow Y_{HF}$ between the input features X, LF predictions Y_{LF}, linear HF predictions $Y_{HF_{lin}}$ and the non-linear HF predictions Y_{HF}. Due to this structure, does the

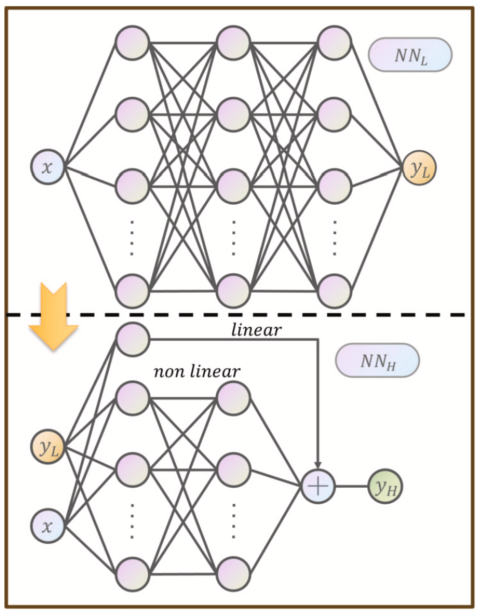


Figure 2.10.: 2-level MF-DF architecture [10]. The goal is to reduce the number of trainable parameters and make the network more resilient against overfitting by using fewer networks than the 3-level MF-DF architecture.

3-level MF-MF architecture outperform the All-in-One MF-DF architecture if it comes to complex LF/HF relations. A disadvantage of the 3-level MF-DF architecture is that it needs more parameters to compute the output predictions just because it consists of three separate networks. The bigger number of parameters leads faster to overfitting for smaller datasets compared to the original All-in-One MF-DF architecture. If a model overfits, it starts to learn the train dataset value-by-value and does not learn any pattern - this is not wanted.

An approach to minimise the number of parameters of the 3-level MF-DF is to use fewer networks by fusing learning tasks. Motamed [8] and Wang et al. [7] used a 2-level MF-DF architecture similar to the one depicted in Fig. 2.10. The 2-level MF-DF consists of 2 networks, which get assembled to one composite NN. Like the 3-level MF-DF architecture possesses the 2-level MF-DF architecture a NN_L which learns the LF function (upper network in Fig. 2.10). The difference between the 2-level and the 3-level MF-DF architecture is, that the 2-level architecture unites the learning of the linear- and non-linear LF/HF relation in a single network. The 2-level MF-DF learns both, the linear and the non-linear relation separately in the same network and predicts out of their aggregation the HF values y_H, see Fig. 2.10. The 2-level MF-DF architecture is the attempt to increase the performance relative to the original All-in-One MF-DF architecture by distributing the relations to be learnt over several networks and to lower the risk of overfitting for smaller train datasets compared to 3-level MF-DF architecture by lowering the number of parameters by using two and not three different networks in its architecture. If the goal is to minimise the amount of trainable parameters to further minimise the overfitting problem, TLNN can be the solution. TLNN is the second big

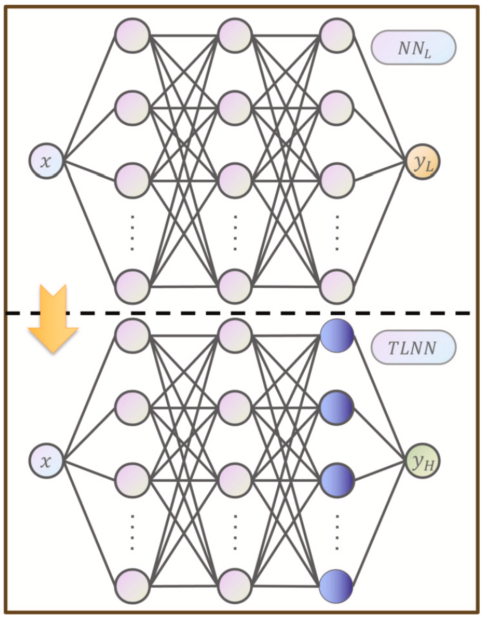


Figure 2.11.: TLNN [10]. Transfer Learning is the second big mainstream method to apply multi-fidelity modelling.

mainstream approach regarding multi-fidelity modelling using NN.

Transfer-Learning is a special method of training a neural network. A typical Transfer-Learning process is shown in Fig. 2.11. According to [38] Transfer-Learning can be described as follows: first there should be defined two datasets - a source D_S and a task domain T_S, and there must be a source D_T and a task target T_T, where $D_S \neq D_T$ and $T_S \neq T_T$. Usually do the source and target domain not share any samples. The Transfer-Learning is done in two separate steps. In the first training, the whole network gets pre-trained with the source dataset. The first training is depicted by the upper half of Fig. 2.11. After the first training, some network layers get frozen. Freezing a layer means that all weights of the affected layer will not be updated (not trained because of being immutable) in the following second training step. In Fig. 2.11, the purple layer in the lower half of the figure marks the unfrozen layer - all layers to the left of the purple layer are frozen. After the first pre-training with the source dataset and freezing of layers, the second training gets started - this time with the task dataset. The forward propagation during the training gets done through all layers, but only the unfrozen layers will be updated during the backpropagation. The usage of Transfer-Learning in MF modelling is a special case because here, the big LF dataset defines the source dataset (pre-training), and the smaller HF dataset defines the task dataset (transfer-learning aka. as fine-tuning). The Master Thesis focuses on CNNs and their role in MF modelling. A typical CNN architecture is structured as follows: first is the block of convolutional layers and the input, followed by a second block of FC layers and the output. The convolutional layers do feature extraction, and the fully-connected layers at the end do feature relation detection and classification/regression. If using transfer learning on CNNs, typically, the pre-trained

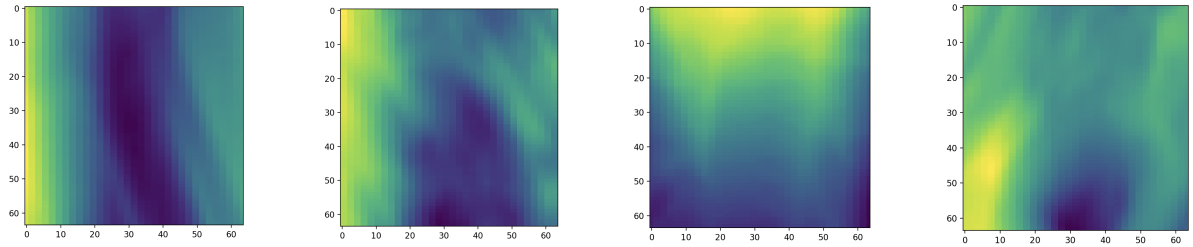
convolutional layers get frozen, and the unfrozen fully-connected layers get fine-tuned using transfer learning. In a nutshell, if combining multi-fidelity modelling, transfer learning and CNNs, then the LF data trains the pattern recognition of the model and the HF dataset trains the classification/regression. A quick comparison between the MF-DF and TLNN architecture: MF-DF models learn the LF/HF relation explicitly, while TLNN models learn this relation implicitly. The advantage of the TLNN against MF-DF architecture is the smaller number of parameters in the network and, therefore, the reduced risk of overfitting during training. This results also out of the fact that in the TLNN architecture, a single network gets trained and not two different networks like in the 2-level MF-DF architecture. The disadvantage of the TLNN architecture is that it can have convergence problems e.g., because of negative transfer. Negative transfer describes the effect where the network unlearns knowledge during the transfer-learning. After the transfer learning, the network cannot do a task it was capable of as it was only pre-trained.

3. Evaluating Convolutional Neural Networks in Multi-Fidelity Modeling

The two mainstream methods regarding MF modelling are MF-DF and TLNN. So far, so good. Until now, theoretical concepts have been discussed. From now on, the practical implementation of theoretical concepts will be documented. The implementations get realised by building APIs, making them accessible for input data and embedding representative model architectures in those APIs. In this section, the crucial necessities like data, software, API design and chosen model architectures will be discussed. The APIs are meant to check the architectures on their MF modelling performance. The results of these experiments will be presented and discussed in the next section. An important selection criterion for the representative model architectures is that those are CNNs due to the initial focus of the Master Thesis on this type of NN.

3.1. Datasets

As depicted in the next sections, three different model architectures were chosen to be investigated further. The MDACNN [2] represents the MF-DF architecture, the MFCNN-TL [9] represents the TLNN architecture, and the MF-TLNN [10] is more or less a composite network, mixing both main architecture types (MF-DF and TLNN) into one model. Due to different architectures and different inputs, each model architecture needed its individual dataset - therefore, three different datasets were used throughout the investigations. All datasets originate from the same source because all references were obtained by analysing the same rotating (rover-) wheel moving through the sand. All datasets were provided by the DLR [3, 4].
The MDACNN takes tabularised data. The data is structured as visualised in Fig. 3.4b. The only difference to the tabularised data in Fig. 3.4b is that the used dataset is not 2D but 12D. The 12 dimensions define the 3D positional coordinates of the centre of the wheel regarding the surface, 3D translational velocity, 3D rotational velocity, and 3D gravity vector. The ground truth of the 12D data is also 12D and defines the LF and HF values for the 3D force and 3D torque. The MFCNN-TL possesses as CNN several convolutional layers and processes images. An according dataset provides 64x64 images which show the sand structure related to the movement of the wheel through the sand. Samples of these images are shown in Fig. 3.1. The MFCNN-TL architecture does transfer learning, there the network needs two types of images: one set of images with only LF values for the 6D ground truth features (3D force, 3D torque) and one set of images with only HF values. The third and last network is the MF-TLNN model architecture. The model is a composite-network-like build and requires



(a) Example image 1 (b) Example image 2 (c) Example image 3 (d) Example image 4

Figure 3.1.: Typical examples for images with content. The originally provided dataset was filtered after those images and these images were used to train the models.

two different inputs for a prediction - an image and the corresponding LF value. The images are the same 64x64 sand structure showing images like for the MFCNN-TL. But to train the network, the model architecture requires images with bi-fidelity features - the 6D ground truth is needed twice with feature values for LF and HF. The 6D ground truth (3D force, 3D torque) holds six different features out of those only three are interesting for the research: the traction force (X-dim. force), the normal force (Z-dim. force) and the yaw rotation (Y-dim. torque). All experiments were done using the traction force only. The traction force was used as an exemplary feature because due to redundancy all model architectures and all performance optimization methods on the model architectures are 1:1 applicable to the other two features of interest as well if they work for the traction force.

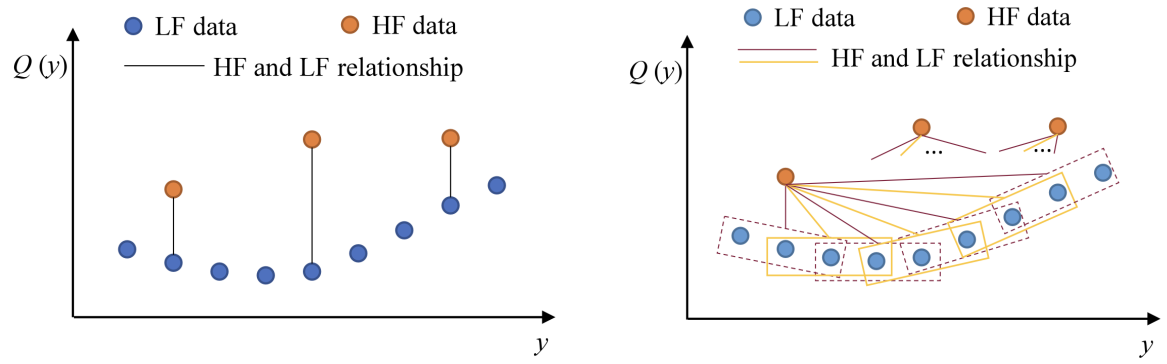
3.2. Software

All experiments, all APIs, and model architectures were implemented using the programming language Python. The model architectures were implemented using the library TensorFlow and there, especially Keras [39]. All Python files were edited and executed using PyCharm as integrated development environment (IDE). PyCharm was executed using the operating system (OS) Somona 14.6.1 on an M1 System-on-Chip (SoC) with an ARM architecture. Four experiments were carried out as part of the master thesis. The code for the two experiments regarding the MDACNN architecture can be found in [1], the code for the experiment regarding the MFCNN-TL architecture in [2] and the code for the experiment regarding the MF-TLNN architecture in [3].

[1] `https://github.com/pwkutz/MasterThesis__MDACNN.git`
[2] `https://github.com/pwkutz/MasterThesis__MFCNN-TL.git`
[3] `https://github.com/pwkutz/MasterThesis__MT-TLNN.git`

3.3. MDACNN architecture

The model chosen to represent the MF-DF architecture is the Multi-Fidelity Aggregation Convolutional Neural Network (MDACNN), which was presented by Chen et al. [2] in 2021. The model was chosen because it combines the current state-of-the-art approach regarding MF-DF networks and combines it with CNN architecture.



(a) Point-Point relation [2].

(b) Point-Domain relation [2].

Figure 3.2.: Left: A typical Point-Point relation [2] learned by the models designed by Liu et al. [7], Motamed [8] and Meng [6]. Point-Domain relation learned by the MDACNN designed by Chen et al. [2].

The MDACNN architecture presented by [2] can be categorised as follows in the MF-DF architecture universe: out of the original All-in-One MF-DF architecture developed Meng et al. [6] a 3-level MF-DF architecture where the first network learns the X/LF relation, the second network the linear LF/HF relation and the third network the non-linear LF/HF relation. Out of the 3-level MF-DF architecture got developed the 2-level MF-DF architecture by Liu et al. [7] and Motamed [8] where the first network is similar to the first network in the 3-level MF-DF architecture but the second network in the 2-level MF-DF architecture learns the linear and nonlinear LF/HF relation at once, see the lower NN_H architecture in Fig. 2.10, and represents therefore the second and third network in the 3-level MF-DF architecture in one model. The MDACNN architecture drops the first network in the 2-level MF-DF architecture, keeps its second network and attaches to the front of the network a convolutional layer, creating effectively a CNN. The main idea behind using a convolutional layer at the front is to increase the performance of the model architecture by improving the predictions by increasing the database that the model can access per prediction. The original MF-DF, the 3-level MF-DF and the 2-level MF-DF architectures have all in common that they learn a Point-to-Point relationship depicted in Fig. 3.2a. The Point-to-Point relationship for the MDACNN architecture can be expressed by

$$f_{HF}(x) = f_{MF}(x, f_{LF}(x)) \tag{3.1}$$

due to its two inputs x and $f_{LF}(x)$.

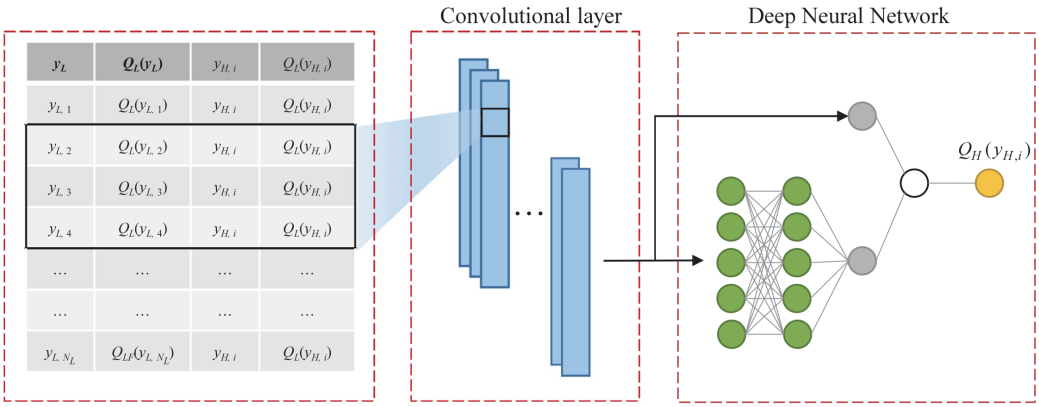


Figure 3.3.: The architecture of the MDACNN [2]. The network learns linear and non-linear features into two separate network branches before they get aggregated together before the final prediction happens. The linear relationship gets learned using the characteristic skip-connection, which is inspired by the residual connections from the ResNet [40].

Chen et al. [2] extend the concept of the 2-level MF-DF by upgrading the learning result from a Point-to-Point relationship to a Point-to-Domain relationship depicted in Fig. 3.2b. The basic idea behind using Point-to-Domain instead of Point-to-Point relations can be described as follows: in the first step, a sample x gets defined whose HF value needs to be computed. In the second step, the closest neighbours $\mathbf{x}$ of sample x get identified. In the next step, the LF values $f_{LF}(x)$ and $f_{LF}(\mathbf{x})$ of the sample x and its neighbours $\mathbf{x}$ get defined. The sample x, its neighbours $\mathbf{x}$, and their LF values define a local domain, and in the last step, the MDACNN architecture model computes out of this domain the HF value for the initial sample x. This was just the breath idea behind using local domains for Point-to-Domain relations. Chen et al. [2] went a step further by using not only the closest neighbours but the whole dataset for a single prediction. The dataset gets split up into a set of local domains, as depicted in Fig. 3.2b. Combining all local domains into a global domain allows Chen et al. [2] to use the whole dataset to predict a single HF value. The convolutional layer is important because its kernel splits the dataset into a set of local domains. Each kernel position defines a local domain.
To enable the usage of multiple local domains per prediction (to use the whole LF dataset with all its samples), the data must be organised in a tabular form, and a moving window needs to be introduced. The table is divided into columns with LF samples $\mathbf{x_{LF}}$ and their LF values $f_{LF}(\mathbf{x_{LF}})$ and into columns for an HF sample x_{HF} and its LF value $f_{HF}(x_{HF})$. The columns should be ordered according to the structure [$\mathbf{x_{LF}}$, $f_{LF}(\mathbf{x_{LF}})$, x_{HF}, $f_{LF}(\mathbf{x_{LF}})$], similar like depicted in Fig. 3.4. The organisation of the tabularised data is from a row perspective quite easily: each row corresponds to an LF sample. The moving window will go row-wise step-by-step through the table, and in each step, it computes the relation of the HF sample x_{HF} with the current local domain of LF samples $\mathbf{x_{LF}}$ and their LF values $f_{LF}(\mathbf{x_{LF}})$. Each table is associated with one HF sample x_{HF} whose HF value $f_{HF}(x_{HF})$ needs to be predicted. This

y_L	$Q_{L1}(y_L)$	$Q_{L2}(y_L)$	$y_{H,i}$	$Q_{L1}(y_{H,i})$	$Q_{L2}(y_{H,i})$
$y_{L,1}$	$Q_{L1}(y_{L,1})$	$Q_{L2}(y_{L,1})$	$y_{H,i}$	$Q_{L1}(y_{H,i})$	$Q_{L2}(y_{H,i})$
$y_{L,2}$	$Q_{L1}(y_{L,2})$	$Q_{L2}(y_{L,2})$	$y_{H,i}$	$Q_{L1}(y_{H,i})$	$Q_{L2}(y_{H,i})$
$y_{L,3}$	$Q_{L1}(y_{L,3})$	$Q_{L2}(y_{L,3})$	$y_{H,i}$	$Q_{L1}(y_{H,i})$	$Q_{L2}(y_{H,i})$
$y_{L,4}$	$Q_{L1}(y_{L,4})$	$Q_{L2}(y_{L,4})$	$y_{H,i}$	$Q_{L1}(y_{H,i})$	$Q_{L2}(y_{H,i})$
…	…	…	…	…	…
…	…	…	…	…	…
y_{L,N_L}	$Q_{L1}(y_{L,N_L})$	$Q_{L2}(y_{L,N_L})$	$y_{H,i}$	$Q_{L1}(y_{H,i})$	$Q_{L2}(y_{H,i})$

(a) Multiple LF functions [2].

y_{L1}	y_{L2}	$Q_L(y_{L1}, y_{L2})$	$y_{H1,i}$	$y_{H2,i}$	$Q_L(y_{H1,i}, y_{H2,i})$
$y_{L1,1}$	$y_{L2,1}$	$Q_L(y_{L1,1}, y_{L2,1})$	$y_{H1,i}$	$y_{H2,i}$	$Q_L(y_{H1,i}, y_{H2,i})$
$y_{L1,2}$	$y_{L2,2}$	$Q_L(y_{L1,2}, y_{L2,2})$	$y_{H1,i}$	$y_{H2,i}$	$Q_L(y_{H1,i}, y_{H2,i})$
$y_{L1,3}$	$y_{L2,3}$	$Q_L(y_{L1,3}, y_{L2,3})$	$y_{H1,i}$	$y_{H2,i}$	$Q_L(y_{H1,i}, y_{H2,i})$
$y_{L1,4}$	$y_{L2,4}$	$Q_L(y_{L1,4}, y_{L2,4})$	$y_{H1,i}$	$y_{H2,i}$	$Q_L(y_{H1,i}, y_{H2,i})$
…	…	…	…	…	…
…	…	…	…	…	…
y_{L1,N_L}	y_{L2,N_L}	$Q_L(y_{L1,N_L}, y_{L2,N_L})$	$y_{H1,i}$	$y_{H2,i}$	$Q_L(y_{H1,i}, y_{H2,i})$

(b) Multiple features per sample [2].

Figure 3.4.: The data tables necessary for the MDACNN architecture can be easily extended to process multiple LF functions (left) and multiple features per sample (right)

row-wise step-by-step locomotion of a moving window is equal to the locomotion of a kernel in a convolutional layer - therefore, this processing step gets realised using a convolutional layer in the MDACNN.
Chen et al. [2] updated the 2-level MF-DF architecture and the newly designed input table (tabularised input data) is depicted in Fig. 3.3. The tabularised data brings some advantages. The tabularized data makes it possible to easily extend the given input data with data from more fidelities without significantly changing the architecture of the MDACNN. Fig. 3.4a shows a 3-fidelity case where the system consists of one HF function and two LF functions. The extension happens just by adding for each new LF function a new column - both LF functions use the samples $\mathbf{x_{LF}}$ from the same big LF dataset. Another effect of the tabularised data is that the dimensionality of the LF and HF samples is less important and does not form a bottleneck for the functional capability of the MDACNN architecture. Due to the tabularization, the dimensionality of the samples can be increased or decreased simply by adding or removing columns in the table. Each column in the table represents a feature (a dimension). Fig. 3.4b shows how the network can be easily extended to data of higher dimensions.

In summary, it can be concluded that due to the tabularised data, the MDACNN architecture possesses advantages over the conventional 2-level MF-DF architecture.

1. The MDACNN architecture defines a single network and can be trained in a single training run. The multi-level MF-DF architectures are composite networks where each network needs to be trained separately before the network can be used.

2. The MDACNN architecture uses the whole LF dataset as a base for a prediction. The MDACNN architecture switches from a Point-to-Point to a Point-to-Domain relation.

3. The MDACNN architecture can be easily extended to process data of higher dimensionality with more fidelity functions. Unlike the 2-level or 3-level MF-DF architecture, the

MDACNN architecture does not need to change its structure significantly for that.

The MDACNN architecture is shown in Fig. 3.3. Regarding the hyper-parameters, the MDACNN uses the Mean Squared Error (MSE) as a loss function and the Adam Optimizer with a learning rate of 0.001 to update the weights during Gradient Descent. In [2], it is nowhere explicitly written that pre-processing of the input data like normalisation or regularisation methods like batch normalization, pooling and dropout are used. The NN starts with a convolutional layer. The kernel width is 3. The kernel height is equal to the number of columns in the input table and depends, therefore, on the dimensionality of the LF and HF samples and the number of fidelities considered. In total, the kernel shape can be defined by 3xN. With a kernel width of three, the kernel always considers 3 LF samples at the same time per local domain. Because the kernel height is equal to the number of columns, the executed convolution on multi-dimensional input data can be compared to a 1D convolution. The kernel moves with a stride of one. The convolution results get flattened and forwarded into the linear and non-linear branch. The non-linear branch (learns the non-linear LF/HF relation) is visualised with the green neurons in Fig. 3.3. The colour green marks neurons with a non-linear activation function. The non-linear branch consists of three layers. The first two layers possess each ten neurons with a tanh activation function. The last layer possesses a single grey neuron. The colour grey marks neurons with a linear activation function. The linear branch - represented by the skip-connection - possesses just a single grey neuron with a linear relationship. The outputs of both branches get aggregated together and forwarded to the output neuron. Due to the regression task of the MDACNN, has the yellow output neuron a linear activation function.

Algorithm 1 API MDACNN architecture

```
1:  procedure Train and Test, and Performance Check
2:      if modus = "Train and Test" then
3:          data ← retrieveBasicData()
4:          TrainData ← Dataset(data)
5:          ValData ← Dataset(data)
6:          TestData ← Dataset(data)
7:          MDACNNmodel ← MDACNN()
8:          MDACNNmodel ← trainMDACNN(TrainData, ValData)
9:          MDACNNmodel.analysis(TestData)
10:     else if modus = "Performance Check" then
11:         data ← retrieveCommonDataset()
12:         DataTables ← Dataset(data)
13:         MSEloss ← CrossValidation(DataTables)
```

Chen et al. [2] developed their MDACNN architecture using the datasets built out of LF and HF functions presented in Fig. 4.1. The investigations as part of the master thesis checked the MDACNN architecture performance on the benchmark datasets, but the main focus during the experiments was on the performance of the MDACNN architecture if applied to

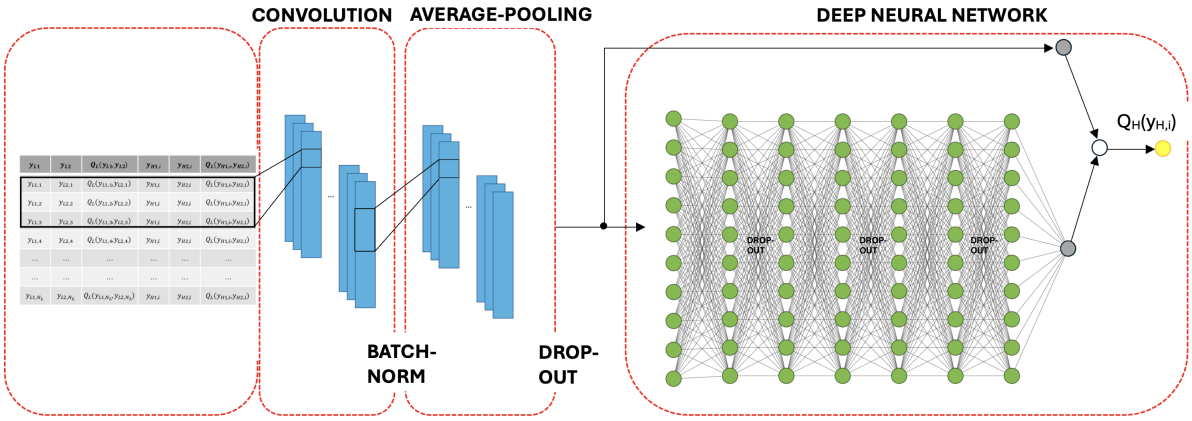


Figure 3.5.: The regularized 7x32 MDACNN architecture. The layer is a convolutional layer with 64 3x26 kernels and with stride 1. It follows a batch normalisation layer and an average-pooling layer with a 5x1 kernel and stride 3. After flattening, follow a dropout layer and the architecture's linear and non-linear branches (Deep Neural Network). The non-linear branch consists of 7 FC layers with 32 non-linear neurons each. Every two FC layers is located a dropout layer. The linear branch consists of a single neuron with a linear activation function. The output is formed by a linear neuron due to the regression task.

the dataset provided by the DLR [3][4]. The investigations concluded that the performance of the MDACNN architecture on the provided custom data was poor but could be improved. As a result of the following performance optimisation efforts was developed the Regularised 7x32 MDACNN architecture, as shown and described in Fig. 3.5. Out of all developed architectures, it has the one with the lowest loss with a reasonable amount of parameters for an MF model. All conclusions which lead to the design of the Regularised 7x32 MDACNN architecture are described in section 4.1.

During the investigations, an API for processing the MDACNN architecture was developed. The API enabled experiments with the network. The API is visualised in Alg. 1 as pseudo code and in Fig. A.1 as a flowchart. The implemented MDACNN model is embedded into that API. The API generally allows two main processes: training and testing with the default train and test dataset ("Train+Test") or checking the overall performance of the model by using K-Fold Cross-Validation ("Performance Check"). If activating the first main process, "Train+Test", the first step is to load the dataset and pre-process it. During the pre-processing, the data gets tabularised, and train, validation and test datasets get generated. In the second step, the MDACNN model gets initialised and trained using the train and validation dataset. In the third step, the trained MDACNN model gets analysed by checking its performance with the test dataset; the loss gets measured using MSE and the predicted HF values for the test samples get plotted. If activating the second main process, "Performance Check", then the train and test datasets get taken and merged into one big dataset. This one dataset gets then pre-processed like described for "Train+Test". Then, a 10-Fold Cross-Validation is performed

by training the same model ten times with different train datasets to check its overall ability to generalise. Ten folds for the cross-validation were chosen because a 90/10 split of the dataset into train/test datasets is common practice. The performance check via cross-validation is done because the given dataset has a predefined split into train and test datasets. This predefined split into train and test datasets is only given for the MDACNN architecture, not for the other architectures. The test dataset contains several smaller sub-datasets, from which the biggest two are chosen to be the Validation and Test dataset during "Train+Test". By shuffling all datasets together and training and testing the model on datasets with varying favourable and unfavourable splits, the true performance of the model architecture gets captured.

3.4. MFCNN-TL architecture

The discussed MDACNN architecture [2] is a CNN but possesses only a single convolutional layer, and the rest of the network is a perceptron. And that single convolutional layer is not used to process images - for what convolutional layers originally were intended for - but data tables.

However, MF modelling can also be carried out with larger CNNs with more convolution

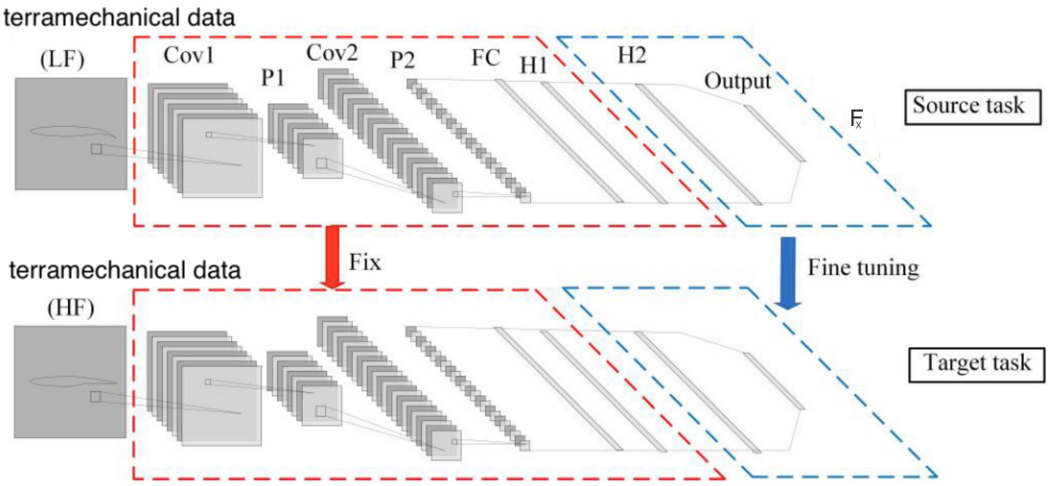


Figure 3.6.: The original MFCNN-TL architecture by [9], processing data provided by the DLR [3][4]. The predicted feature is the traction force F_X. The figure visualizes the structure of the transfer learning process

layers and with other architectural approaches than the MF-DF architecture. CNNs have the advantage that they are very effective in the extraction of local patterns in spatial data like images. Therefore, a typical application of CNNs is image recognition, which is commonly used in areas like aerospace. A typical application in aerospace is the design optimisation of wings. Liao et al. [9] designed the MFCNN-TL architecture, which is a CNN, implemented into an optimisation framework to optimise the shape of wings and make them as aerodynamic as

possible. The work of Liao et al. [9] was chosen due to several reasons. First, the designed MFCNN-TL architecture is a representative of the TLNN architecture type, therefore, one of those architectures learning the LF/HF implicitly. Second, the MFCNN-TL architecture is supposed to be trained by transfer learning. Third, the MFCNN-TL architecture is an MF model. Fourth, due to the fact of being designed as an MF model, the MFCNN-TL architecture is designed to be a small-scale and light-weighted architecture like depicted in the Fig. D.1 and Fig. D.2 (appendix). Fifth, the MFCNN-TL architecture is a CNN that is supposed to process images. Therefore, [9] will be used as the main information source in this section, and if not marked otherwise, all information will come from there. The original MFCNN-TL architecture is depicted in Fig. 3.6. The architecture expects as input data 128x128 greyscale images. The MFCNN-TF architecture represents a CNN, which consists of two convolutional and two FC layers with already implemented regularisation. The architecture is designed as follows: The first layer is a convolutional layer whose goal is to extract the shape from the objects on the input image. The original architecture uses four Sobel functions as kernels.

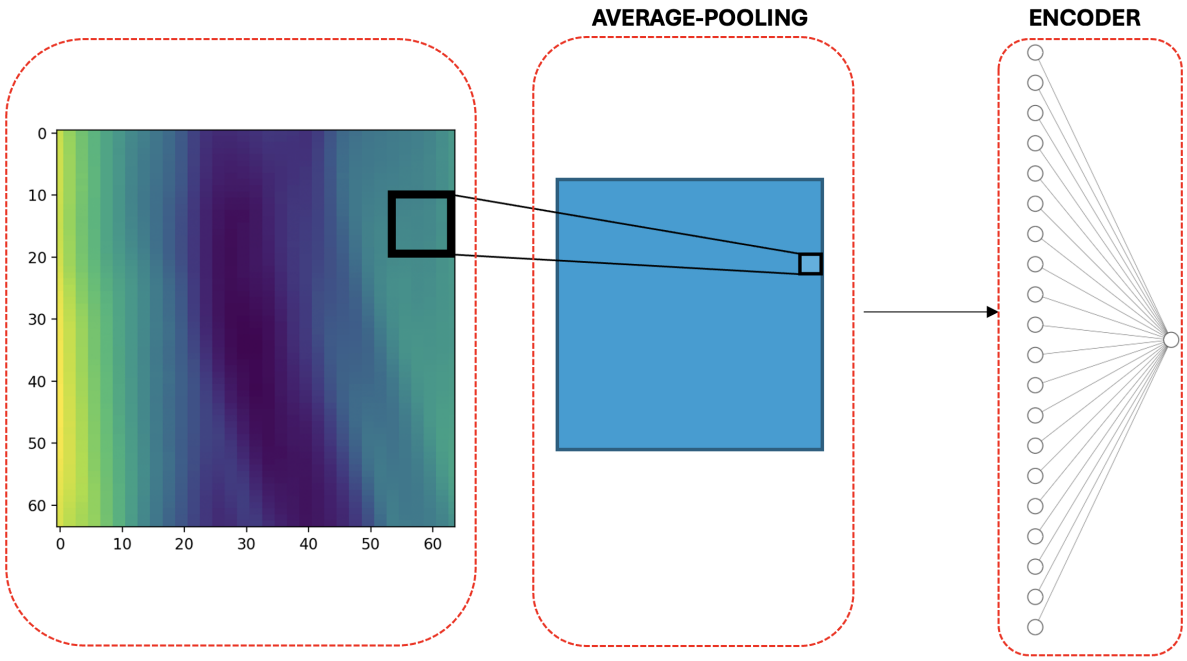


Figure 3.7.: The Perceptron MFCNN-TL architecture. The first layer is an average pooling layer with a 2x2 kernel and a stride of 2. After flattening, the first FC layer has 1024 nonlinear neurons, and the output is formed via a single linear neuron.

A Sobel function is a 3x3 kernel specialising in extracting lines (of a certain orientation) in an image. In this case, the four Sobel functions cover the lines of angles ± 45. During the investigations, the amount of kernels in the first convolutional layer was extended to improve the shape extraction. The kernels get moved with a stride of 1 over the input image. The output is a 126x126 image. After the first convolutional layer follows a max-pooling layer to delete the redundancies among the data and condense the information density. The

max-pooling layer possesses a 2x2 kernel with a stride of 1. The second convolutional layer extracts advanced geometric features like thickness, relative height, curvature, and tangent direction out of the found wing shape. The layer possesses 12 5x5 kernels with a stride of 2. It follows again a max-pooling with a 2x2 kernel and a stride of 2. Now, the output gets flattened and forwarded into the first FC layer. The MFCNN-TL architecture possesses two FC layers of the same design. Both FC layers possess 1024 neurons with a PReLU, aka. Leaky ReLU as a non-linear activation function. The PReLU is defined by:

$$\begin{cases} y = 0.1x, & x \leq 0 \\ y = x, & 0 \leq x \end{cases}.$$

The output layer predicts several predictions due to multi-task learning. In multi-task learning, the model predicts several features for a single input sample. Although the investigations of the Master Thesis prioritise three features (traction force, normal force and yaw rotation) and multi-task learning would have been an option, the implemented MFCNN-TL model architecture will learn just a single feature per training run. As loss function gets used MSE. Fig. 3.6 documents the structure of the fine-tuning of the MFCNN-TF. The whole network gets trained in the first training step with the Source dataset D_S - the LF dataset. In the second training step, all layers get frozen, which are encircled by the red line in Fig. 3.6 - only the blue encircled layers remain trainable and get fine-tuned with the Target dataset D_T aka. HF data. Effectively only the last two layers get trained with the HF dataset.
During the investigations associated with the master thesis, the original MFCNN-TL architecture was applied to the dataset provided by the DLR[3][4]. These investigations conclude that the original MFCNN-TL architecture performs poorly on the custom dataset, but improving its efficiency by utilising performance optimisation by altering the model architecture is possible. The 1x8+2x24 MFCNN-TL architecture and the 1x24 MFCNN-TL architecture were developed to check the influence on the performance of an emphasised shape feature extraction (extended second convolutional block), respectively, an emphasised shape extraction (extended first convolutional block). The 1x8+2x24 MFCNN-TL architecture is shown and closer described in Figure 3.8 and the 1x24 MFCNN-TL architecture is shown and closer described in Figure 3.9. As a result of the investigations, the Perceptron MFCNN-TL architecture, depicted and described in Fig. 3.7, was developed. The Perceptron MFCNN-TL architecture has the most optimal loss out of all investigated MFCNN-TL-related architectures. Section 4.2 shows the conclusions made which led to the development of the Perceptron MFCNN-TL architecture. For the investigations, the MFCNN-TL architecture was implemented as a model into a corresponding API visualised in Alg. 2 as pseudo code and in Fig. A.2 (appendix) as flowchart. The APIs for the MDACNN and MFCNN-TL architecture implementations do not differ much from each other. Both possess two main processes: "Train+Test" and "Performance Check" - both are already described in the MDACNN architecture section. Of course, there are differences, as, on the one hand, the MDACNN API focuses on tabularizing data during the data pre-processing, and it trains an MDACNN model. On the other hand, does the MFCNN-TL API focus on images during the data pre-processing and train an MFCNN-TL model. A bigger difference is the extended data pre-processing for the MFCNN-TL API

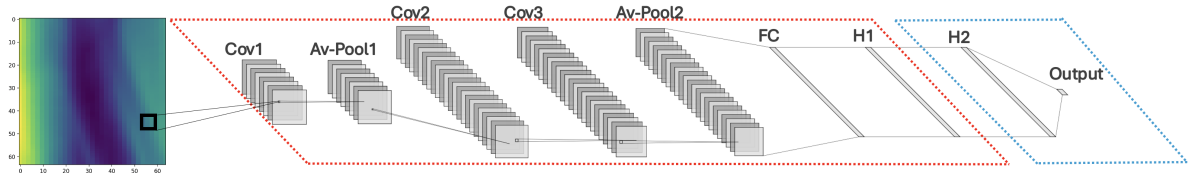


Figure 3.8.: 1x8+2x24 MFCNN-TL architecture. The architecture emphasises an extended second convolutional block and, therefore, shape feature extraction. The shape gets extracted by the first convolutional layer. Convolutional layer Cov1 has 8 3x3 layers with a stride 1. Average-Pooling Layer Av-Pool1 has a 2x2 kernel with a stride 2. The convolutional layers Cov2 and Cov3 have 24 5x5 kernels with a stride 2. The average-pooling layer Av-Pool2 has a 2x2 kernel with a stride 2. FC describes an FC layer with four linear neurons. The FC layers H1 and H2 possess 1024 non-linear neurons each. The output defines a single linear neuron. More information on the 1x8+2x24 MFCNN-TL architecture is in the appendix, see Figure D.3 and fig. D.4. The red box marks all layers which get pre-trained with the LF data set and the blue box marks all layers which get fine-tuned with HF data during transfer learning.

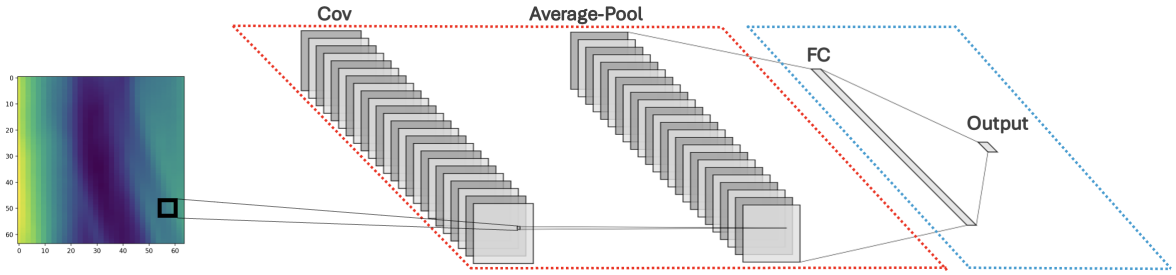


Figure 3.9.: 1x24 MFCNN-TL architecture. The architecture emphasises the first convolutional layer and, therefore, shape extraction. The convolutional layer Cov has 24 3x3 kernels with a stride 1. The average-pooling layer Av-Pool has a 2x2 kernel with a stride 1. The FC layer FC has 1024 non-linear neurons, and the output is built out of a single linear neuron. More information on the 1x24 MFCNN-TL architecture is in the appendix, see Figure D.5 and Figure D.6 The red box marks all layers which get pre-trained with the LF data set, and the blue box marks all layers which get fine-tuned with HF data during transfer learning.

depicted in Fig. 3 as pseudo code and in Fig. A.2 (appendix) as a flowchart. In the first step, unsuitable images get sorted out. Some images in the dataset are without information because they are completely blacked out and do not show any sand structures like the rest of the images. To improve the training and testing performance of the model, these samples get dropped out of the dataset. In the second step, the LF and HF functions get denoised by smoothing their LF and HF values (aka. Y values, aka. ground truth values, aka. target values)

Algorithm 2 API MFCNN-TL architecture

```
1: procedure TRAIN AND TEST, AND PERFORMANCE CHECK
2:     if modus = "Train and Test" then
3:         HFdata ← dataloader.loadTrainData()
4:         LFdata ← dataloader.loadTrainData()
5:         MFCNNTLmodel ← MFCNNTL()
6:         MFCNNTLmodel ← TransferLearning(HFdata, LFdata)
7:         MFCNNTLmodel.analysis()
8:     else if modus = "Performance Check" then
9:         data ← retrieveCommonDataset()
10:        DataTables ← Dataset(data)
11:        MSEloss ← CrossValidation(DataTables)
```

Algorithm 3 Extended Image Preprocessing using function preprocessing(data).
The function gets used inside dataloader.loadTrainData() in Alg. 4 and Alg. 2

```
1: procedure PREPROCESSING(DATA)
2:     data ← RemoveContentlossImages(data)
3:     data ← MovingAverage(data)
4:     data ← NormalizeData(data)
5:     TrainData, TestData ← SplitDataset(data)
6:     return TrainData, TestData
```

using the Moving Average. In the third step, the data gets normalised to increase the training performance of the model. In the fourth and last step, the dataset gets split into a train and test set. This step was not necessary for the MDACNN model because the provided dataset was already pre-divided into train and test datasets. The split into train and test datasets happens randomly. Therefore, the MFCNN-TL model gets trained in each training run with a different train dataset. Due to the already random splitting in the "Train+Test" main process, it is debatable whether the MFCNN-TL API needs its own K-Fold Cross-Validation process.

3.5. MF-TLNN architecture

With the MDACNN architecture [2] and with the MFCNN-TL architecture [9] were introduced networks which learn the LF/HF relation either explicitly or implicitly. Why not combine both learning strategies in one architecture to increase performance? This is the main thought behind the development of the MF-TLNN architecture by Zhang et al. [10].
The original MF-TLNN architecture is shown in the appendix in Fig. E.3. The MF-TLNN architecture is designed for MF modelling. Therefore, the goal of all individual networks which are connected to the MF-TLNN architecture is to learn the LF/HF relation. In a nutshell, the MF-TLNN architecture can be explained as a composite-network-like structure, which computes its output through a weighted aggregation of the predictions of two parallel

arranged networks where one network learned the LF/HF relation implicitly (NN_L in Fig. E.3) and the other one explicitly (AE_L in Fig. E.3). The MF-TLNN architecture needs to be trained in three steps: in the first step, the implicit learning NN_L and the explicit learning AE_L network get pre-trained with the LF dataset. In the second step, both networks get combined via a weighted aggregation of their outputs to the MF-TLNN. In the third step, the whole network gets fine-tuned by freezing all but the last layers of the former NN_L and AE_L networks and applying transfer learning using the HF dataset.

The explicit learning AE_L network is explicitly defined by Zhang et al. [10] as an autoencoder

Algorithm 4 API MF-TLNN architecture

```
1:  procedure TRAIN AND TEST
2:      HFdata ← dataloader.loadTrainData()
3:      LFdata ← dataloader.loadTrainData()
4:      AEmodel ← NN("AE")
5:      NNLmodel ← NN("NNL")
6:      AEmodel ← AEmodel.SourceTraining(LFdata)
7:      NNLmodel ← NNLmodel.SourceTraining(LFdata)
8:      MFTLNNmodel ← NN("MFTLNN")
9:      MFTLNNmodel ← MFTLNNmodel.TaskTraining(HFdata)
10:     MFTLNNmodel.analysis()
```

(AE). An AE has two parts: the encoder and the decoder. The task of AE is to split data from noise by compressing the input sample to a minimum and rebuilding the sample out of its reduced representation. Usually, a high-dimensional sample or an image gets forwarded into the AE, and the encoder compresses the input sample into a low-dimensional representation in the latent space. The compressed representation in the latent space outlines the complete usable data of the input sample without noise and redundancies. The decoder takes this compressed latent space representation and develops it through up-sampling back to the original input sample. However, [10] defines that input into the AE architecture is a single scalar value. Therefore, an inverted AE needs to be implemented, where the decoder and encoder have swapped positions. The scalar input sample gets interpreted as the already compressed latent space representation. The architecture of the constructed AE is depicted in Fig. 3.10 and with a focus on the parameter distribution in Fig. E.1 and Fig. E.2 (appendix). The architecture type of the NN_L network is not as clearly defined as for the AE. The only definition is that the network needs to be an MF model and needs to learn the LF/HF relation implicitly. The NN_L network was implemented using the MFCNN-TL architecture because the MFCNN-TL architecture fulfils all required criteria, and the original MFCNN-TL architecture is a CNN.

As a summary, for the investigations carried out for the master thesis, the MF-TLNN architecture was implemented using the already described AE architecture (as AE_L) and the Perceptron MFCNN-TL architecture (as NN_L) as the best performing version of the MFCNN-TL architecture if applied to the datasets provided by the DLR [3][4]. The architecture of the implemented MF-TLNN architecture is shown and described in Fig. 3.10.

The implementation of the MF-TLNN architecture got embedded into the API shown in Fig. 4 as pseudo code and in Fig. A.3 (appendix) as a flowchart. The MF-TLNN API is mostly similar to the MFCNN-TL API in Fig. 2 and corresponds there to the "Train+Test" branch. Due to the nested MFCNN-TL network in the MF-TLNN architecture processes, the MF-TLNN architecture model images. The pre-processing of the dataset is mostly the same, together with the extended pre-processing API presented in the MFCNN-TL. The only major difference is that the MFCNN-TL API trains only the MFCNN-TL architecture model while the MF-TLNN API trains first the NN_L, AE_L and only then the MF-TLNN architecture model gets initialised and trained as well.

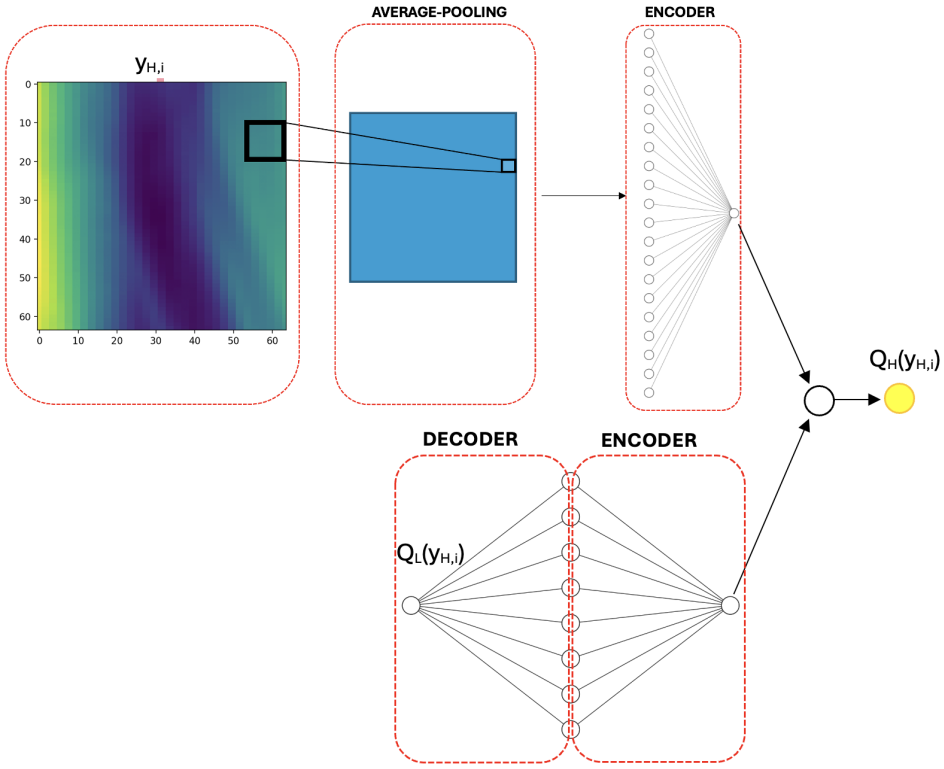


Figure 3.10.: The Perceptron MF-TLNN architecture. The upper model is built according to the Perceptron MFCNN-TL architecture, see Fig. 3.7. The model gets as input an image $y_{H,i}$ and predicts its HF value $Q_H(y_{H,i})$. The lower model is an AE. The model takes as input $Q_L(y_{H,i})$, the scalar LF value of the image $y_{H,i}$. The middle layer consists of eight non-linear neurons that decode the scalar value. The output layer consists of one non-linear neuron that encodes the data back into a scalar value. Both models are combined via a weighted aggregation of their HF value $Q_H(y_{H,i})$ predictions. Both models get first pre-trained on LF data and later on fine-tuned with transfer learning using HF data. More information on the number of parameters of the Perceptron MF-TLNN architecture is provided in Fig. E.4 and Fig. E.5. The utilised dataset is provided by the DLR [3][4].

4. Experiments

This section tests the MDACNN architecture as a representative of the MF-DF architecture type and the MFCNN-TL and MF-TLNN architectures as representatives of the TLNN architecture type. All architectures get investigated using the custom dataset provided by the DLR[3][4] and described in section 3.1. The MDACNN architecture gets tabularised data, and the MFCNN-TL and MF-TLNN architecture get images. The MDACNN architecture will be tested additionally utilising default benchmarks, see Tab. A.1, due to its structure and input data.

4.1. Experiments on the MDACNN architecture

4.1.1. Performance Optimization based on default Benchmarks

In research, benchmarks are commonly used. Also, in the research regarding MF modelling, a benchmark gives the ability to define a default case with a defined desired outcome and enables to investigate the ability to generalise for different architectures. By becoming a default in research, a benchmark makes it possible to define common learning goals. The default benchmarks in MF modelling are shown in Tab. 4.1.

Tab. 4.2 visualizes for each benchmark the hyperparameters applied to the MDACNN architecture model during training.

The seven default benchmarks are: [1)] **Continuous Functions with Linear Relation** (see Fig. B.1a). [2)] **Discontinuous function with linear relationship** (see Fig. 4.1a). [3)] **Continuous function with non-linear relationship** (see Fig. B.2a). [4)] **Continuous oscillating function with non-linear relationship** (see Fig. 4.2a). [5)] **Phase-shifted oscillation** (see Fig. B.3a). [6)] **Different periodicity** (see Fig. B.4a) and [7)] **50-dimensional function** (see Fig. **??**).

Two short disclaimers at the beginning: First, with "Benchmark provided by [2]" like in the caption in Fig. 4.1a means running the benchmark without normalising the input data. Normalisation was implemented afterwards to improve the generalisation and, therefore, the performance of the MDACNN architecture. Normalisation was not applied immediately from the beginning due to the announcement that: "the proposed method can handle data aggregation from multiple sources across different scales" [2], (p. 1). Secondly, all plots shown below and in the appendix visualise how the trained MDACNN architecture models reacted to the applied test dataset. The test dataset was used to check the generalisation capability of the individual models. The MDACNN was trained according to Tab. 4.1 and Tab. 4.2). The almost perfect benchmark results given by [2] and displayed in Fig. B.1a, 4.1a, B.2a, 4.2a, B.3a, B.4a and B.5a could only be verified by propagating the training dataset (known data) through the original MDACNN architecture and could be only achieved with the test

Table 4.1.: All seven default benchmarks used in MF modelling. Each benchmark defines its LF and HF function.

No.		LF model		HF model	
(a)	Continuous functions with linear relationship	$Q_L(y) = 0.5(6y-2)^2 \sin(12y-4) + 10(y-0.5) - 5, \quad 0 \le y \le 1$	(3)	$Q_H(y) = (6y-2)^2 \sin(12y-4), 0 \le y \le 1$	(4)
(b)	Discontinuous functions with linear relationship	$Q_L(y) = \begin{cases} 0.5(6y-2)^2 \sin(12y-4) + 10(y-0.5), & 0 \le y \le 0.5 \\ 3 + 0.5(6y-2)^2 \sin(12y-4) + 10(y-0.5), & 0.5 < y \le 1 \end{cases}$	(5)	$Q_H(y) = \begin{cases} 2Q_L(y) - 20y + 20, & 0 \le y \le 0.5 \\ 4 + 2Q_L(y) - 20y + 20, & 0.5 < y \le 1 \end{cases}$	(6)
(c)	Continuous functions with nonlinear relationship	$Q_L(y) = 0.5(6y-2)^2 \sin(12y-4) + 10(y-0.5) - 5$	(7)	$Q_H(y) = (6y-2)^2 \sin(12y-4) - 10(y-1)^2$	(8)
(d)	Continuous oscillation functions with nonlinear relationship	$Q_L(y) = \sin(8\pi y), 0 \le y \le 1$	(9)	$Q_H(y) = (y-\sqrt{2})Q_L^2(y), 0 \le y \le 1$	(10)
(e)	Phase-shifted oscillations	$Q_L(y) = \sin(8\pi y)$	(11)	$Q_H(y) = y^2 + Q_L^2(y + \pi/10)$	(12)
(f)	Different periodicities	$Q_L(y) = \sin(6\sqrt{2}\pi y)$	(13)	$Q_H(y) = \sin(8\pi y + \pi/10)$	(14)
(g)	50-dimensional functions	$Q_L(y) = 0.8Q_H(y) - \sum_{i=1}^{49} 0.4 y_i y_{i+1} - 50, -3 \le y_i \le 3$	(15)	$Q_H(y) = (y_1 - 1)^2 + \sum_{i=2}^{50} (2y_i^2 - y_{i-1})^2, -3 \le y_i \le 3$	(16)

Table 4.2.: Hyperparameter settings for each benchmark described in Fig. 4.1

No.	Number of epochs	Batch size	Learning rate	Regularization rate	Number of feature maps	Kernel width	Number of LF data	Number of HF data
(a)	5,000	4					21	4
(b)	5,000	5					38	5
(c)	5,000	5					21	5
(d)	5,000	10	0.001	0.01	64	3	51	15
(e)	5,000	10					51	16
(f)	5,000	10					51	15
(g)	1,000	50					1,000	100

dataset (unknown data) if upgrading the API or the MDACNN architecture. Each benchmark is evaluated using three figures and focuses on the generalisation performance of the model. The first image shows the benchmark result provided by [2], like Fig. 4.1a, the second image shows the benchmark results if implementing the API and MDACNN architecture exactly like they are described in [2], like Fig. 4.1b, and the third image shows the benchmark results after improving either the API or the MDACNN architecture, like Fig. 4.1c. The

X-axis of the plots shows the sample values. The sample values are chosen from the interval $[0, 1]$. The Y-axis of the plot shows the function values of the LF-, HF-, and approximated HF-function (MDACNN). The blue graph represents the LF function, the red/orange graph represents the HF function, and the green graph represents the approximated HF function (MDACNN). This section represents a representative of relations among non-oscillating functions in Fig. 4.1 and a representative of relations among oscillating functions in Fig. 4.2. All other benchmark results are moved to the appendix. This was done because the research showed that the MDACNN architecture has significant performance differences when applied to non-oscillating and oscillating functions. The results of the experiments can be summarized as follows:

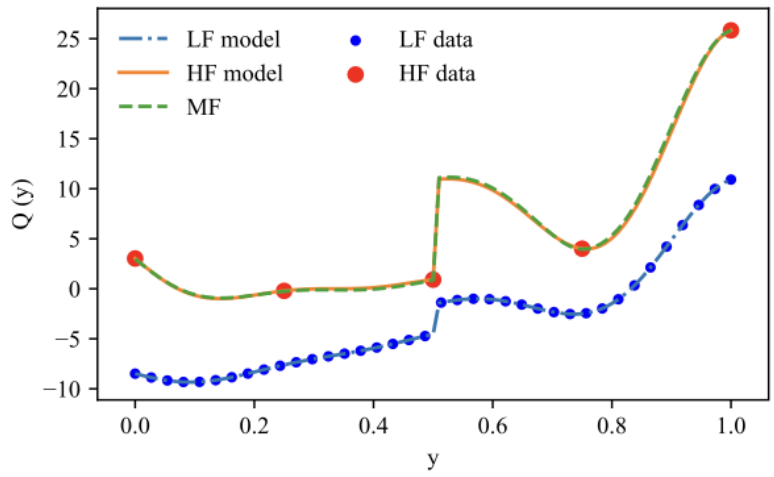
1. The MDACNN architecture generalises well on continuous and discontinuous functions and high dimensional data. The MDACNN architecture can generalise well on continuous and discontinuous functions because the relationship complexities between those functions are comparable simple and, therefore, easy to learn for the small architecture. The MDACNN architecture performs well on high dimensional data due to the tabularized input data and its kernel in the convolutional layer, which spans over all feature dimensions and considers, therefore, per convolution step, all features at the same time. Namely, the affected benchmarks are: [1)] continuous functions with linear relation, [2)] discontinuous function with linear relationship, [3)] continuous function with non-linear relationship and [7)] 50-dimensional function.

2. The MDACNN generalises badly or does not generalise on oscillating functions. A reason for this can be that the oscillating character of the functions increases the complexity of their relationships with each other, which complicates the generalisation on them for the small MDACNN architecture. Affected are the benchmarks: [4)] continuous oscillating function with non-linear relationship, [5)] phase-shifted oscillation and [6)] different periodicity.

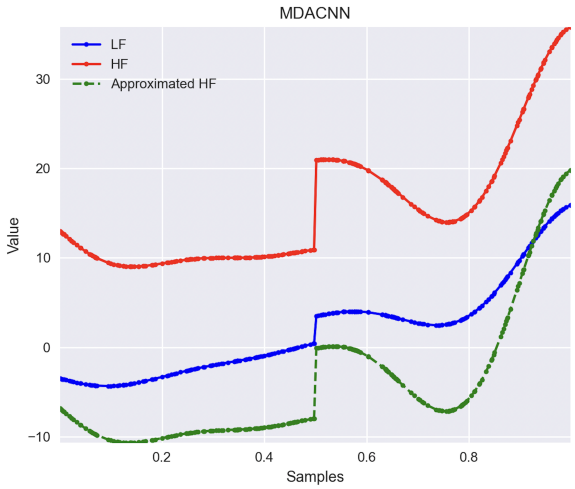
The benchmark with the most simple relationship is [1)] continuous functions with linear relation. As visible in Fig. B.1b, the benchmark result of the first implementation has a huge bias. The bias can be removed by making the training dataset bigger by increasing the number of HF samples from the recommended 4 HF samples (see appendix, Fig. 4.2) to a minimum of 15 HF samples. With more data, the model does not underfit, learns the pattern and the bias is gone, as shown in Fig. B.1c.

The next benchmark is [2)] discontinuous function with a linear relationship. Again, the MDACNN architecture has not generalised well in the benchmark of the first implementation and a huge bias forms shown in Fig. 4.1b. To remove the bias, it was necessary to normalise the input data. By normalising the input data, all input features get re-scaled to a common scale. This way, all features matter the same when training and the training is more efficient. The more efficient training eliminates the bias, as can be seen in Fig. 4.1c.

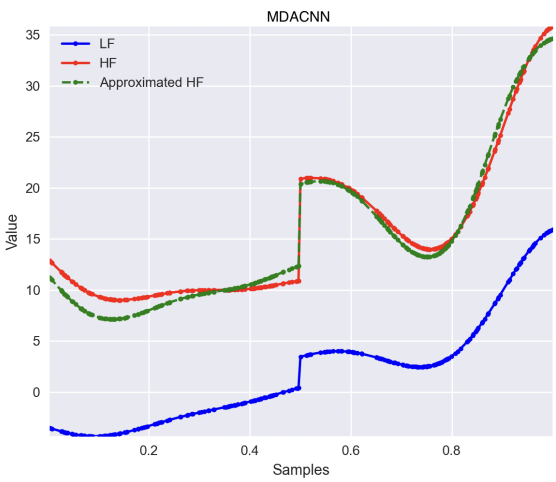
The benchmark [3)] continuous function with non-linear relationship is one of the best examples, where the benchmark of the first implementation achieved a comparably low bias and good generalisation. Fig. B.2b visualises only a small bias from X=0.0 until X=0.55. Like in the



(a) Benchmark provided by [2]

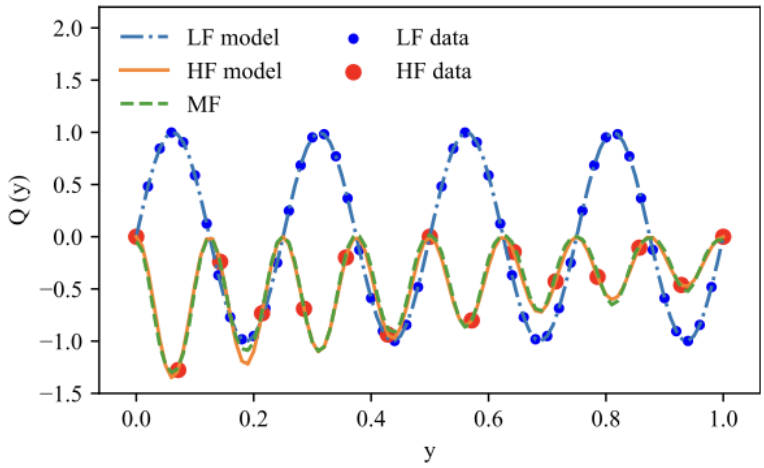


(b) Benchmark after implementing the MDACNN model and utilise the API and MDACNN architecture described in [2].

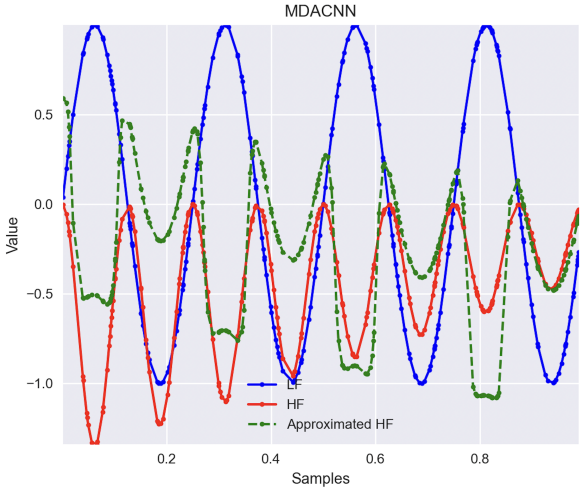


(c) Benchmark after improving the data pre-processing of the API by normalizing the input data.

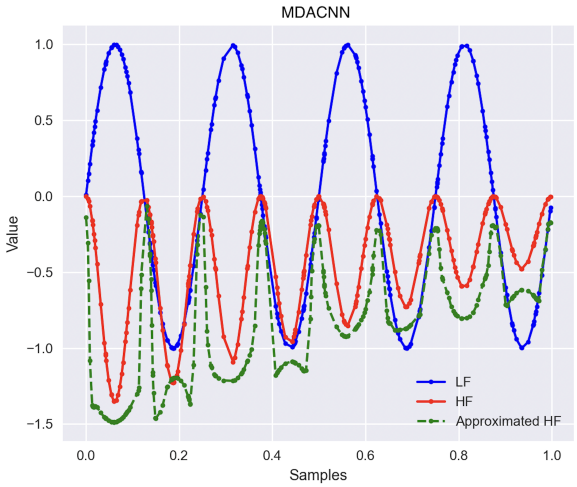
Figure 4.1.: Benchmark: [2)] discontinuous functions with linear relationship. This benchmark represents benchmarks with non-oscillating LF and HF functions.



(a) Benchmark provided by [2]



(b) Benchmark after implementing the MDACNN model and utilise the API and MDACNN architecture described in [2].



(c) Benchmark after improving the generalisation ability of the MDACNN architecture by extending the non-linear branch in the MDACNN architecture and adding more neurons and layers.

Figure 4.2.: Benchmark: [4] Continuous oscillation functions with nonlinear relationship. This benchmark represents all benchmarks with oscillating LF and HF functions.

prior benchmark, it is possible to remove the bias by normalising the input data, enabling the model to train using more features and making the training more efficient. The improved generalisation is visible in Fig. B.2c. The first benchmark where the model does not generalise well (even if improved) is [4] Continuous oscillation functions with non-linear relationship. Regularisation, an increase in training data or normalisation of the input data did not help to overcome the bad learning performance from Fig. 4.2b. By extending the non-linear branch of the MDACNN with more neurons and layers, it was possible to improve the generalisation

visible in Fig. 4.2c. The relationship between the LF and HF functions is non-linear. By extending the non-linear branch of the MDACNN architecture with additional neurons and layers, the MDACNN architecture becomes more effective in learning non-linear relationships and has better performance. Although the result got better, the generalisation is still not good (compared to prior benchmarks). The approximated HF function does not match at any point the actual HF function. The overshoot of the approximated HF function (MDACNN architecture model) has just been reduced.
The benchmark [5] phase-shifted oscillations, like its predecessor, again examines the relationship between oscillating functions with a non-linear relationship. The non-linear branch needs again the same extension with more neurons and layers like in the prior benchmark. The performance of the first implementation in Fig. B.3b improved to the performance shown in Fig. B.3c with a lower bias, and the HF and approximated HF function are closer together, but the result is not satisfactory even after the improvement.
Like the two previous benchmarks, the current benchmark -[6] different periodicity - also works with oscillating functions, as shown in Fig. B.4b. As in the two previous benchmarks, regularisation, an increase in the training data or normalisation of the input data do not help here either. Here too, the extension of the non-linear branch is necessary to slightly improve the generalisation capability of the MDACNN architecture model, as shown in Fig. B.4c. And again, the generalisation is not satisfactory, e.g. due to massive overshoots and noisy peaks.
The last benchmark is [7] 50-dimensional functions. The goal is to check whether the MDACNN architecture can process high-dimensional data (here: 50 dimensions) and whether a big input data dimensionality has an impact on the accuracy of the model. As visible in Fig. **??**, the results are very good. The MDACNN architecture is capable of processing high-dimensional data. This is due to the convolution kernel, which processes all feature dimensions of all covered samples within a position at once.

4.1.2. Performance Optimisation based on terramechanical Data

The benchmarks pre-define LF-, HF-function and the interval range of sample value x to ensure a common standard for datasets if the benchmark is executed by different research groups for different research projects (on the topic of multi-fidelity modelling). In the prior section, the generalisation performance of the MDACNN was tested - and, if needed, improved. Now, we shift our focus from the default benchmarks to an application-oriented dataset provided by the DLR [3][4]. The challenge is that the LF/HF relation of the terramechanical data is much more complex than the relationships provided by the benchmarks. Each benchmark focuses on testing a certain LF/HF relation with a function which is as simple as possible to be able to draw the wanted relation (to make the test case not unnecessarily complex). The terramechanical data can be more difficult to process for the MDACNN architecture because the LF/HF relation in the data can be more complex due to the combination of many individual relations defined by the default benchmark in Tab. 4.1.
Due to the increased complexity when switching from the default benchmarks to the terramechanical data, the performance optimisation of the MDACNN architecture is divided into parts. In the first step, the ability of the MDACNN architecture to be generally able to

Table 4.3.: Learning performance of different MDACNN architecture versions. The test was done by propagating the train dataset as test dataset through the trained MDACNN model. The following two versions of the MDACNN architecture were checked: the original MDACNN architecture, visualised in Fig. 3.3, and the 3x32 MDACNN architecture, visualised in the appendix in Fig. C.3. The 3x32 MDACNN architecture has an extended non-linear branch with additional neurons and layers. The lowest loss value for each iteration, the lowest average loss value and the lowest minimum loss value are printed in bold.

Iteration	**Original**	**3x32**
1	**0.05286**	0.05396
2	0.05323	**0.03713**
3	**0.03209**	0.03834
4	**0.03541**	0.03966
5	0.04503	**0.03920**
6	0.05538	**0.04222**
7	**0.03899**	0.04355
8	0.05995	**0.05727**
9	0.06277	**0.04704**
10	0.04889	**0.02987**
Average Loss	0.04846	**0.04282**
Minimum Loss	0.03209	**0.02987**

learn the training data will be secured. The general ability to learn the train dataset gets tested by training the model with the train dataset and later on, propagating through the trained model the used train dataset. If the architecture can learn the train dataset, then it will predict all train samples correctly. If the architecture will not be able to predict the train samples correctly, then optimization methods need to be applied to increase the performance of the architecture. The second step aims to check whether the model can generalise well. This is the most important trait which the model needs to possess - at least after optimisation. The model needs to be able to learn the pattern of the LF/HF relation in the terramechanical train dataset and recognise those patterns again later on in the corresponding test dataset.
Independently of which step gets performed: if the original MDACNN architecture is not capable of learning the train dataset or is not capable of generalising well, then the model architecture needs to be adjusted. Adjusting the architecture has the goal of extracting more features by increasing the learning efficiency of the network. A feature in images can be a line or its position to another line in the image. But what is a feature or a sub-feature if the data is tabularized like the terramechanical data is? Features equal the individual columns in the input table. However, after passing through the network layers, a feature can describe a value distribution between different columns in the input table or the relation between the occurrences of the two value distributions at two different kernel positions. Out of those extracted features, does the MDACNN architecture predict the HF value of interest.

Table 4.4.: Generalization performance of different MDACNN architecture versions. All versions utilised the same training and test set. The following five versions of the MDACNN architecture were checked: the **Original** MDACNN (Figure 3.3), **3x32** MDACNN (Figure C.3), **5x32** MDACNN, **7x32** MDACNN (Figure C.5), **regularised 7x32** MDACNN (Figure 3.5) and the **9x32** MDACNN architecture. All architectures except the original MDACNN architecture possess an extended non-linear branch with, e.g., 9 FC layers with 32 non-linear neurons each in the case of the 9x32 MDACNN architecture. The lowest loss value for each iteration, the lowest average loss value and the lowest minimum loss value are printed in bold.

Iteration	**Original**	**3x32**	**5x32**	**7x32**	**regularised 7x32**	**9x32**
1	0.22993	0.18552	0.32667	**0.11121**	0.17362	0.24567
2	0.25434	0.20435	0.25000	**0.19931**	0.35256	0.22700
3	0.30194	0.15534	0.70030	**0.14216**	0.14317	0.17309
4	0.16265	0.18596	0.12657	0.15847	**0.12521**	0.15668
5	0.16294	0.15874	0.13114	0.17826	**0.10971**	0.18347
6	0.28296	0.25577	**0.15383**	0.22139	0.23594	0.23592
7	0.16455	0.25872	0.14790	0.15687	**0.12208**	0.17752
8	0.31542	0.50171	0.13886	0.15448	0.22634	**0.13303**
9	0.19315	0.25448	0.14083	0.13628	**0.09335**	0.11902
10	0.21297	0.17083	0.16380	**0.12965**	0.14063	0.18302
Average Loss	0.22909	0.23315	0.22799	**0.15905**	0.17226	0.18345
Minimum Loss	0.16265	0.15534	0.12657	0.11121	**0.09335**	0.11902

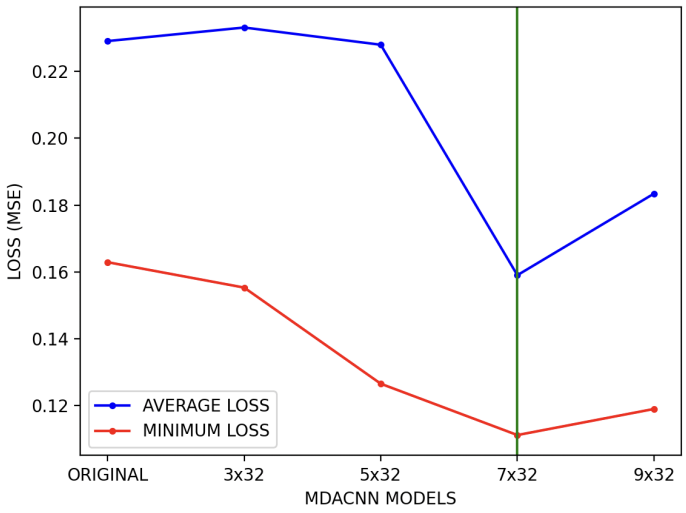
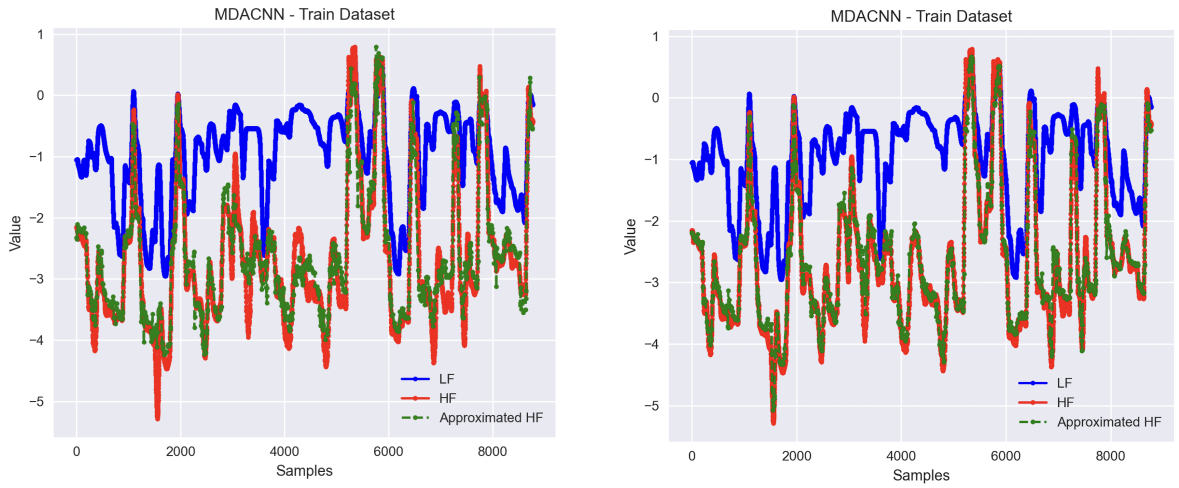


Figure 4.3.: Distribution of the average loss and minimum loss over the different MDACNN architecture versions, which are benchmarked in Table 4.4. The green line marks the global optimum (lowest losses) reached by the 7x32 MDACNN model.

Common ways to adjust the architecture to improve the performance of the model (enable the model to learn more complex LF/HF relation with higher precision) are by adding layers, neurons and regularisation. The more layers a network possesses, the more sub-features a model can learn. And the more sub-features a model extracts, the more data it has on which basis it can do good regression (the MDACNN is a regression model). This is at least the core thought behind adding more layers. If the model does not perform very well, it can have too few layers. A layer always learns the sub-features of the feature learnt by the prior layer. Each neuron learns a feature. The more neurons a layer possesses the more features it can learn. Therefore, the addition of more neurons to the model can increase its regression performance. The last step is implementing regularization. Regularization is less focused on installing more infrastructure (more neurons, more layers) to learn more features. Regularisation aims to make the learning of the already existing infrastructure more efficient to obtain even better results. During the experiments generally were first added neurons and layers and only then was regularization implemented to fine-tune the model.
MF models have a limitation in their complexity and amount of parameters. MF models like the MDACNN architecture have the mission to approximate the HF function value as accurately as possible while being computationally cheap - in construction, training and later during execution. In short, being as precise as an HF model but as computationally cheap as an LF model. Therefore, there cannot be added an infinite amount of neurons and layers to the MDACNN architecture to improve its performance - the complexity of the model needs to stay at a reasonable scale to not lose the original intended use of the MDACNN architecture as an MF model. Another reason why adding more and more neurons (and layers) to the model is not always the best idea is because, yes - each added neuron learns a new feature - but not each feature learnt will be used later on by the model to develop its regression output at the end. However, these unused neurons still need to be trained and increase unnecessarily the precious computational cost. Therefore, the goal during the performance optimisation is to obtain a result as good as possible while keeping the model architecture in a reasonable and appropriate size.
During the experiments, several versions of the original MDACNN architecture were designed, trained and tested. The loss is used to evaluate the performance of the models because the original MDACNN architecture is a regression model. On the one hand, describes the loss the numerical difference between the predicted value and the ground truth value and is therefore suited to evaluate regression models. On the other hand, accuracy (the other big evaluation method) captures whether the predicted value is equal to the ground truth or not and is, therefore, better suited to evaluate classification models. Independently from the evaluation, by default, the loss function gets utilised as the objective function during training.
Before skipping to the presentation of the results, another short disclaimer: the goal of the performance optimisation is to optimise the performance of the MDACNN architecture regarding the terramechanical dataset provided by the DLR. The terramechanical dataset is already pre-split into a train and test dataset. Therefore, the priority is to increase the performance of the MDACNN architecture regarding exactly this data split. The top goal is to create a model which extracts as much information as possible out of the provided train



(a) Original MDACNN architecture by Chen et al. [2].

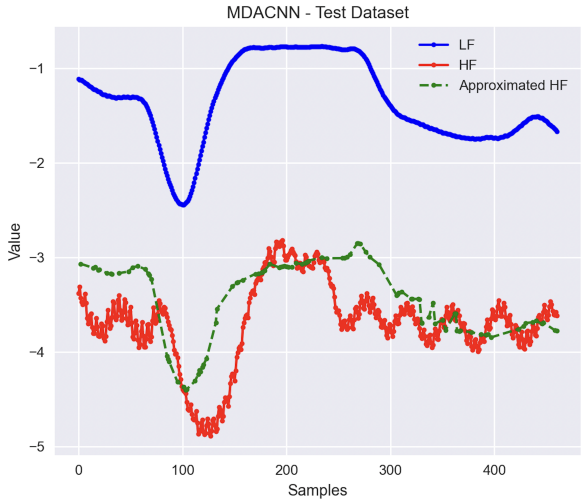
(b) 3x32 MDACNN architecture.

Figure 4.4.: First steps in adapting the MDACNN to the provided terramechanical data. Visualised are the learning performances of the Original MDACNN and the 3x32 MDACNN architecture. The goal is to ensure the MDACNN architectures possess enough parameters to learn the train dataset. In both cases, the train dataset got propagated through the already trained model. The test result is that the 3x32 MDACNN architecture, with its extended non-linear branch, performs better than the original MDACNN architecture.

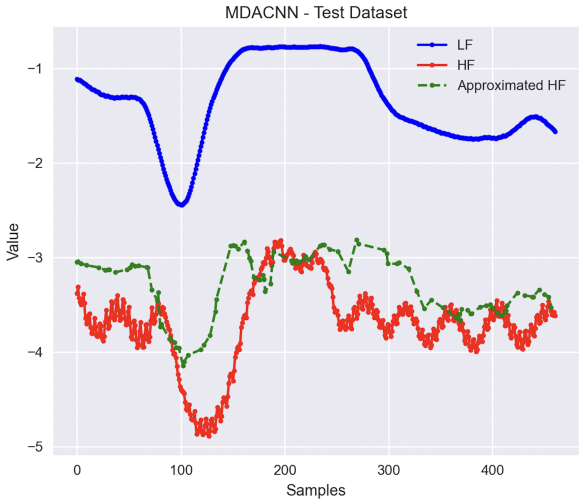
dataset (generalise well on the LF/HF relation) and apply as much knowledge as possible to the test dataset. K-Fold Cross-Validation will be used later on to check how the model performs for different data splits. As already mentioned, the loss will be used to evaluate the performance of the different MDACNN architecture versions. A problem during training is that the same architecture can be trained with the same training and test dataset several times, and each time, the trained model performs a little bit differently. But to evaluate and benchmark different versions of the MDACNN model with each other, it is necessary to characterise each model with an average performance (described by loss). This problem was solved using the law of big numbers: to approximate the average performance of a model architecture, several randomly sampled model training runs were taken, and their average performance (average loss) was computed. Important is to randomly sample those trained models and their losses because this ensures that the resulting approximation is unbiased and, therefore, more accurate. E.g. to approximate the performance of an architecture, each architecture was trained ten times - as a result, there were ten different losses, and out of these ten different losses, the average performance of the architecture was approximated.

Starting with the first optimisation step, checking whether the original MDACNN architecture is capable of learning the terramechanical train dataset provided by the DLR. The result can be seen in Figure 4.4a. The blue line is the LF function, red is the HF function, and green is the approximated HF function (by the MDACNN). The lines of interest are the

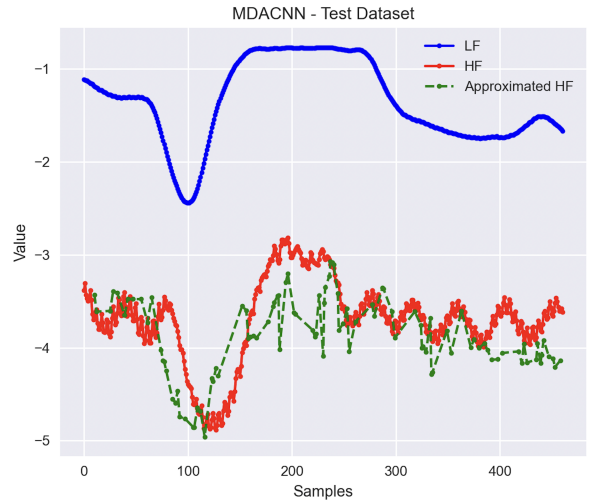
red and the green lines because the HF function and the approximated HF function should be equal in an ideal case. In reality, the original MDACNN architecture can match the red and green lines with each other most of the time, but a careful observer will realise that the original MDACNN architecture has problems, especially when learning peak values. Not all peaks, but many peaks are not correctly interpretable for the original MDACNN architecture - although the trained model processed known training data. A reason could be that the original MDACNN architecture possesses too few parameters and is not able to learn the training data. The problem could be solved by adding more neurons and layers. It needs to be considered that the LF/HF relationship between the blue and red graph is highly non-linear. Therefore, the extra neurons and layers must be added to the non-linear branch of the MDACNN. The architecture of the original MDACNN architecture is depicted in Figure 3.3 and in the appendix with more detail in Figure C.1 and with a focus on parameters in Figure C.2. The experiments revealed that there exists a small sweet spot between the number of parameters and accuracy in learning the training dataset for the MDACNN network and within this sweet spot lies the 3x32 MDACNN architecture. The 3x32 MDACNN architecture is a version of the original MDACNN architecture where in the non-linear branch the two FC layers with each ten non-linear neurons got replaced with three FC layers with each non-linear 32 neurons. The improvement in results is shown in Table 4.3. The difference is not big, but the 3x32 MDACNN architecture delivers a better average loss and better models with a lower minimum loss than the original MDACNN architecture. The 3x32 MDACNN architecture is depicted in the appendix with a focus on its architecture in Figure C.3 and with focus on its parameters in Figure C.4. As visible in Figure 4.4b does the 3x32 MDACNN architecture learn the training data better than the original MDACNN architecture. The green and red lines are better matching and most of all peaks were recognized by this MDACNN architecture version. In the second and last optimization step, the goal is to improve the generalisation ability of the MDACNN architecture regarding the provided terramechanical train and test dataset. In Figure 4.4a and Figure 4.4b is visible how the original MDACNN architecture and the 3x32 MDACNN architecture do perform on the test dataset. Again, the blue line is the LF function, red is the HF function, and green is the approximated HF function (by the MDACNN model). Both MDACNN architectures do not perform well. Both architectures possess a big bias. The bias is a sign of underfitting, which means they do not learn enough features from the training data. Generally, in both cases, the green line follows the red line, but only in a rough manner. Due to the underfitting of the original MDACNN architecture and the 3x32 MDACNN architecture, the experiments aimed to add neurons and layers and achieve this way better results. And because the LF/HF relation is non-linear those neurons and layers got added to the non-linear branch of the MDACNN. The optimisation process was done in phases: first, add neurons and layers to let the model learn more features (act against overfitting) second, find the best-performing architecture, and third, add regularisation for fine-tuning to make the learning process of the existing architecture more effective. Table 4.4 shows the progression of the most important experiments, their order, and with which results the experiments were done. The experiments started from the 3x32 MDACNN architecture because it has better learning capabilities regarding the train dataset than the original



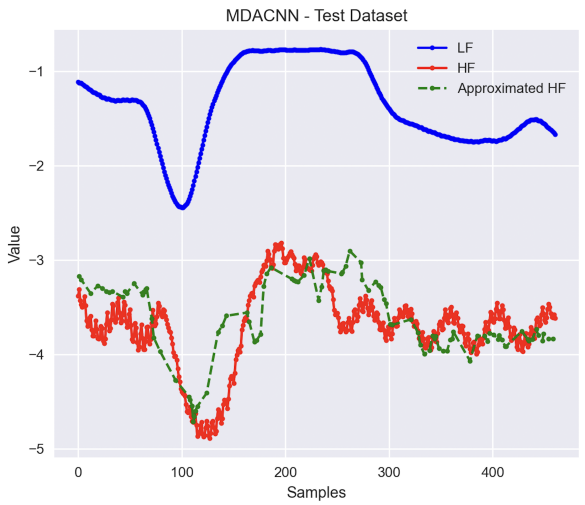
(a) Original MDACNN. The non-linear branch consists out of two layers, each containing ten neurons.



(b) 3x32 MDACNN. The non-linear branch is build out of three layers with each 32 neurons.



(c) 7x32 MDACNN. The original non-linear branch got replaced with seven layers, each containing 32 neurons.



(d) Regularized 7x32 MDACNN.

Figure 4.5.: Benchmarking four different MDACNN versions regarding their generalization abilities. All models got trained with the same training and testing dataset to secure that all models had to learn the same LF/HF relationship.

MDACNN architecture (see Table 4.3). The original MDACNN, 3x32 MDACNN on the 5x32 MDACNN, 7x32 MDACNN and the 9x32 MDACNN architecture were tested. In Figure 4.3, it is visible how the average loss and the minimum loss (best-trained model) develop. The average loss remains almost constant for the original MDACNN, 3x32 MDACNN and the 5x32 MDACNN architecture. This means that these MDACNN architecture versions produce models which possess, after training, on average, the same performance. Starting from the original MDACNN architecture, it can be generally said that the more layers are established, the better the optimal model, and the lower the optimal loss (...until the 9x32 MDACNN architecture). The absolute lowest average loss and minimum loss were achieved by the 7x32 MDACNN architecture during the experiments. After the 7x32 MDACNN architecture, both features go up for the 9x32 MDACNN architecture - therefore, the regularisation got implemented for the 7x32 MDACNN architecture. The 7x32 MDACNN architecture is visualised in the appendix with a focus on its architecture in Figure C.5 and with a focus on its parameters in Figure C.6. The generalisation performance of the 7x32 MDACNN architecture is visualised in Figure 4.5c. The 7x32 MDACNN has a way lower bias than the original MDACNN and 3x32 MDACNN architecture. And the 7x32 MDACNN architecture follows visually more the red HF function than both others do. But the output is pretty noisy - e.g., there are huge peaks from $X = [1.500, 2.500]$ (label x-label "Samples" as X). One goal of the regularisation is, therefore, to get rid of these spikes. The regularised 7x32 MDACNN architecture is depicted in Figure 3.5 and in the appendix in Figure C.7 and Figure C.8. The regularisation methods used are batch normalisation, average pooling and dropout. Batch normalisation and average pooling are used after the convolution layer, and dropout is used firstly after the flattening of the convolution results and secondly inside the non-linear branch. The experiments showed that the batch normalisation, in combination with the average pooling, helps to learn the valley-like curve from $X = [750, 2.000]$ unbiased and the dropout immediately after flattening helps to learn the hill-like curve from $X = [1.500, 2.500]$. The dropout inside the non-linear branch just smooths (de-noise) the learned LF/HF relation. The result is that the model has, according to Table 4.4, a higher average loss than the original 7x32 MDACNN architecture, but the minimum loss (optimal trained model) could be further reduced to a global minimum if compared to all other MDACNN architecture versions. The generalisation performance of the regularised 7x32 MDACNN architecture is depicted in Figure 4.5d.

The 10-fold cross-validation is used as a final performance check to investigate the generalisation performance of the most important MDACNN architecture versions on different data splits into train and test dataset than the one predefined and provided by the terramechanical data. 10-fold cross-validation was used because it is common practice to split a dataset into 90 & train dataset and 10 % test dataset. In 10-fold cross-validation, different average losses are calculated for the same architecture after each run. To approximate the overall average loss for each MDACNN architecture version, 10-fold cross-validation was therefore applied ten times to each architecture according to the previously applied law of large numbers (LLN). The most important MDACNN architectures are the original MDACNN, 3x32 MDACNN, the 7x32 MDACNN and the regularised 7x32 MDACNN architecture. These architectures are benchmarked with each other in a final comparison in Table 4.6. The original MDACNN

Table 4.5.: Final verification of the most important MDACNN architecture versions and their generalisation performance by applying 10-fold cross-validation on them. The 10-fold cross-validation was executed ten times for each architecture. The **Original** MDACNN, **3x32** MDACNN, **7x32** MDACNN and the **regularised 7x32** MDACNN architecture were tested. The aim is to check whether the architectures not only generalise well on the predefined data split of the terramechanical data into a train and a test dataset but also whether they can handle other data splits. The lowest loss value for each iteration, the lowest average loss value and the lowest minimum loss value are printed in bold.

Iteration	**Original**	**3x32**	**7x32**	**regularised 7x32**
1	0.00480	0.00361	0.00478	**0.00342**
2	0.00403	0.00429	0.00515	**0.00299**
3	0.00452	0.00361	0.00538	**0.00284**
4	0.00420	0.00411	0.00489	**0.00246**
5	0.00477	0.00375	0.00555	**0.00314**
6	0.00499	0.00656	0.00571	**0.00317**
7	0.00470	0.00423	0.00597	**0.00315**
8	0.00396	0.00411	0.00589	**0.00291**
9	0.00523	0.00515	0.00512	**0.00263**
10	0.00419	0.00395	0.00478	**0.00255**
Average Loss	0.004539	0.00434	0.00532	**0.00293**
Minimum Loss	0.00396	0.00361	0.00478	**0.00246**

architecture [2] is the initial one from where all the research started. The 3x32 MDACNN architecture is the smallest architecture which learns the provided terramechanical train dataset better than the original MDACNN architecture, see Figure 4.4. The 7x32 MDACNN architecture was the best generalising, non-regularised architecture in Table 4.4 and the regularised 7x32 MDACNN architecture was in Table 4.4 generally the best-performing architecture on the provided terramechanical data. The results in Table 4.5 are surprising: like in Table 4.4 outperforms the 3x32 MDACNN architecture the original MDACNN architecture. A reason could be, that independent from the data split, regarding the terramechanical data, generalises the 3x32 MDACNN architecture better the the original one. A big surprise was the 7x32 MDACNN architecture. Although it outperformed all other non-regularised MDACNN architecture versions in Table 4.4, it performed the worst in Table 4.5. An explanation is, that the 7x32 MDACNN architecture is especially well suited for generalising on the particular data split which is predefined in the provided terramechanical data. But generally has the 7x32 MDACNN architecture bigger problems in generalising on the terramechanical data. The best-performing architecture in Table 4.4 is also the best-performing one in Table 4.5. The regularised 7x32 MDACNN architecture outperforms all other architectures. This leads to the conclusion that, regardless of the data split, the regularised 7x32 MDACNN architecture generalises better than all other MDACNN architecture variants.

Table 4.6.: Summarised comparison of the **Original** MFCNN-TL, **3x32** MFCNN-TL, **7x32** MFCNN-TL and **regularized 7x32** MFCNN-TL architectures with each other. The architectures get benchmarked on training data, prediction time, number of parameters and their average loss. Mind: the 7x32 MDACNN architecture has a lower average loss than the regularised 7x32 MDACNN architecture, but the regularised 7x32 MDACNN architecture generates the best-performing models with minimal loss (Table 4.4) and performs better on different data splits (Table 4.5). For both architectures, the same train and test dataset was used. The training time refers to the biggest time needed to process an epoch during training. The prediction time refers to processing the whole test dataset. The lowest value for each feature is printed in bold.

	Original	**3x32**	**7x32**	**reg. 7x32**
Training Time	**14 s**	15 s	26 s	38 s
Prediction Time	**77 ms**	70 ms	1 s	889 ms
Number of Parameters	**141.062**	414.852	419.076	148.996
Average Loss	0.22909	0.23315	**0.15905**	0.17226

4.2. Experiments on the MFCNN-TL architecture

In this section, the MFCNN-TL architecture is investigated. The MFCNN-TL architecture belongs to the TLNN architecture type, where an NN-based MF model gets trained utilising transfer learning. As part of investigations regarding the MFCNN-TL architecture, all developed MFCNN-TL architecture versions and their experimentally obtained generalisation performances are noted in Table 4.7. The performance of each architecture is measured using the average loss and the minimum loss. Each architecture model had different loss values after the same training with the same train dataset. This is problematic when the goal is to retrieve the loss of an architecture. Therefore, each MFCNN-TL architecture version was trained 10 times to approximate the actual mean loss of the architecture using the LLN. Important for the correct application of the LLN is that all trained architecture models with their corresponding loss values are sampled randomly. The same method was already used for the MDACNN architecture in the prior section. All architectures get trained, utilising the terramechanical image data provided by the DLR [3][4].
The original MFCNN-TL architecture [9] was the first architecture to be tested in Table 4.7. The original MFCNN-TL architecture is shown in Figure 3.6 and closer described in the appendix in Figure D.1 and Figure D.2. The original MFCNN-TL architecture has an approximated average loss of 0.03556 and a minimum loss of 0.03247. These loss values need to be beaten by alternative MFCNN-TL architecture versions.
The 1x8+2x24 MFCNN-TL architecture is the second architecture investigated in Table 4.7. The architecture is shown and closer described in Figure 3.8. The 1x8+2x24 MFCNN-TL architecture focuses on shape feature extraction. Shape extraction focuses on the extraction of the shape of an object from a given image. The shape of an object is normally characterised

Table 4.7.: Generalization performance of different MFCNN-TL architecture versions. Investigated are the **Original** MFCNN-TL (Figure 3.6), **1x8+2x24** MFCNN-TL (Figure 3.8), **1x24** MFCNN-TL (Figure 3.9) and the **Perceptron** MFCNN-TL architecture (Figure 3.7). All models are described in the appendix in detail as well. The 1x8+2x24 MFCNN-TL network is the worst-performing architecture, and the Perceptron MFCNN-TL network is the best-performing architecture. The lowest loss value for each iteration, average loss, and minimum loss value are printed in bold.

Iteration	Original	1x8+2x24	1x24	Perceptron
1	0.03452	0.03830	0.02498	**0.01469**
2	0.03642	0.03634	0.02256	**0.01077**
3	0.03519	0.03740	0.02195	**0.01658**
4	0.03615	0.03717	0.02346	**0.01306**
5	0.03247	0.03770	0.02272	**0.01519**
6	0.03567	0.03860	0.02341	**0.01333**
7	0.03698	0.03954	0.02287	**0.01135**
8	0.03537	0.03786	0.02508	**0.01559**
9	0.03590	0.03638	0.02272	**0.01405**
10	0.03684	0.03859	0.02093	**0.01419**
Average Loss	0.03556	0.03786	0.02278	**0.01387**
Minimum Loss	0.03247	0.03634	0.02093	**0.01077**

by lines. Therefore, shape extraction is mostly equal to line recognition. Shape feature extraction focuses on finding features of the extracted shape, like relative height, thickness, etc. The 1x8+2x24 MFCNN-TL architecture has an enlarged second convolutional block to enable an extended shape feature extraction. The investigations in Table 4.7 showed, that with an average loss of 0.03786 and minimum loss of 0.03634 performs the 1x8+2x24 MFCNN-TL architecture even worse than the original MFCNN-TL architecture. It can be concluded, that the shape feature extraction is not responsible for a high generalization performance of the MFCNN-TL architecture regarding the terramechanical image data. The 1x24 MFCNN-TL architecture is the third architecture which got tested. The architecture is shown and closer described in Figure 3.9. The main feature of the 1x24 MFCNN-TL architecture is the emphasised shape extraction. Therefore, the architecture has an enlarged first convolutional layer compared to the original MFCNN-TL architecture. Extract better the shape of the sand structures on the terramechanical images by using more kernels. The second convolutional block (used for shape feature extraction) and some FC layers were removed as well to emphasise the shape extraction. The breath thought behind the 1x24 MFCNN-TL architecture is that if the enlarged shape feature extraction of the 1x8+2x24 MFCNN-TL architecture has no positive effect on the performance, then the shape extraction must be the operation which influences the performance of the architecture. The investigations in Table 4.7 show an increasing generalisation performance of the 1x24 MFCNN-TL architecture with an average loss of 0.02278 and minimal loss of 0.02093. These are better loss values

than for both architectures tested before. The hypothesis that the shape extraction of the first convolutional block of the original MFCNN-TL architecture has a big influence on the performance was thus confirmed.
Out of the checked architectures is the Perceptron MFCNN-TL architecture in Table 4.7, the biggest surprise. The Perceptron MFCNN-TL architecture is shown in Figure 3.7 and closer described in the appendix in Figure D.7 and Figure D.8. The outstanding performance of

(a) Original MFCNN-TL architecture

(b) Perceptron MFCNN-TL architecture

Figure 4.6.: Visualizing the optimisation results. It is visible that the Perceptron MFCNN-TL architecture has a much higher generalisation performance than the original MFCNN-TL architecture.

the Perceptron MFCNN-TL architecture surprises because the input data are images, and CNNs are generally better suited and more effective to process images than perceptrons. The reason for that is that CNNs are mainly built out of convolutional layers and every single convolutional layer learns local patterns, applies the learned local pattern (kernel) all over the image and therefore reuses their parameters. Therefore, a convolutional layer needs by tendency fewer parameters than a comparable fully connected layer and a smaller training dataset. Perceptrons consist of FC layers, and FC layers are much less effective in processing images than convolutional layers. Fully connected layers do not learn local features (no kernels), but they need to learn to interpret the object explicitly at all possible positions in the image - this requires much more data and training effort than for CNNs. FC layers are also much more ineffective regarding parameters because they do not reuse weights as convolutional layers do. The Perceptron MFCNN-TL architecture is smaller than any other tested MFCNN-TL architecture version. The training dataset is huge. It can be assumed that the big training dataset possesses all possible sand structures and, for each sand structure, all possible positions in the image. Therefore, the Perceptron MFCNN-TL is enabled to effectively process images from the provided distribution (after being trained). The question remains, why does the Perceptron MFCNN-TL architecture (perceptron) perform better than the 1x24

MFCNN-TL architecture (CNN)? One possible answer is that the knowledge learned by the Perceptron MFCNN-TL architecture for each specific sand structure at each position has a higher information value than the generalised local pattern learned by the 1x24 MFCNN-TL architecture and leads, therefore, to a better prediction performance.

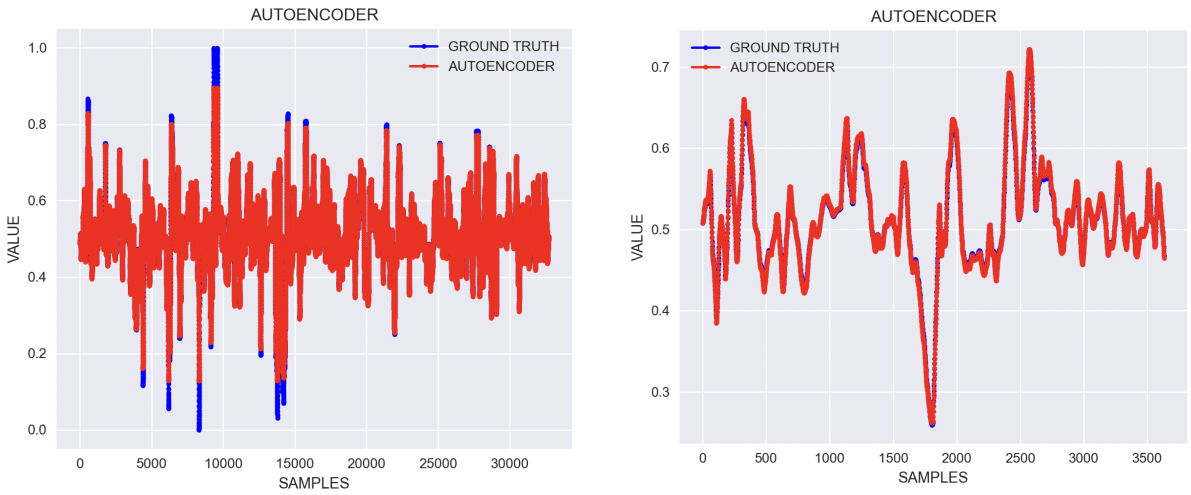
Table 4.8.: Summarised comparison of the **Original** MFCNN-TL, **1x8+2x24** MFCNN-TL, **1x24** MFCNN-TL and **Perceptron** MFCNN-TL architectures with each other. The architectures get benchmarked on training data, prediction time, number of parameters and their average loss. For both architectures, the same train and test dataset was used. The training time refers to the biggest time needed to process an epoch during training. The prediction time refers to processing the whole test dataset. The lowest value for each feature is printed in bold.

	Original	**1x8+2x24**	**1x24**	**Perceptron**
Training Time	24 s	78 s	134 s	**4 s**
Prediction Time	1 s	2 s	3 s	**85 ms**
Number of Parameters	1.079.496	1.139.973	91.449.585	**1.050.625**
Average Loss	0.03556	0.03786	0.02278	**0.01387**

Other advantages of the Perceptron architecture, next to the better performance, are highlighted in Table 4.8. The Perceptron MFCNN-TL architecture is lightweight compared to the other MFCNN-TL architectures, and its training is much faster than that of all other CNN architectures. Its performance is roughly twice as good as the best-performing CNN-based MFCNN-TL architecture (would be the 1x24 MFCNN-TL architecture). Fig. 4.6 benchmarks the generalisation performance of the original MFCNN-TL MFCNN-TL architecture with the Perceptron MFCNN-TL architecture - here gets visualised how much better the perceptron performs on the given dataset.

4.3. Experiments on the MF-TLNN architecture

The original MF-TLNN architecture was defined by Zhang et al. [10] and was introduced in 2024. The main idea behind the MF-TLNN architecture is to combine explicit and implicit learning in a single architecture to improve the learning efficiency and prediction performance of the HF values. Therefore, the MF-TLNN architecture is implemented by combining an implicit and explicit learning architecture in a composite network-like architecture by adding the outputs of both architectures in a weighted aggregation. Both combined architectures predict the same HF value for an input sample X. After aggregating both (pre-trained) architectures together, the MF-TLNN architecture fine-tunes its architecture weights using transfer learning. Therefore, it belongs to the group of TLNN architectures. In general, the construction process of the MF-TLNN architecture can be described as follows: First, an implicit and an explicit learning architecture are defined, and both architectures get pre-trained utilising LF data. Initially, before they are aggregated and fine-tuned, both architectures are pre-defined by learning the X/LF relationship (relationship between the



(a) Train Dataset

(b) Test Dataset

Figure 4.7.: Visualisation of the results of propagating the training and test data sets through the autoencoder (AE). Although the AE has not learnt the training dataset very well (the model has problems learning the peaks), the model performs well on the test dataset. The AE can recover the test patterns well.

input sample and its LF value). Second, create the MF-TLNN architecture by aggregating the outputs of both pre-trained architectures using a weighted aggregation. Third, fine-tune the MF-TLNN architecture with HF data, so that the network learns the X/HF (implicit) and LF/HF (explicit) relation. The combination of implicit and explicit learning in a single architecture is a novel approach because the MF-DF architecture type aims only at explicit learning and the TLNN architecture type only at implicit learning. The MF-DF architecture type has so far been represented by the MDACNN architecture [2], and the TLNN architecture type has so far been represented by the MFCNN-TL architecture [9]. The implemented MF-TLNN architecture is visualised in Figure 3.10 and in the appendix in Figure E.1 and Figure E.2. Zhang et al. [10] defined the explicit learning architecture to be an AE and the implicit learning architecture is simply described as an NN. Zhang et al. [10] did not define a general architecture to be used for neither the AE nor the NN. Therefore, first of all, both architectures got constructed based on the terramechanical data provided by the DLR [3][4]. The architecture of the AE gets closer described in Figure 3.10 and in the appendix in Figure E.1 and Figure E.2. The constructed AE is small, lightweight and has good performance as shown in Figure 4.7. Therefore, this AE architecture is used for all presented MF-TLNN architecture versions.

Tab. 4.9 shows the performance results for the two best-performing models. Due to its good performance, the same AE architecture was used in all investigated MF-TLNN architecture versions. As NN was chosen the best-performing MFCNN-TL architecture from the prior Section 4.2. This was done because the MF-TLNN architecture and the MFCNN-TL are well suited to process images and to perform transfer learning. The Perceptron MF-TLNN

Table 4.9.: Generalization performance of the **1x24** MF-TLNN architecture and the **Perceptron** MF-TLNN architecture. The original MF-TLNN architecture is visualised in Figure E.3. The 1x24 MF-TLNN architecture was realised by implementing the 1x24 MFCNN-TL architecture as NN_L in the original MF-TLNN architecture. The Perceptron MF-TLNN architecture was implemented by using the Perceptron MFCNN-TL architecture as NN_L in the original MF-TLNN architecture. The 1x24 MFCNN-TL and the Perceptron MFCNN-TL architectures are described in Section 4.2. The lowest loss value for each iteration, the lowest average loss value, the lowest standard deviation loss and the lowest minimum loss value are printed in bold.

Iteration	**1x24**	**Perceptron**
1	**0.00160**	0.00283
2	**0.00125**	0.01065
3	0.00615	**0.00562**
4	0.01521	**0.00491**
5	**0.00299**	0.00622
6	0.02013	**0.01720**
7	0.01165	**0.01108**
8	0.01835	**0.00207**
9	**0.01141**	0.01450
10	**0.00250**	0.00311
Average Loss	0.00912	**0.00782**
Standard Deviation Loss	0.00681	**0.00497**
Minimum Loss	**0.00125**	0.00207

architecture was the first MF-TLNN network to be implemented. The Perceptron MF-TLNN architecture uses the Perceptron MFCNN-TL architecture as its NN. The performance of the Perceptron MFCNN-TL architecture is described in the prior Section 4.2. The Perceptron MFCNN-TL architecture was chosen because it showed the best performance among all MFCNN-TL architecture versions for the provided terramechanical dataset. The second MF-TLNN architecture in Table 4.9 is the 1x24 MF-TLNN architecture. The 1x24 MFCNN-TL architecture was implemented in the prior Section 4.2. The performance of the 1x24 MFCNN-TL architecture is visualised in Table 4.7. The 1x24 MFCNN-TL architecture is the second best-performing MFCNN-TL architecture version and is therefore used in the 1x24 MF-TLNN architecture to benchmark the Perceptron MF-TLNN architecture. Both architectures - the Perceptron MFCNN-TL and the 1x24 MFCNN-TL architecture - get again compared with each other because the dataset for the MF-TLNN architectures is smaller than the one for the MFCNN-TL architectures - which can have an impact on the model performance, especially if using perceptrons for image processing. The datasets for the MFCNN-TL and MF-TLNN architecture come from the same distribution.

According to Tab. 4.9, does the Perceptron MF-TLNN architecture outperform the 1x24

Figure 4.8.: Propagation result of the 1x24 MF-TLNN architecture for test data. Generally speaking, the MF-TLNN architecture performs well - the predicted output function (red) matches the HF function (blue) nearly everywhere. The course of the predicted output function shows almost no influences of the LF function (green).

MF-TLNN architecture if comparing the average loss. The average loss is an approximation of the true (and unknown) average loss of the models, which was computed by using the LLN and randomly sampled loss values of different trained models of the same architecture. On the one hand, the Perceptron MF-TLNN architecture generates the best models on average; on the other hand, the 1x24 MF-TLNN architecture generates the absolute best models. The best model generated by the 1x24 MF-TLNN architecture has a minimum loss of 0.00125 (MSE), roughly twice as accurate as the best model generated by the Perceptron MF-TLNN architecture with a minimum loss of 0.00207 (MSE). In general, the 1x24 MF-TLNN architecture has the highest standard deviation and, therefore, less stable training results than the Perceptron MF-TLNN architecture. A possible cause could be the utilised dataset and its splitting into train and test dataset. The dataset for the MF-TLNN architecture is smaller than that for the MFCNN-TL architecture. Both architectures are MF networks, but - e.g. for training and testing purposes - the MFCNN-TL architecture requires its data samples to have either an LF or an HF value, while the MF-TLNN architectures require all data samples to have an LF and an HF value. A CNN like the 1x24 MFCNN-TL architecture learns patterns and needs less data to learn those to be capable of processing images than a perceptron. If a perceptron is to be used for image processing, it cannot learn a generalised local pattern but must explicitly learn all possible object positions for each object in the image so that the perceptron - ineffective learning. A smaller dataset can bring the risk, that not all positions are covered. This could explain why the 1x24 MF-TLNN architecture sometimes outperforms the Perceptron MF-TLNN architecture that heavily. However, it does not explain why the

1x24 MF-TLNN architecture has the biggest standard deviation, whereas the Perceptron MF-TLNN architecture is more stable. One possible reason for this could be that the data set is randomly split into a training and a test dataset at the beginning of each training and test run of an architecture model (90 % training, 10 % test). Due to the different splits for each training, there may be data splits that are beneficial for learning patterns. A typical property for a good split would be that all patterns in the test dataset are also included in the training dataset. And there may be data splits that are only beneficial for learning one object and all its positions on the image - these splits are especially beneficial for the Perceptron MF-TLNN architecture and less beneficial for the 1x24 MF-TLNN architecture. Taking this into account, it can be concluded that the Perceptron MF-TLNN architecture is less sensitive to different data splits than the 1x24 MF-TLNN architecture due to the lower standard deviation. The Perceptron MF-TLNN and the 1x24 MF-TLNN are benchmarked in a final step with each other in Table 4.10.

Table 4.10.: Summarised comparison of the **1x24** MF-TLNN and **Perceptron** MF-TLNN architectures with each other. The architectures get benchmarked on training data, prediction time, number of parameters and their average loss. The same AE was used for both versions of the MF-TLNN architecture. The training time refers exclusively to the time required to calculate the NN_L network, as the training time for all other networks is the same for both MT-TLNN architecture versions. For both architectures, the same train and test dataset was used. The training time refers to the biggest time needed to process an epoch during training. The prediction time refers to processing the whole test dataset. The lowest value for each feature is printed in bold.

	1x24	**Perceptron**
Training Time	134 s	**4 s**
Prediction Time	63ms	**30 ms**
Number of Parameters	274.348.358	**1.050.652**
Average Loss	0.00912	**0.00782**

5. Conclusion and Future Work

5.1. Conclusion

Mathematical models represent the optimal mathematical solution process to solve a problem. Numerical models are approximations of mathematical models applying theoretical mathematical solutions in practice. HF models are numerical models which approximate better and have a higher accuracy and computational cost. LF models are the other kind of numerical models which approximate worse and get, therefore, lower accuracy and computational cost. There is a trade-off between accuracy and computational cost. The goal of MF modelling is to beat this trade-off and to create high-accuracy, low-cost models. MF models typically learn the relation between the LF and HF values (learn the relation between the LF and HF function) out of pre-given LF and HF predictions. There are two main ways how an MF model can learn the relation between the LF and HF function: implicitly and explicitly. If learning implicitly the MF model just gets the sample X alone and then it needs to compute the corresponding HF value. An explicit learning MF model gets as input the LF value (and the sample X) to compute directly out of the LF value the HF value. MF modelling can be applied to classification and regression tasks. This master thesis utilised MF modelling for regression tasks to approximate the HF value as a scalar out of a continuous interval. Typical regression methods are linear, logistic regression and Co-Kriging for easy relations between LF and HF. If the relation gets more complicated, regression methods like Gaussian Processes and NN are used. But of course can Gaussian processes and NN be applied to linear relations as well. NN are well suited to learning non-linearities like those between many LF and HF functions. However, due to the nature of MF modelling, there are also borders set: MF models must be small and lightweight. A MF model must be accurate and computationally cheap. The goal of this Master Thesis is to investigate how well NN can be used for MF modelling if learning complex relations between the LF and HF function and keeping the network architecture as small and lightweighted as possible. The typical bottleneck regarding NN and MF modelling is that an NN needs more neurons and a bigger architecture, the more complicated the LF/HF relations are. The Master Thesis investigates NN with a special focus on CNNs.
Three different architectures were investigated in this Master Thesis: the explicit learning MDACNN, the implicit learning MFCNN-TL and the MF-TLNN architecture which learns the relation between the LF and HF function implicitly and explicitly. The MDACNN and MFCNN-TL architecture represent the two mainstream architecture types (MF-DF and TLNN architecture type) regarding MF learning implemented using NN while the MF-TLNN architecture is a novel proposal by [10] to improve the mainstream methods. The MF-TLNN architecture is leaning towards the TLNN architecture type.

The first main architecture type discussed is the MF-DF architecture type. The original MF-DF architecture, as visualised in Figure 2.8, is a single network which learns the relation between LF and HF implicitly. Research proved that the original implicitly learning MF-DF architecture is less performing than comparable explicitly learning architectures. Explicit learning MF-DF architectures are called multi-level MF-DFs because these architectures are designed as composite NNs where each sub-NN learns explicitly only a single relation (each single relation is a level). The most common multi-level MF-DF architectures are the 2-level MF-DF (Figure 2.10) and the 3-level MF-DF (Figure 2.9). Chen et al. [2] built based on the 2-level MF-DF a CNN whose architecture can be described as the design of the second and last model (in the 2-level MF-DF architecture) with an upstream convolutional layer. This novel architecture is called MDACNN. The advantage of the MDACNN architecture and its convolutional layer against conventional 2-level MF-DF architectures is that the MDACNN architecture can use the whole input dataset (all samples with all their LF values) to predict for a single sample X the HF value. Figure 3.3 shows the MDACNN architecture and how the tabular input data gets forwarded through the convolutional layer. The MDACNN architecture and its performance were tested in two separate experiment runs: using default benchmarks and the custom data provided by the DLR. To check its general performance, the MDACNN architecture was first checked using the default benchmarks for MF modelling shown in Table 4.1. The benchmarks showed that the model tends to perform better for bigger datasets and that the normalisation must be done. Benchmarks also revealed that the architecture can learn most of the common relationship types between two functions (LF and HF function), but the hardest relations to learn for the architecture are those between oscillating functions shown in Figure 4.2, Figure B.3 and Figure B.4. The experiments on the custom data provided by the DRL show, that the MF-DF architecture (here: the MDACNN architecture) can be used for MF modelling. The only criterion is that the MDACNN architecture needs to be adjusted to each new dataset depending on the relation and the complexity of the relation between the LF and HF functions it provides. In an overwhelming amount of cases, the datasets differ in the non-linear complexity of their LF/HF relation. In the case of the provided data by the DLR, the relation between LF and HF was more complex than those from the default benchmarks. To prevent the inability to learn features and, therefore, underfitting, there is the need to increase the number of neurons and layers in the non-linear branch of the MDACNN architecture shown in Table 4.4. In general, it can be concluded from the investigation results that the MDACNN architecture is suited for MF modelling.
The second architecture which got investigated is the MFCNN-TL architecture by Liao et al. [9]. The MFCNN-TL architecture implicitly learns the relation behind the LF and HF functions. The main idea behind the MFCNN-TL architecture is to apply transfer learning to a CNN to enable MF modelling. In transfer learning, a model gets pre-trained with source data, and later on, the unfrozen parts of the same model get fine-tuned with the task data. If transfer learning is applied to MF modelling, then the source data gets defined through the LF dataset, and the task dataset is the HF dataset. The MFCNN-TL architecture is a CNN, and CNNs consist of two main blocks: a convolutional block and a fully connected block. The convolutional block extracts features out of the input images, and the fully connected

block finds the relations among the extracted features and regresses out of them the output. The MFCNN-TL architecture possesses two convolutional layers. The first layer is supposed to catch the shape of the sand structure in the images, and the second layer is there to extract the shape features (thickness, relative height, ...). To increase the performance of the MFCNN-TL architecture, different architecture versions with emphasis on different features were tested as shown in Table 4.7. The architectures which emphasise the shape extraction (first convolutional layer) were the best-performing. The least performing were those which emphasised the shape feature extraction (second convolutional layer). The best-performing CNN possesses only one convolutional layer and extracts only the shape without extracting any further shape features. But the overall best-performing, which outperformed all CNN architectures, was a perceptron. The perceptron gets obtained by taking the best-performing CNN architecture (Figure 3.9) and removing the only convolutional layer so that only the average pooling, a dense and the output layer remain as visualised in Figure 3.7. A possible explanation for the good performance of a perceptron in image processing, as shown in Figure 4.6, could be the low amount of parameters and the high amount of training data. Overall, the MFCNN-TL architecture is suited for MF modelling - in the form of a CNN or perceptron.
The last remaining architecture type is the MF-TLNN architecture by Zhang et al. [10]. The architecture combines the implicit and explicit learning of the LF and HF function to increase the performance of the MF modelling. The combination of both learning strategies in one model gets implemented by building a composite NN out of a implicit learning and an explicit learning model and using their outputs to compute the actual HF value of a sample using weighted aggregation depicted in Figure 3.10 and Figure E.3. Due to the composite character of the architecture, the training procedure of a single MF-TLNN architecture happens in three steps: in the first step, both models (implicit and explicit learning sub-networks) get separately pre-trained. In the second step, both models get combined via their outputs using the weighted aggregation. In the third and last step, the whole MF-TLNN architecture gets fine-tuned using transfer learning. Zhang et al. [10] predefined the explicit learning model to be an AE - they just defined that the model needs to be an autoencoder but did not define its architecture. For the investigations got chosen a small and light-weighted autoencoder (see Figure E.1) which has a good performance like shown in Figure 4.7b. As implicit learning model got chosen the best-performing MFCNN-TL architecture. Two different MF-TLNN architecture versions with two different MFCNN-TL architecture versions - the Perceptron MFCNN-TL and the 1x24 MFCNN-TL (see Table 4.7) - were used to be able to benchmark. Out of the good regression performances of both MF-TLNN architecture versions can be concluded that the MF-TLNN architecture as well is also suited to be used for MF modelling.
In this Master Thesis, three different model architectures were investigated. Although the models receive different input data - the MD-DF architecture takes tables (matrices), the MFCNN-TL and the MF-TLNN architecture use images - all data come from the same origin source. Therefore, the performances of the models are comparable to each other. If comparing the model architectures with each other using the average loss of their best-performing models it is noticeable that all three models have losses of different magnitudes. The highest loss has

Table 5.1.: Average losses of the best-performing model for each architecture type. The best-performing MF-DF model was the regularised 7x32 (see Tab. 4.4), the best MFCNN-TL model was the 1x24 CNN (see Tab. 4.7) and the MF-TLNN model with the lowest average loss was the Perceptron MF-TLNN (see Tab. 4.9). The lowest average loss value is printed in bold.

	MF-DF	**MFCNN-TL**	**MF-TLNN**
Lowest Average Loss	0.17226	0.01387	**0.00782**

the MF-DF and the lowest loss has the MF-TLNN architecture as shown in Table 5.1. The very good performance of the MF-TLNN architecture can be a result of effective training due to the combined learning types. The following conclusions can be drawn from the results, although all three architecture types are well suited for MF modelling, the MF-TLNN architecture is the best-performing among them all.
Based on these final conclusions, the hypothesis set out in the introduction to the master's thesis can be definitively answered:

1. **The MF models using CNNs will outperform the MF models using perceptrons in terms of the terramechanical data provided.** Not generally true. A large, given dataset enabled the perceptron-based Perceptron MFCNN-TL architecture to outperform the best CNN-based 1x24 MFCNN-TL architecture, see Table 4.7. The same happened to the MF-TLNN architecture, but here it needs to be stated that the Perceptron MF-TLNN architecture was on average the best-performing architecture while the CNN-based 1x24 MF-TLNN architecture generated the overall best-performing models, see Table 4.9.

2. **The MF-TLNN architecture will have a higher learning performance than the two other architectures.** True. The investigations showed that a combination of explicit and implicit learning in the same architecture leads to more efficient learning and higher performance. A performance comparison between all investigated architectures is shown in Table 5.1 - the MF-TLNN architecture outperforms the other architectures.

5.2. Future Work

There are two issues, which would be worth to be investigated in future work: one is regarding the MFCNN-TL and the other the MF-TLNN.
The first issue does not address the original MFCNN-TL architecture by Liao et al. [9] but the results of the performance optimisation in Table 4.7. As a surprise, not a CNN but a perceptron (Figure D.7) is the best-performing architecture. It is surprising because CNNs are, in general, better suited to process images than perceptrons. CNNs learn local patterns and reuse their weights, which reduces the number of parameters per network. Perceptrons do not learn patterns and do not reuse their weights. Therefore, they need to learn explicitly for each object for all possible positions in the image, how the object will look like and how this needs to be interpreted to be effective in image processing. The most likely cause for the

outstanding performance of the perceptron architecture is, therefore, the big train dataset and that the dataset contains for all objects all possible locations so that the trained perceptron has no blind spots for each object and is due to this effective. Future work can investigate whether the perceptron architecture is always the best-performing architecture and whether there are sizes of the train dataset where the CNNs start to outperform the perceptron. Another important aspect of these investigations would be to clarify how important the size of the train dataset is for the perceptron to outperform its opponents.
The second possible construction site for future research regards the MF-TLNN architecture. According to the original design by Zhang et al. [10] is the composite-network-like architecture built out of two individual networks like depicted in Figure E.3. The NNL network learns the LF/HF relation implicitly, and the AE learns the LF/HF relation explicitly. In the paper is the NNL not clearly defined as an architecture type, but [10] defines the explicit learning network concrete as an AE. In future work, it can be investigated whether the performance of the MF-TLNN architecture can be improved by using other explicit learning architectures - like the MDACNN model by Chen et al. [2] (an MF-DF architecture). The goal would be to increase the performance of the MF-TLNN even further and make the MF-TLNN architecture even more attractive for MF modelling.

Bibliography

[1] K. Ravi, V. Fediukov, F. Dietrich, T. Neckel, F. Buse, M. Bergmann, and H.-J. Bungartz. "Multi-fidelity Gaussian process surrogate modeling for regression problems in physics". In: *Machine Learning: Science and Technology* 5 (Oct. 2024). DOI: 10.1088/2632-2153/ad7ad5.

[2] J. Chen, Y. Gao, and Y. Liu. "Multi-fidelity data aggregation using convolutional neural networks". In: *Computer methods in applied mechanics and engineering* 391 (2022), p. 114490.

[3] F. Buse. "Development and Validation of a Deformable Soft Soil Contact Model for Dynamic Rover Simulations". PhD thesis. Tohoku University, 2022.

[4] F. Buse. "Fully automated single wheel testing with the DLR Terramechanics Robotics Locomotion Lab (TROLL)". In: *15th Symposium on Advanced Space Technologies in Robotics and Automation, Noordwijk, Netherlands. ESA*. 2019.

[5] M. Guo, A. Manzoni, M. Amendt, P. Conti, and J. S. Hesthaven. "Multi-fidelity regression using artificial neural networks: Efficient approximation of parameter-dependent output quantities". In: *Computer Methods in Applied Mechanics and Engineering* 389 (2022), p. 114378. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2021.114378. URL: https://www.sciencedirect.com/science/article/pii/S0045782521006411.

[6] X. Meng and G. E. Karniadakis. "A composite neural network that learns from multi-fidelity data: Application to function approximation and inverse PDE problems". In: *Journal of Computational Physics* 401 (2020), p. 109020. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2019.109020. URL: https://www.sciencedirect.com/science/article/pii/S0021999119307260.

[7] D. Liu and Y. Wang. "Multi-fidelity physics-constrained neural network and its application in materials modeling". In: *Journal of Mechanical Design* 141.12 (2019). Cited by: 133. DOI: 10.1115/1.4044400. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85084267213&doi=10.1115%2f1.4044400&partnerID=40&md5=a1aedd1b51dc21ac22b13876b1d7a640.

[8] M. Motamed. "A multi-fidelity neural network surrogate sampling method for uncertainty quantification". In: *International Journal for Uncertainty Quantification* 10.4 (2020).

[9] P. Liao, W. Song, P. Du, and H. Zhao. "Multi-fidelity convolutional neural network surrogate model for aerodynamic optimization based on transfer learning". In: *Physics of Fluids* 33.12 (2021).

[10] Z. Zhang, Q. Ye, D. Yang, N. Wang, and G. Meng. "A multi-fidelity transfer learning strategy based on multi-channel fusion". In: *Journal of Computational Physics* 506 (2024), p. 112952. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2024.112952. URL: https://www.sciencedirect.com/science/article/pii/S0021999124002018.

[11] P. N. Stuart Russel. *Artificial Intelligence: A Modern Approach*. Pearson Education Limited, 2022.

[12] T. S. of Machine Learning AI. *K-Means Clustering*. https://www.ml-science.com/k-means-clustering [Accessed: (31.07.2024)]. 2023.

[13] HMKCODE. *Backpropagation Step by Step*. https://hmkcode.com/ai/backpropagation-step-by-step/ [Accessed: (01.08.2024)]. 2019.

[14] M. Reza Keyvanpour and M. B. Shirzad. "Chapter 14 - Machine learning techniques for agricultural image recognition". In: *Application of Machine Learning in Agriculture*. Ed. by M. A. Khan, R. Khan, and M. A. Ansari. Academic Press, 2022, pp. 283–305. ISBN: 978-0-323-90550-3. DOI: https://doi.org/10.1016/B978-0-323-90550-3.00011-4. URL: https://www.sciencedirect.com/science/article/pii/B9780323905503000114.

[15] L. ELSTER. *Convolutional Network What is a convolutional neural network (CNN) and how does it work?* https://mriquestions.com/convolutional-network.html [Accessed: (21.07.2024)]. 2023.

[16] B. Peherstorfer, K. Willcox, and M. Gunzburger. "Survey of Multifidelity Methods in Uncertainty Propagation, Inference, and Optimization". In: *SIAM Review* 60.3 (2018), pp. 550–591. DOI: 10.1137/16M1082469. eprint: https://doi.org/10.1137/16M1082469. URL: https://doi.org/10.1137/16M1082469.

[17] A. H. Abdelaziz, S. Watanabe, J. R. Hershey, E. Vincent, and D. Kolossa. "Uncertainty propagation through deep neural networks". In: *Interspeech 2015*. 2015.

[18] A. Tarantola and B. Valette. "Inverse problems = Quest for information". In: *Journal of Geophysics* 50.1 (1981), pp. 159–170. URL: https://journal.geophysicsjournal.com/JofG/article/view/28.

[19] A. Tarantola. *Inverse problem theory and methods for model parameter estimation*. SIAM, 2005.

[20] J. Kaipio and E. Somersalo. *Statistical and computational inverse problems*. Vol. 160. Springer Science & Business Media, 2006.

[21] A. M. Stuart. "Inverse problems: a Bayesian perspective". In: *Acta numerica* 19 (2010), pp. 451–559.

[22] J. A. Christen and C. Fox. "Markov chain Monte Carlo using an approximation". In: *Journal of Computational and Graphical statistics* 14.4 (2005), pp. 795–810.

[23] Y. Efendiev, T. Hou, and W. Luo. "Preconditioning Markov chain Monte Carlo simulations using coarse-scale models". In: *SIAM Journal on Scientific Computing* 28.2 (2006), pp. 776–803.

[24] T. Cui, C. Fox, and M. O'sullivan. "Bayesian calibration of a large-scale geothermal reservoir model by a new adaptive delayed acceptance Metropolis Hastings algorithm". In: *Water Resources Research* 47.10 (2011).

[25] T. Cui, Y. M. Marzouk, and K. E. Willcox. "Data-driven model reduction for the Bayesian solution of inverse problems". In: *International Journal for Numerical Methods in Engineering* 102.5 (2015), pp. 966–990.

[26] A. J. Booker, J. E. Dennis, P. D. Frank, D. B. Serafini, V. Torczon, and M. W. Trosset. "A rigorous framework for optimization of expensive functions by surrogates". In: *Structural optimization* 17 (1999), pp. 1–13.

[27] A. Keane. "Wing optimization using design of experiment, response surface, and data fusion methods". In: *Journal of aircraft* 40.4 (2003), pp. 741–750.

[28] A. I. Forrester and A. J. Keane. "Recent advances in surrogate-based optimization". In: *Progress in aerospace sciences* 45.1-3 (2009), pp. 50–79.

[29] A. I. Forrester, A. Sóbester, and A. J. Keane. "Multi-fidelity optimization via surrogate modelling". In: *Proceedings of the royal society a: mathematical, physical and engineering sciences* 463.2088 (2007), pp. 3251–3269.

[30] N. M. Alexandrov, J. E. Dennis Jr, R. M. Lewis, and V. Torczon. "A trust-region framework for managing the use of approximation models in optimization". In: *Structural optimization* 15.1 (1998), pp. 16–23.

[31] N. M. Alexandrov, R. M. Lewis, C. R. Gumbert, L. L. Green, and P. A. Newman. "Approximation and model management in aerodynamic optimization with variable-fidelity models". In: *Journal of Aircraft* 38.6 (2001), pp. 1093–1101.

[32] A. March and K. Willcox. "Provably convergent multifidelity optimization algorithm not requiring high-fidelity derivatives". In: *AIAA journal* 50.5 (2012), pp. 1079–1089.

[33] N. V. Queipo, R. T. Haftka, W. Shyy, T. Goel, R. Vaidyanathan, and P. K. Tucker. "Surrogate-based analysis and optimization". In: *Progress in aerospace sciences* 41.1 (2005), pp. 1–28.

[34] M. Eldred, A. Giunta, and S. Collis. "Second-order corrections for surrogate-based optimization with model hierarchies". In: *10th AIAA/ISSMO multidisciplinary analysis and optimization conference*. 2004, p. 4457.

[35] M. C. Kennedy and A. O'Hagan. "Predicting the output from a complex computer code when fast approximations are available". In: *Biometrika* 87.1 (2000), pp. 1–13.

[36] K. Carlberg. "Adaptive h-refinement for reduced-order models". In: *International Journal for Numerical Methods in Engineering* 102.5 (2015), pp. 1192–1210.

[37] Z.-H. Han, S. Görtz, and R. Zimmermann. "Improving variable-fidelity surrogate modeling via gradient-enhanced kriging and a generalized hybrid bridge function". In: *Aerospace Science and technology* 25.1 (2013), pp. 177–189.

[38] S. J. Pan and Q. Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (2010), pp. 1345–1359. DOI: 10.1109/TKDE.2009.191.

[39] F. Chollet et al. "Keras: The python deep learning library". In: *Astrophysics source code library* (2018), ascl–1806.

[40] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

A. Evaluating Convolutional Neural Networks in Multi-Fidelity Modeling

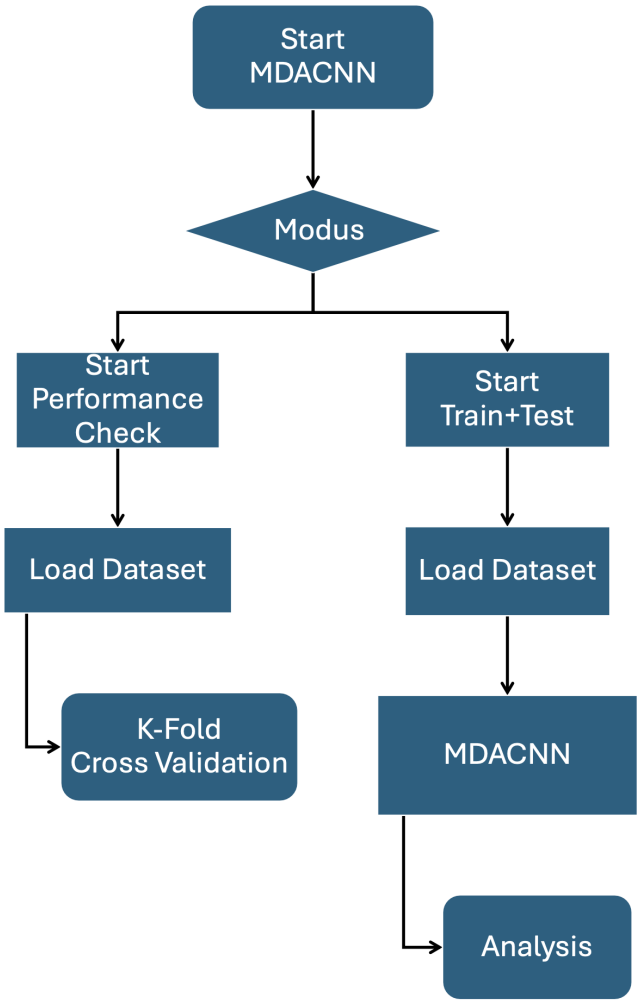


Figure A.1.: Implemented API for the MDACNN architecture. Two main processes are available: "Train+Test", where the default train and test dataset gets applied and the model gets trained and tested, or "Performance Check", where a 10-fold cross-validation gets performed.

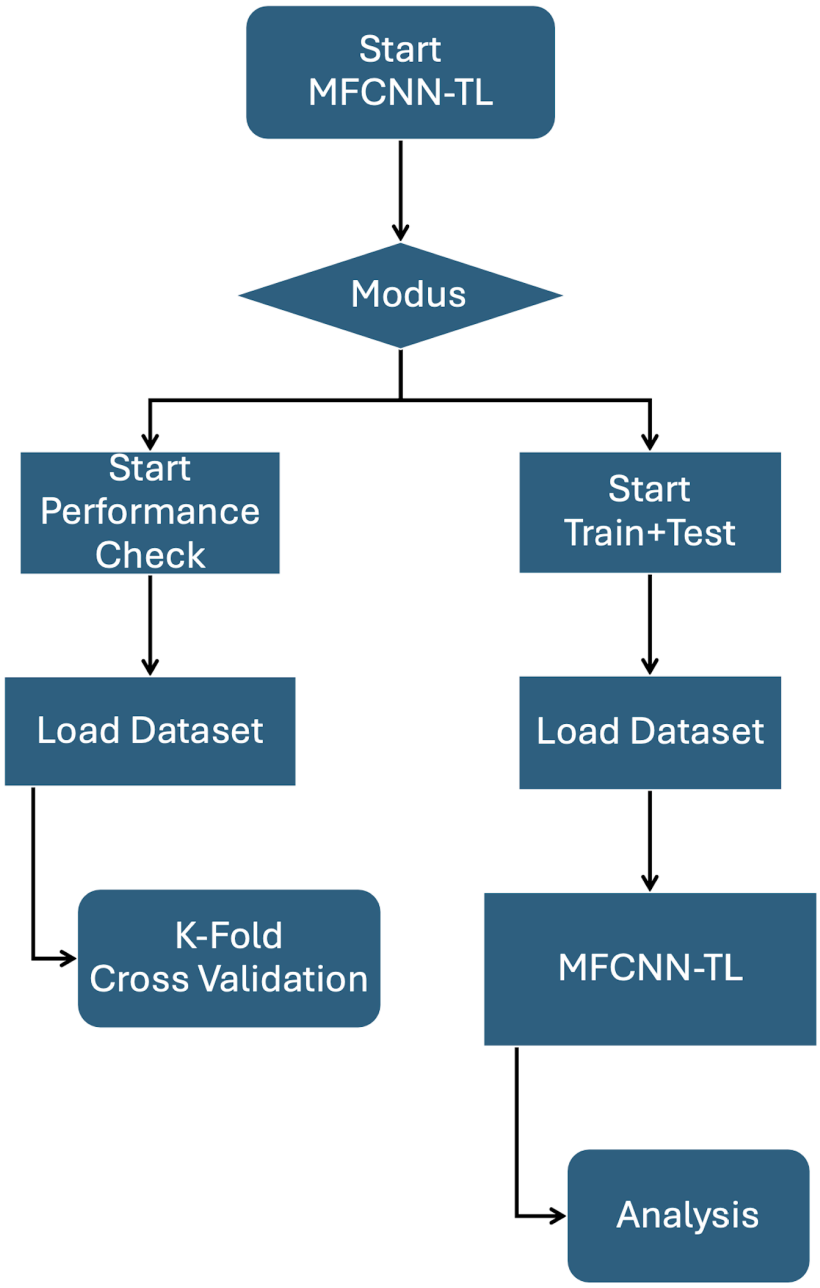


Figure A.2.: Implemented API for the MFCNN-TL architecture. As for the MDACNN architecture, two main processes are available: "Train+Test", where the default train and test dataset gets applied and the model gets trained and tested, or "Performance Check", where 10-fold cross-validation gets performed.

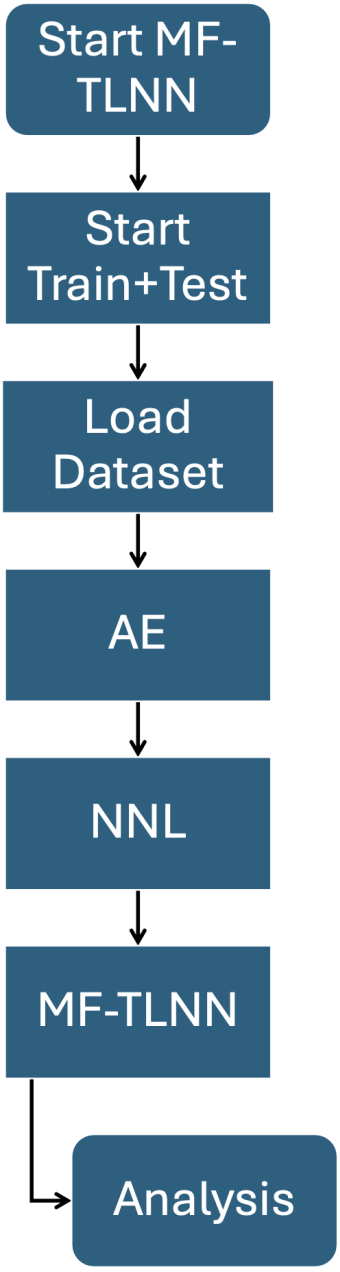


Figure A.3.: Implemented API for the MF-TLNN architecture. The "Train+Test" process looks as follows: first, the dataset gets loaded and processed. With the prepared the AE, and the NNL get initialized and trained. After those, the MF-TLNN gets initialized and trained via transfer learning.

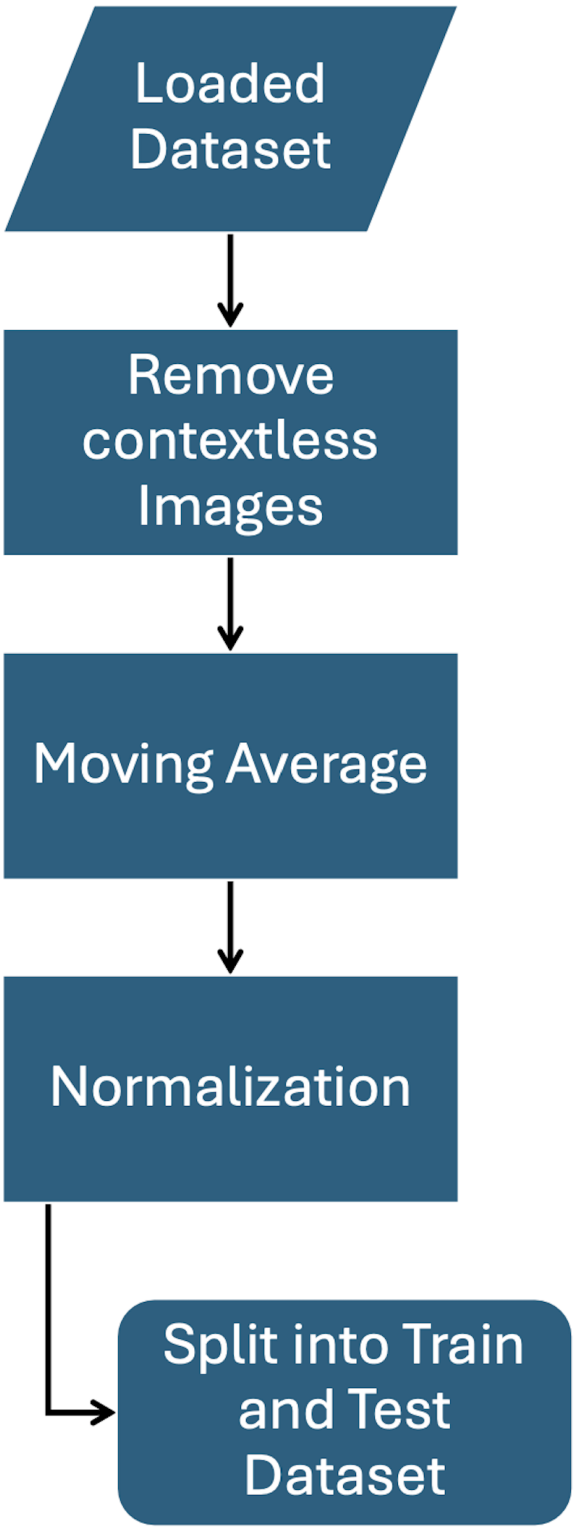
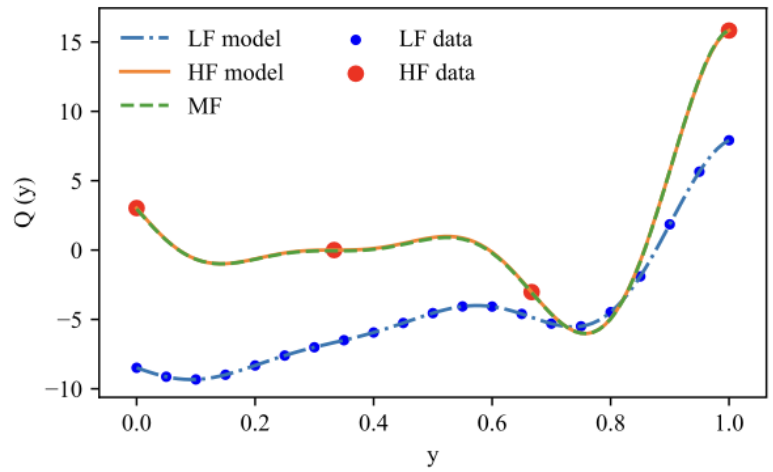


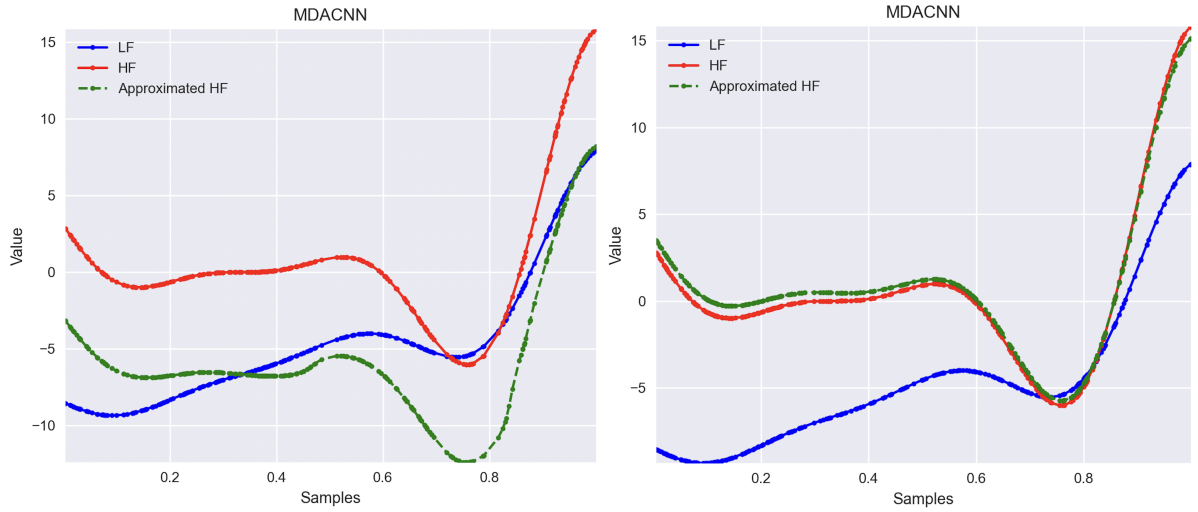
Figure A.4.: Implemented API for extended image preprocessing in the MFCNN-TL and MFTLNN architecture APIs. The API is part of the process "Load Dataset" in the Fig. A.2 and Fig. A.3.

B. Experiments on the MDACNN architecture

Chen et al. [2] defined the MDACNN as a representative of the MF-DF architecture. Chen et al. [2] defined the network architecture with all its parameters and provided benchmarks to prove that the designed MDACNN can learn all the information in the training dataset.



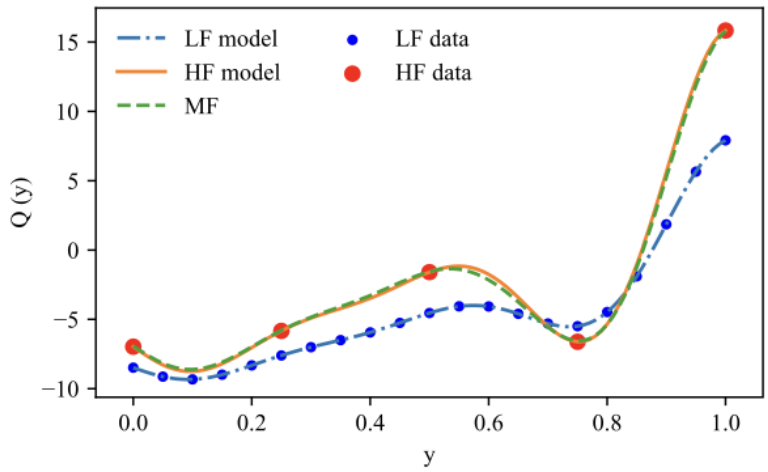
(a) Benchmark provided by [2].



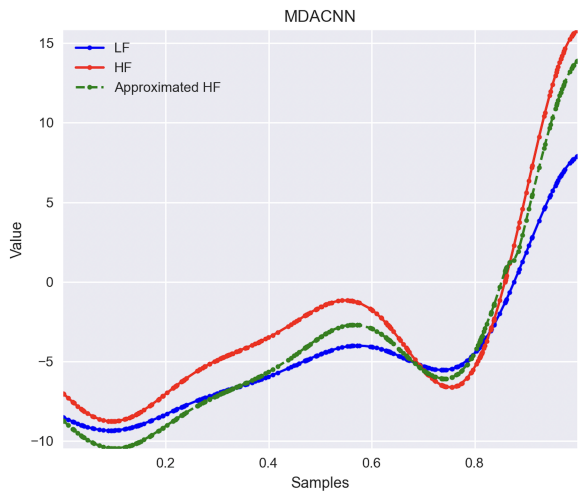
(b) Benchmark after implementing the MDACNN model and utilise the API and MDACNN architecture described in [2].

(c) Benchmark after improving the benchmark by adding more training data.

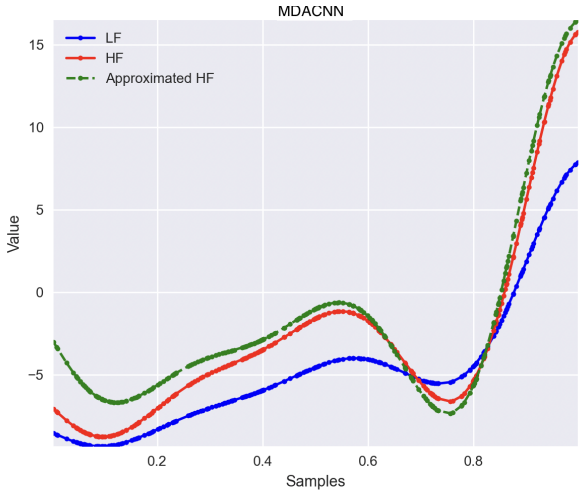
Figure B.1.: Benchmark: [1)] continuous functions with linear relationship.



(a) Benchmark provided by [2].

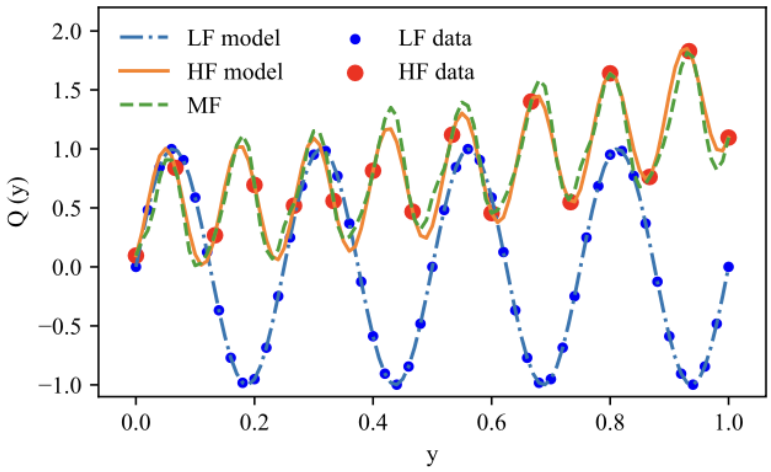


(b) Benchmark after implementing the MDACNN model and utilise the API and MDACNN architecture described in [2].

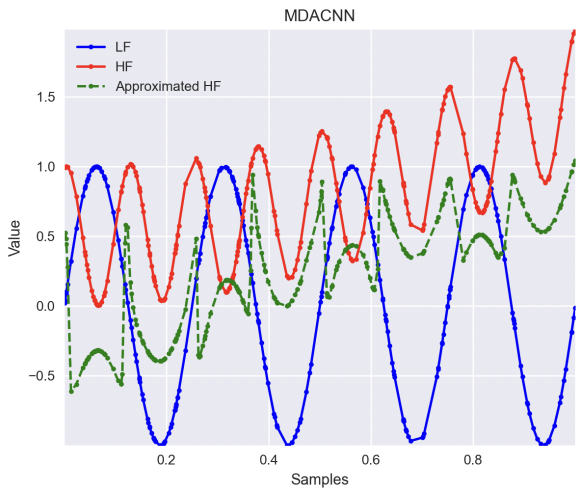


(c) Benchmark after improving the data pre-processing of the API by normalizing the input data.

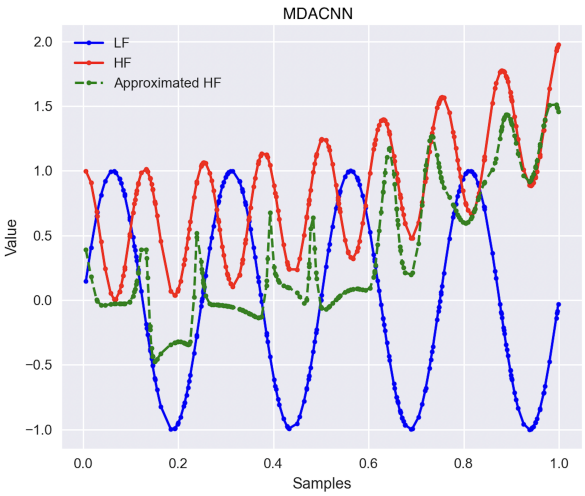
Figure B.2.: Benchmark: [3] continuous functions with nonlinear relationship.



(a) Benchmark provided by [2].

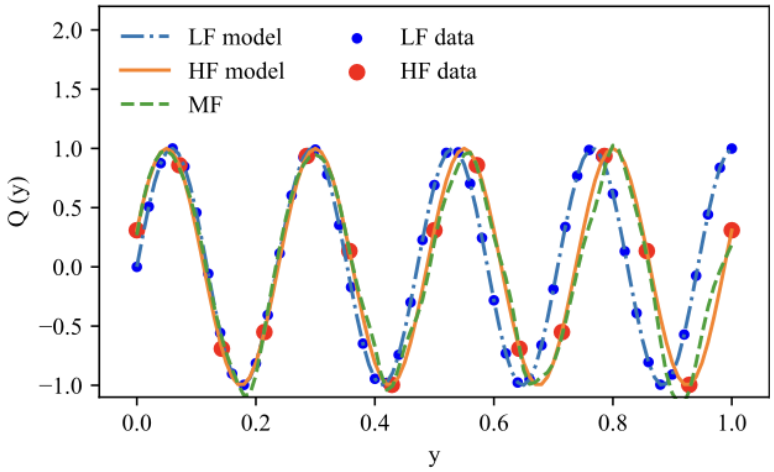


(b) Benchmark after implementing the MDACNN model and utilise the API and MDACNN architecture described in [2].

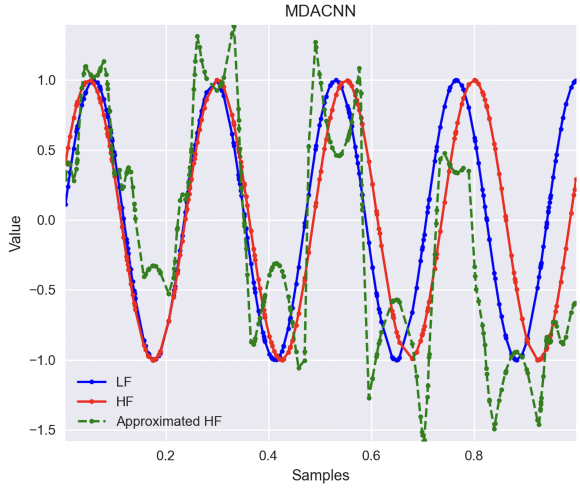


(c) Benchmark after improving the generalisation ability of the MDACNN architecture by extending the non-linear branch in the MDACNN architecture and adding more neurons and layers.

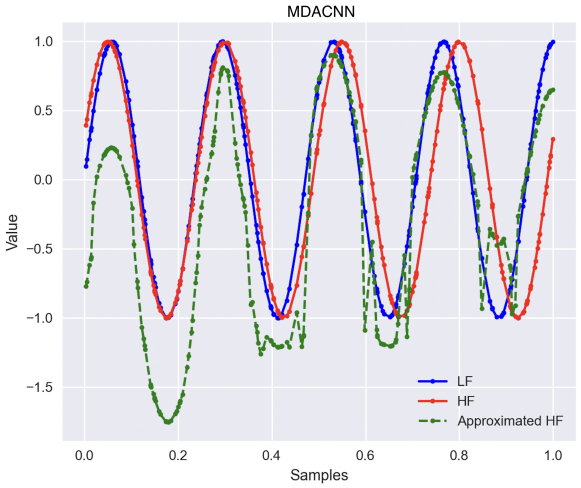
Figure B.3.: Benchmark: [5] Phase-shifted oscillations.



(a) Benchmark provided by [2].

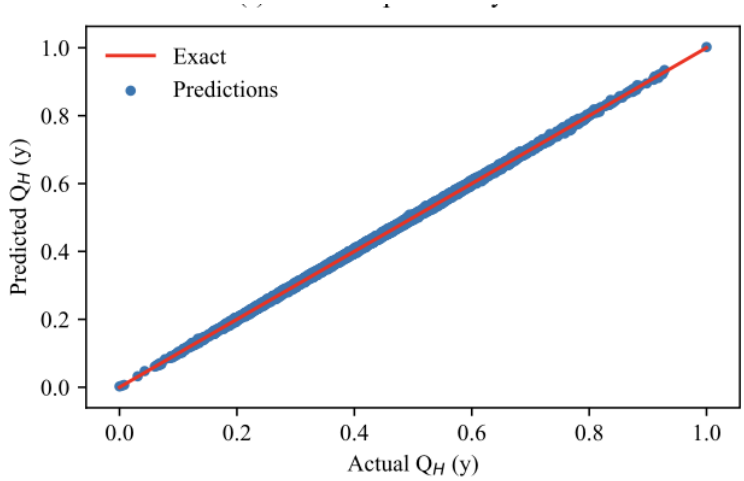


(b) Benchmark after implementing the MDACNN model and utilise the API and MDACNN architecture described in [2].

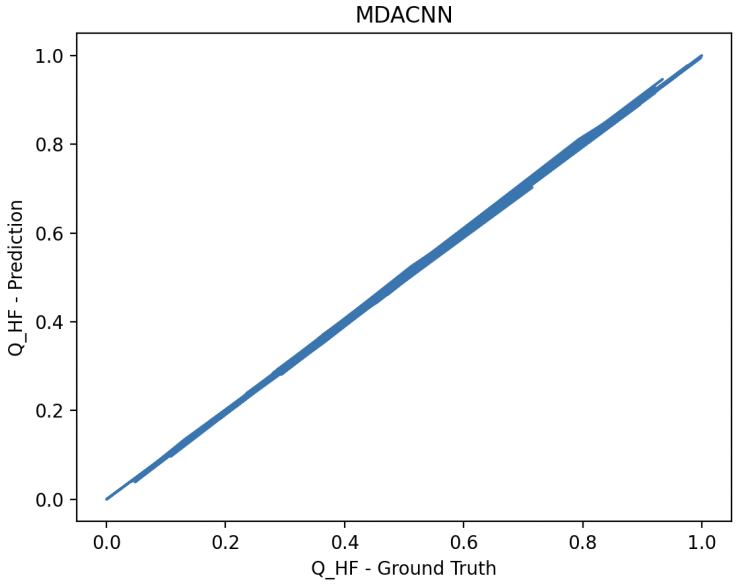


(c) Benchmark after improving the generalisation ability of the MDACNN architecture by extending the non-linear branch in the MDACNN architecture and adding more neurons and layers.

Figure B.4.: Benchmark: [6] Different periodicity .



(a) Benchmark provided by [2].



(b) Benchmark after implementing the MDACNN model and utilise the API and MDACNN architecture described in [2]. Out of all seven benchmarks, this is the only one, where the given MDACNN architecture and the given hyperparameters settings lead to the same result as shown in [2].

Figure B.5.: Original benchmark: [7] 50-dimensional functions

C. Optimization Experiments regarding the MDACNN

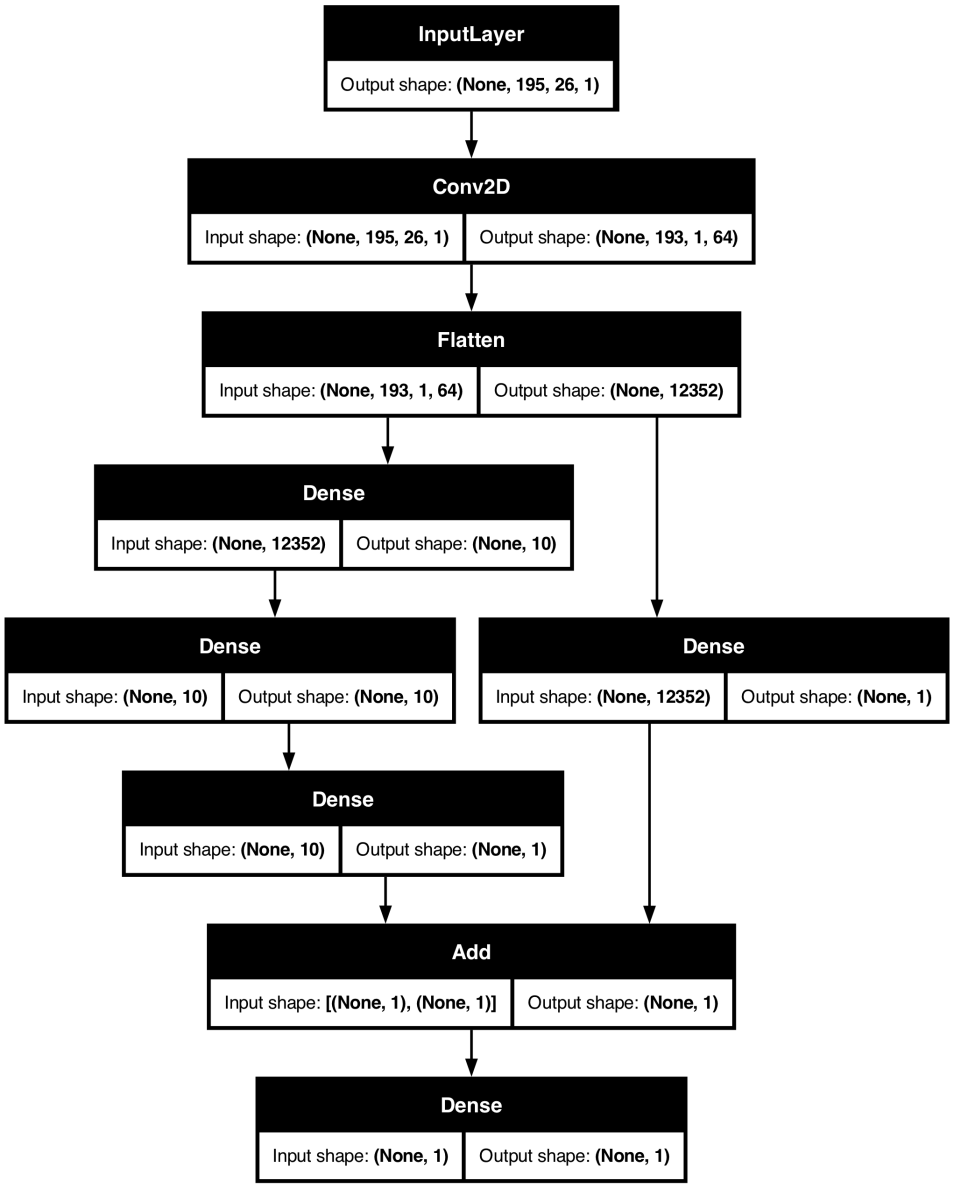


Figure C.1.: Original MDACNN architecture defined by Chen et al. [2]

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 195, 26, 1)	0	-
conv1 (Conv2D)	(None, 193, 1, 64)	5,056	input_layer[0][0]
flatten (Flatten)	(None, 12352)	0	conv1[0][0]
dense_1 (Dense)	(None, 10)	123,530	flatten[0][0]
dense_2 (Dense)	(None, 10)	110	dense_1[0][0]
dense (Dense)	(None, 1)	12,353	flatten[0][0]
dense_3 (Dense)	(None, 1)	11	dense_2[0][0]
add (Add)	(None, 1)	0	dense[0][0], dense_3[0][0]
dense_4 (Dense)	(None, 1)	2	add[0][0]

Total params: 141,062 (551.02 KB)
Trainable params: 141,062 (551.02 KB)
Non-trainable params: 0 (0.00 B)

Figure C.2.: Amount of parameters and their distribution in the original MDACNN

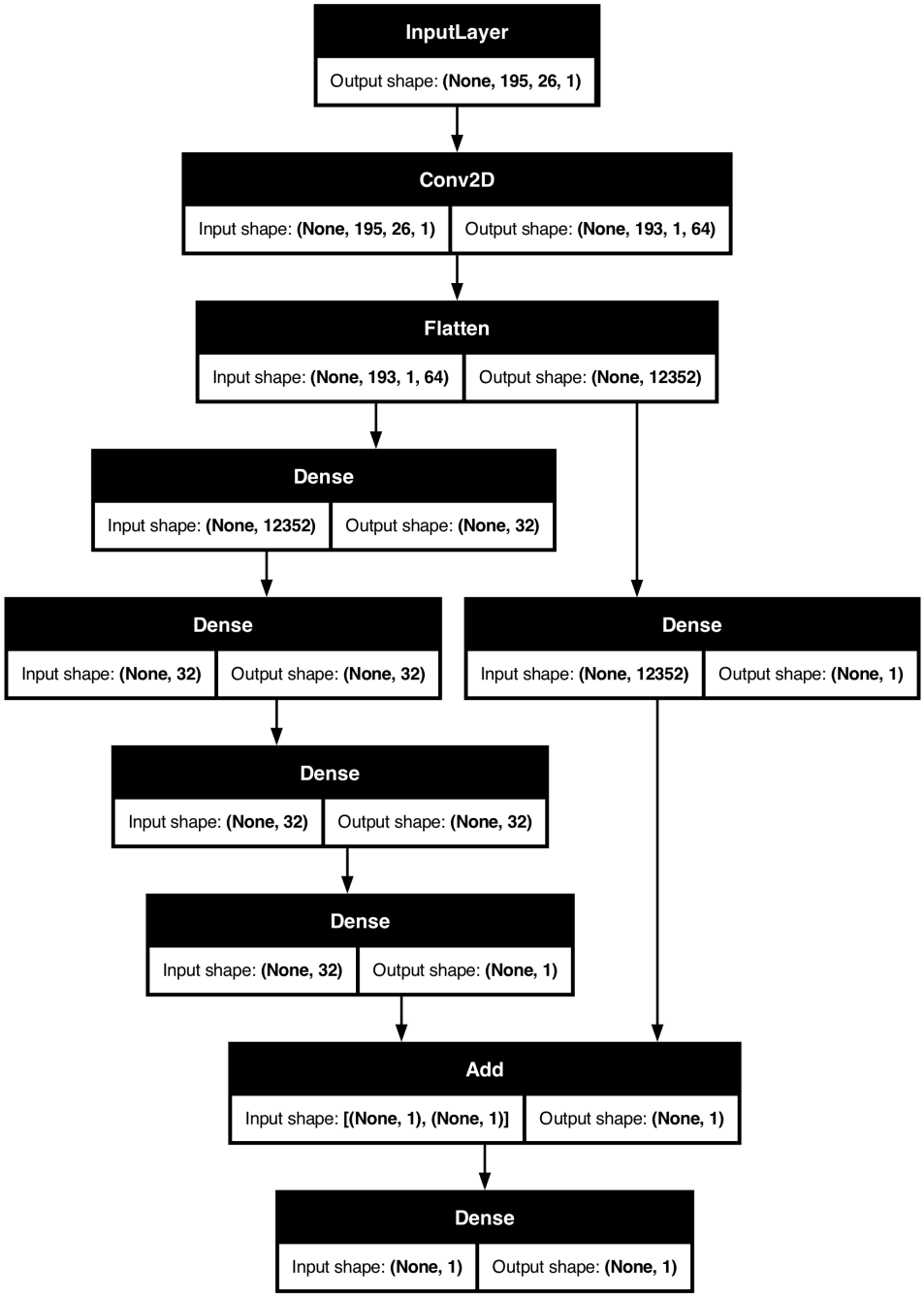


Figure C.3.: 3x32 MDACNN architecture. The 3x32 MDACNN architecture is an updated version of the original MDACNN architecture where the non-linear branch with two layers of each neuron got replaced with three layers of each 32 neurons.

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 195, 26, 1)	0	-
conv1 (Conv2D)	(None, 193, 1, 64)	5,056	input_layer[0][0]
flatten (Flatten)	(None, 12352)	0	conv1[0][0]
dense_1 (Dense)	(None, 32)	395,296	flatten[0][0]
dense_2 (Dense)	(None, 32)	1,056	dense_1[0][0]
dense_3 (Dense)	(None, 32)	1,056	dense_2[0][0]
dense (Dense)	(None, 1)	12,353	flatten[0][0]
dense_4 (Dense)	(None, 1)	33	dense_3[0][0]
add (Add)	(None, 1)	0	dense[0][0], dense_4[0][0]
dense_5 (Dense)	(None, 1)	2	add[0][0]

Total params: 414,852 (1.58 MB)
Trainable params: 414,852 (1.58 MB)
Non-trainable params: 0 (0.00 B)

Figure C.4.: Amount and distribution of parameters in the 3x32 MDACNN architecture.

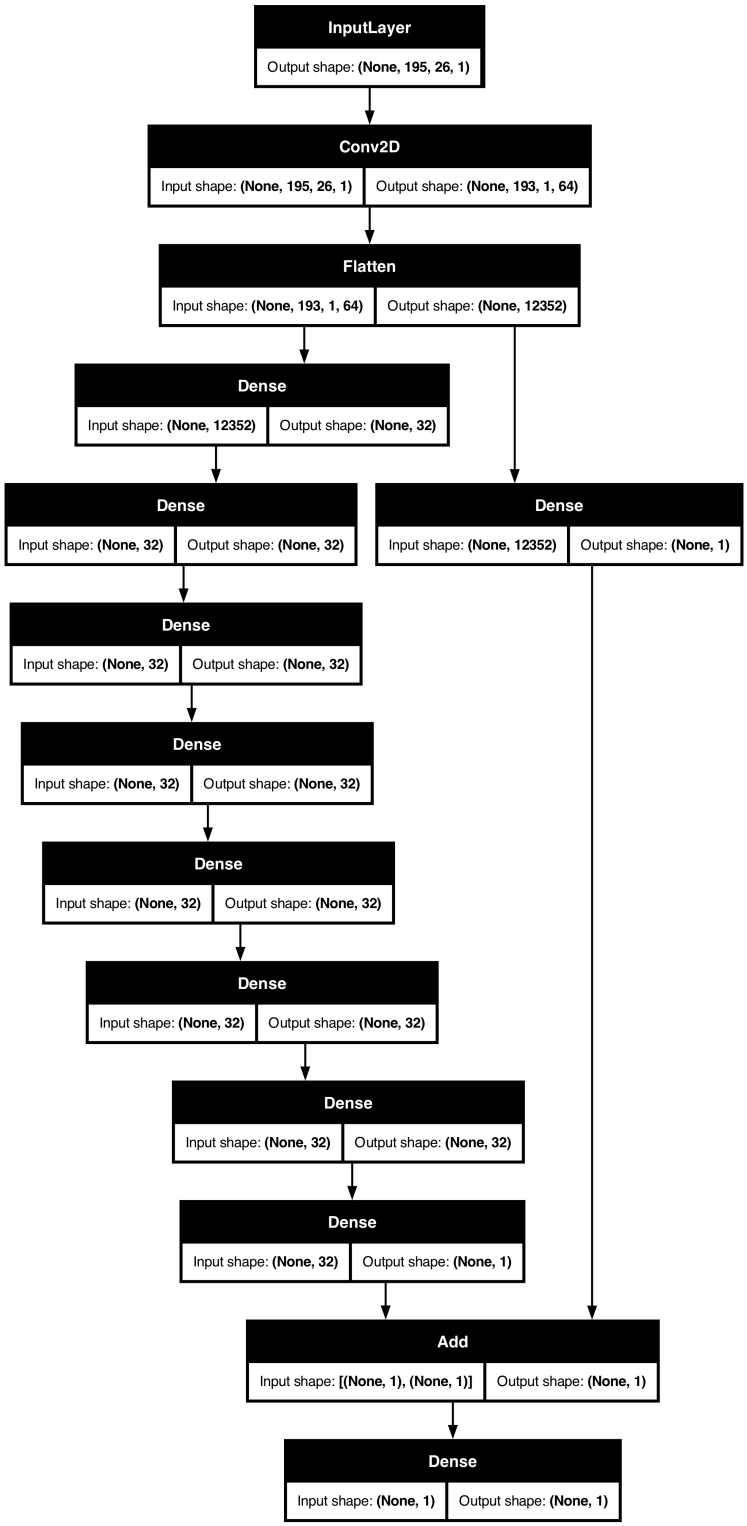
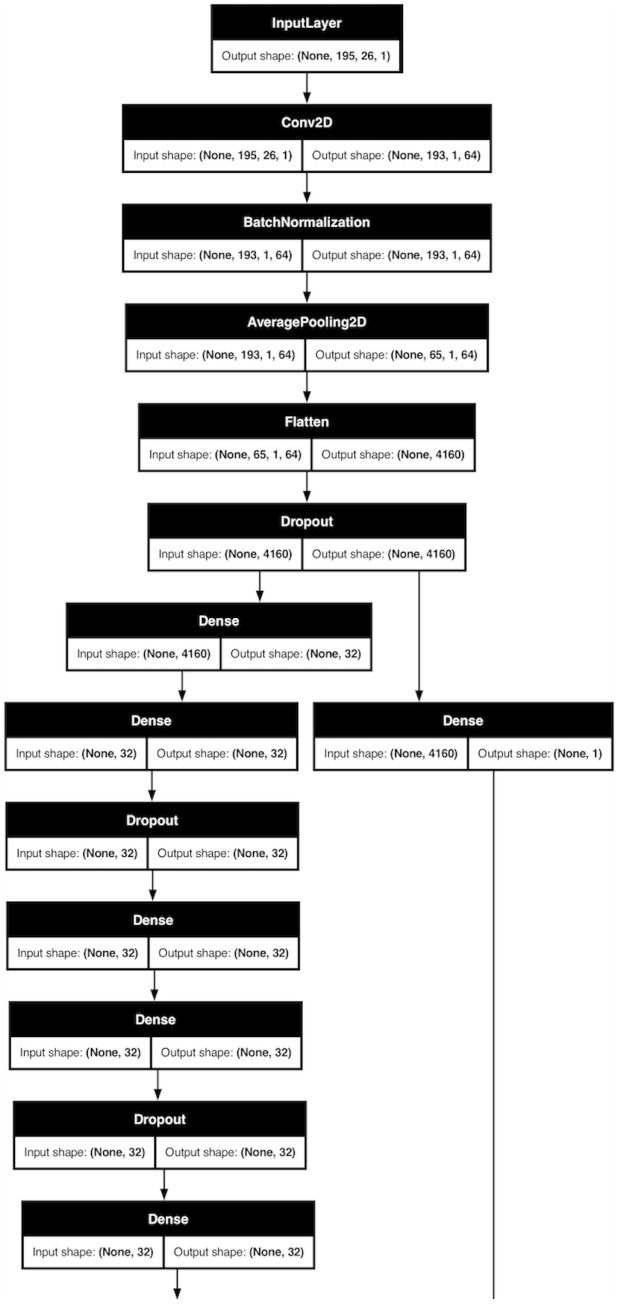


Figure C.5.: 7x32 MDACNN architecture. The 7x32 MDACNN architecture is a version of the original MDACNN architecture where the original non-linear branch with 2 layers with each 10 neurons got replaced with 7 layers of each 32 neurons.

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 195, 26, 1)	0	-
conv1 (Conv2D)	(None, 193, 1, 64)	5,056	input_layer[0][0]
flatten (Flatten)	(None, 12352)	0	conv1[0][0]
dense_1 (Dense)	(None, 32)	395,296	flatten[0][0]
dense_2 (Dense)	(None, 32)	1,056	dense_1[0][0]
dense_3 (Dense)	(None, 32)	1,056	dense_2[0][0]
dense_4 (Dense)	(None, 32)	1,056	dense_3[0][0]
dense_5 (Dense)	(None, 32)	1,056	dense_4[0][0]
dense_6 (Dense)	(None, 32)	1,056	dense_5[0][0]
dense_7 (Dense)	(None, 32)	1,056	dense_6[0][0]
dense (Dense)	(None, 1)	12,353	flatten[0][0]
dense_8 (Dense)	(None, 1)	33	dense_7[0][0]
add (Add)	(None, 1)	0	dense[0][0], dense_8[0][0]
dense_9 (Dense)	(None, 1)	2	add[0][0]

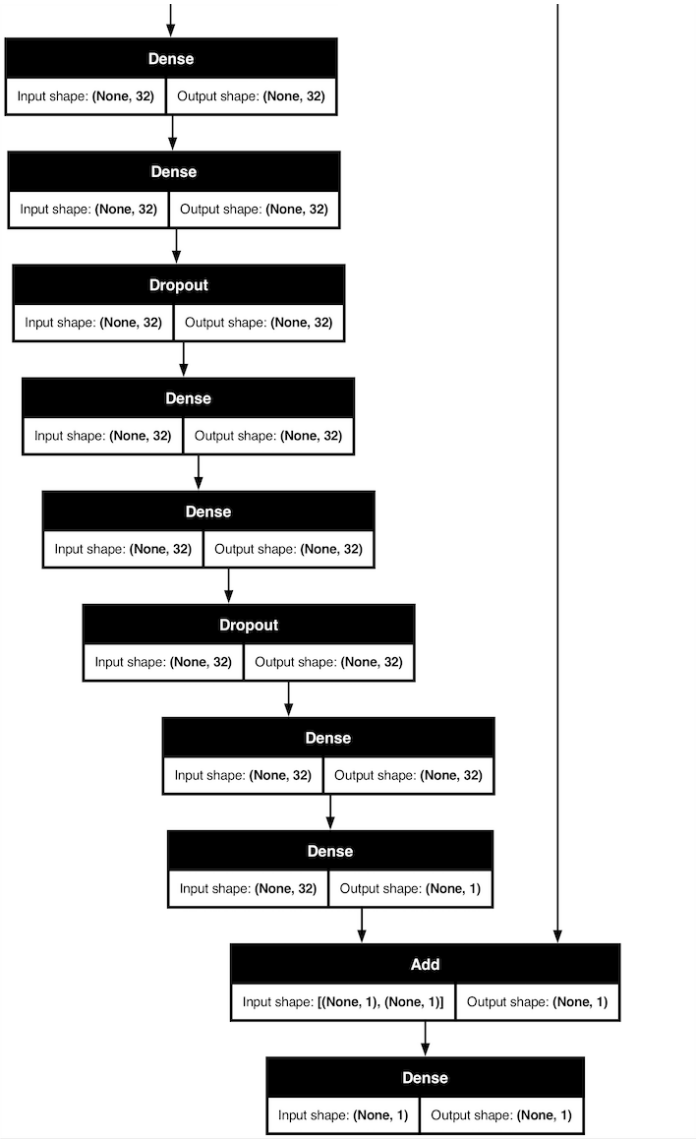
Total params: 419,076 (1.60 MB)
Trainable params: 419,076 (1.60 MB)
Non-trainable params: 0 (0.00 B)

Figure C.6.: Amount and distribution of parameters in the 7x32 MDACNN architecture.



(a) Upper half.

Figure C.7.: Regularized 7x32 MDACNN architecture. The regularisation used the methods of batch normalization, pooling and dropout.



(b) Lower half.

Figure C.7.: Regularized 7x32 MDACNN architecture. The regularisation used the methods of batch normalization, pooling and dropout.

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 195, 26, 1)	0	-
conv1 (Conv2D)	(None, 193, 1, 64)	5,056	input_layer[0][0]
bn_conv1 (BatchNormalization)	(None, 193, 1, 64)	256	conv1[0][0]
average_pooling2d (AveragePooling2D)	(None, 65, 1, 64)	0	bn_conv1[0][0]
flatten (Flatten)	(None, 4160)	0	average_pooling2d[0][0]
dropout (Dropout)	(None, 4160)	0	flatten[0][0]
dense_1 (Dense)	(None, 32)	133,152	dropout[0][0]
dense_2 (Dense)	(None, 32)	1,056	dense_1[0][0]
dropout_1 (Dropout)	(None, 32)	0	dense_2[0][0]
dense_3 (Dense)	(None, 32)	1,056	dropout_1[0][0]

(a) Upper half.

Figure C.8.: Amount and distribution of parameters in the Regularized 7x32 MDACNN architecture.

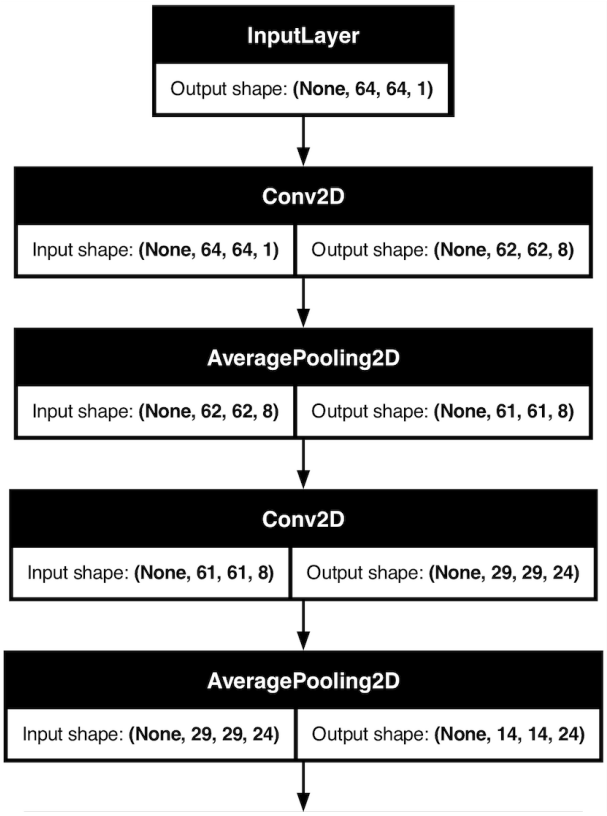
dense_4 (Dense)	(None, 32)	1,056	dense_3[0][0]
dropout_2 (Dropout)	(None, 32)	0	dense_4[0][0]
dense_5 (Dense)	(None, 32)	1,056	dropout_2[0][0]
dense_6 (Dense)	(None, 32)	1,056	dense_5[0][0]
dropout_3 (Dropout)	(None, 32)	0	dense_6[0][0]
dense_7 (Dense)	(None, 32)	1,056	dropout_3[0][0]
dense (Dense)	(None, 1)	4,161	dropout[0][0]
dense_8 (Dense)	(None, 1)	33	dense_7[0][0]
add (Add)	(None, 1)	0	dense[0][0], dense_8[0][0]
dense_9 (Dense)	(None, 1)	2	add[0][0]

Total params: 148,996 (582.02 KB)
Trainable params: 148,868 (581.52 KB)
Non-trainable params: 128 (512.00 B)

(b) Lower half.

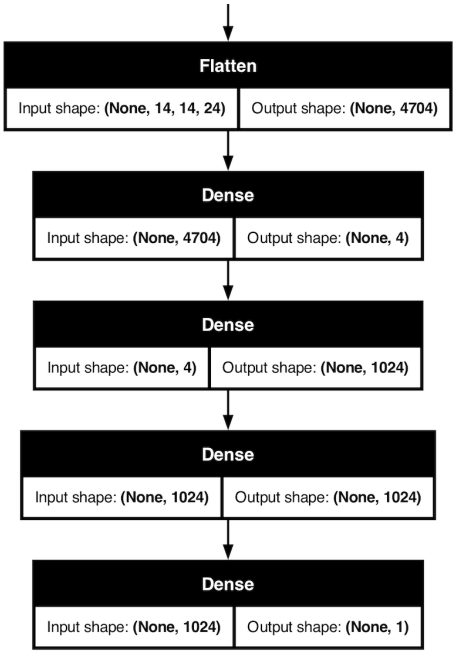
Figure C.8.: Amount and distribution of parameters in the Regularized 7x32 MDACNN architecture.

D. Experiments on the MFCNN-TL architecture



(a) Upper half.

Figure D.1.: Original MFCNN-TL architecture by [9]. The first CONV2D has 8 3x3 kernels with a stride of 1 and a linear activation. The first AveragePool2D has a 2x2 kernel with a stride of 1. The second CONV2D possesses 24 5x5 kernels with a stride of 2 and a linear activation. The second AveragePool2D possesses a 2x2 kernel with a stride of 2. The first Dense possesses four neurons with linear activations. The second and third Dense possess 1024 neurons with PReLU activation function. The PReLU activation function was realised using a LeakyReLU during the experiments. The output layer possesses a single neuron with a linear activation.



(b) Lower half.

Figure D.1.: Original MFCNN-TL architectur by [9]. The tensor flow through the layers is shown. The architecture is closer described in the upper half of Fig. D.1.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 64, 64, 1)	0
conv1 (Conv2D)	(None, 62, 62, 8)	80
AveragePool1 (AveragePooling2D)	(None, 61, 61, 8)	0
conv2 (Conv2D)	(None, 29, 29, 24)	4,824
AveragePool2 (AveragePooling2D)	(None, 14, 14, 24)	0
flatten (Flatten)	(None, 4704)	0
FC1 (Dense)	(None, 4)	18,820
FC2 (Dense)	(None, 1024)	5,120
FC3 (Dense)	(None, 1024)	1,049,600
FC4 (Dense)	(None, 1)	1,025

Total params: 1,079,469 (4.12 MB)
Trainable params: 1,079,469 (4.12 MB)

Figure D.2.: Total number and distribution of parameters in the MFCNN-TL.

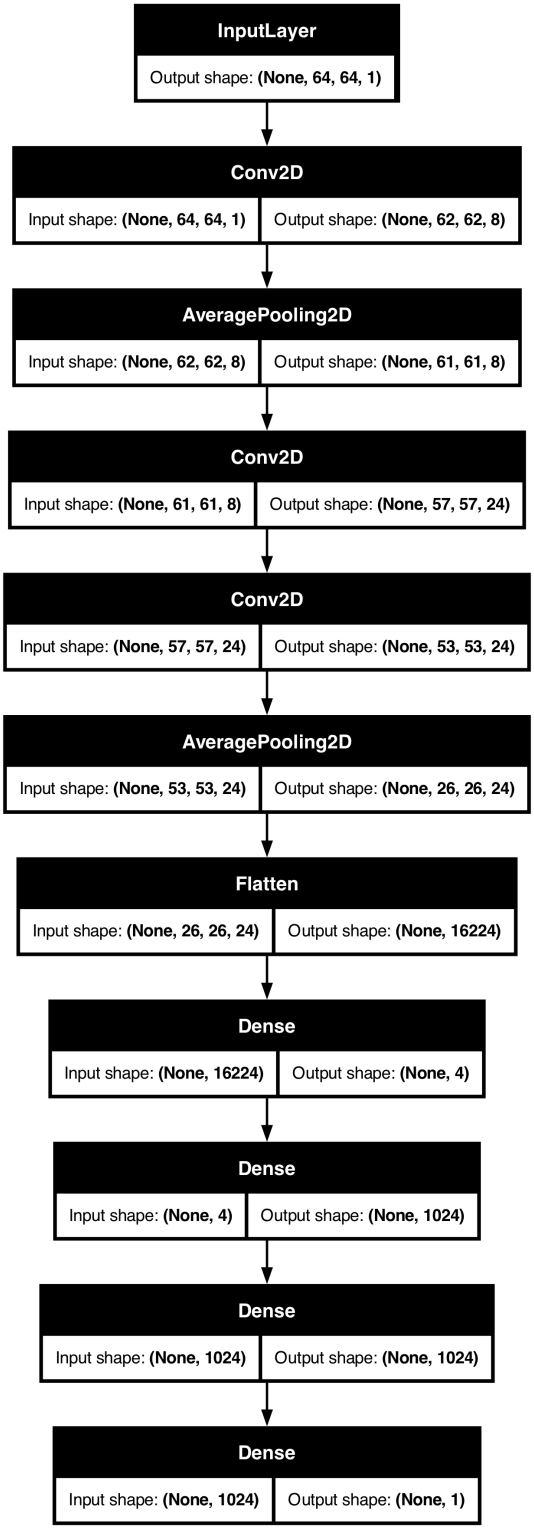


Figure D.3.: 1x8+2x24 MFCNN-TL architecture. The first Conv2D has 8 3x3 kernels with a stride of 1 and linear activation. The second and third Conv2D have each 24 5x5 kernels with a stride of 1 and linear activation. The rest of the architecture is similar to the original MFCNN-TL architecture.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 64, 64, 1)	0
conv1 (Conv2D)	(None, 62, 62, 8)	80
AveragePool1 (AveragePooling2D)	(None, 61, 61, 8)	0
conv21 (Conv2D)	(None, 57, 57, 24)	4,824
conv23 (Conv2D)	(None, 53, 53, 24)	14,424
AveragePool2 (AveragePooling2D)	(None, 26, 26, 24)	0
flatten (Flatten)	(None, 16224)	0
FC1 (Dense)	(None, 4)	64,900
FC2 (Dense)	(None, 1024)	5,120
FC3 (Dense)	(None, 1024)	1,049,600
FC4 (Dense)	(None, 1)	1,025

Total params: 1,139,973 (4.35 MB)
Trainable params: 1,139,973 (4.35 MB)
Non-trainable params: 0 (0.00 B)

Figure D.4.: Total number and distribution of parameters in the 1x8+2x24 MFCNN-TL architecture.

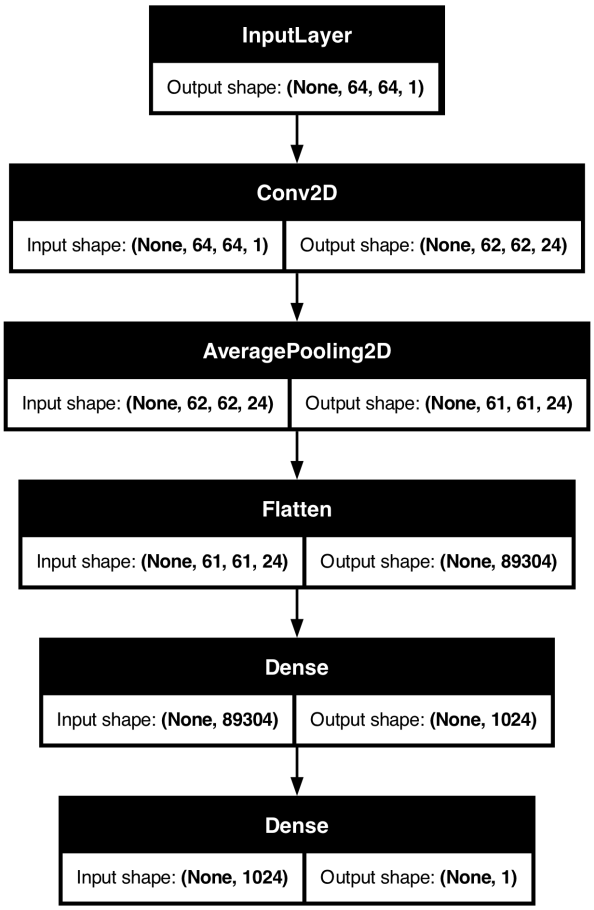


Figure D.5.: 1x24 MFCNN-TL architecture. The 1x24 MFCNN-TL architecture is a derivation from the Original MFCNN-TL architecture by [9]. Visualised is the tensor flow through the layers. Conv2D possesses 24 3x3 kernels with a stride of 1 and a linear activation. AveragePooling2D utilises a 2x2 kernel with a stride of 1. The first dense (fully connected layer) layer has 1024 neurons and the output layer has one neuron with a linear activation.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 64, 64, 1)	0
conv1 (Conv2D)	(None, 62, 62, 24)	240
AveragePool1 (AveragePooling2D)	(None, 61, 61, 24)	0
flatten (Flatten)	(None, 89304)	0
FC3 (Dense)	(None, 1024)	91,448,320
FC4 (Dense)	(None, 1)	1,025

Total params: 91,449,585 (348.85 MB)
Trainable params: 91,449,585 (348.85 MB)
Non-trainable params: 0 (0.00 B)

Figure D.6.: Total number and distribution of parameters in the 1x24 MFCNN-TL architecture.

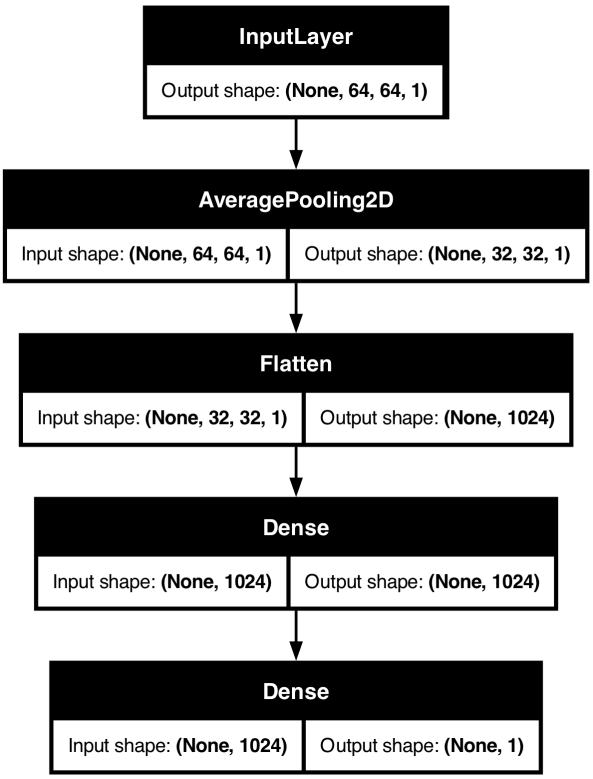


Figure D.7.: Perceptron MFCNN-TL architecture. Visualised is the tensorflow through the layers. AveragePool2D possesses a 2x2 kernel with stride 2. The first Dense has 1024 neurons with a PReLU activation (got realised with a LeakyReLU activation). The output layer possesses a single neuron with linear activation.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 64, 64, 1)	0
AveragePool1 (AveragePooling2D)	(None, 32, 32, 1)	0
flatten (Flatten)	(None, 1024)	0
FC3 (Dense)	(None, 1024)	1,049,600
FC4 (Dense)	(None, 1)	1,025

Total params: 1,050,625 (4.01 MB)
Trainable params: 1,050,625 (4.01 MB)
Non-trainable params: 0 (0.00 B)

Figure D.8.: Total number and distribution of parameters of the Perceptron MFCNN-TL architecture.

E. Experiments on the MF-TLNN architecture

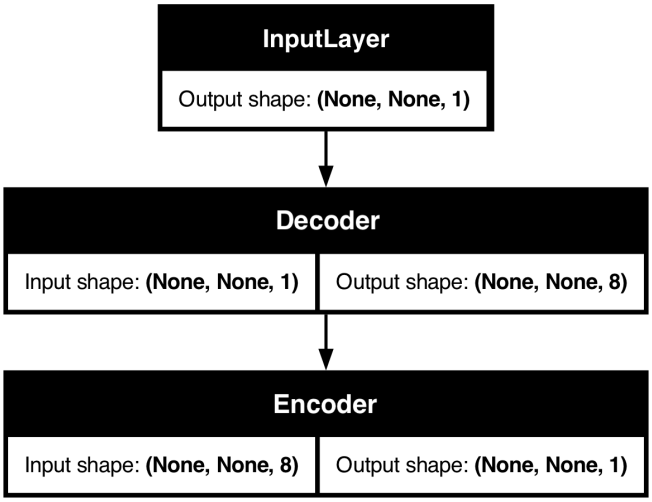


Figure E.1.: Architecture of the used Autoencoder (AE).

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, None, 1)	0
decoder (Dense)	(None, None, 8)	16
encoder (Dense)	(None, None, 1)	9

Total params: 25 (100.00 B)
Trainable params: 25 (100.00 B)

Figure E.2.: Total amount and distribution of parameters in the used Autoencoder (AE).

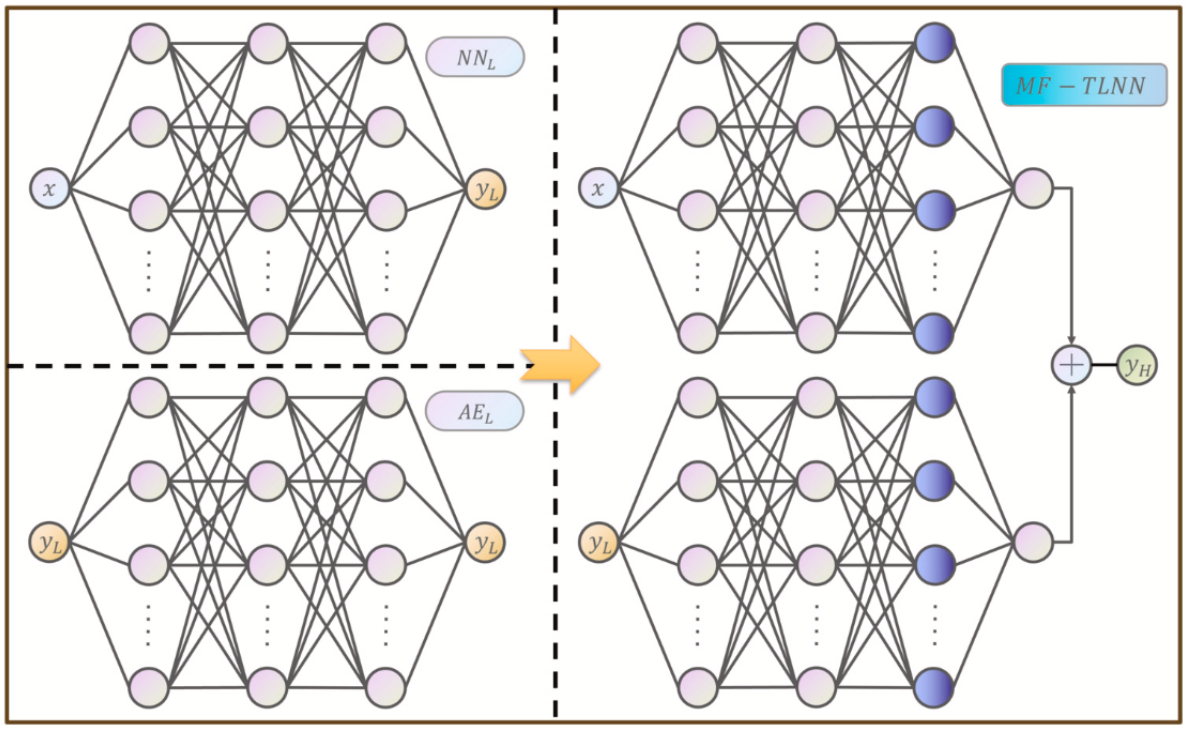


Figure E.3.: Original MF-TLNN architecture by Zhang et al. [10].

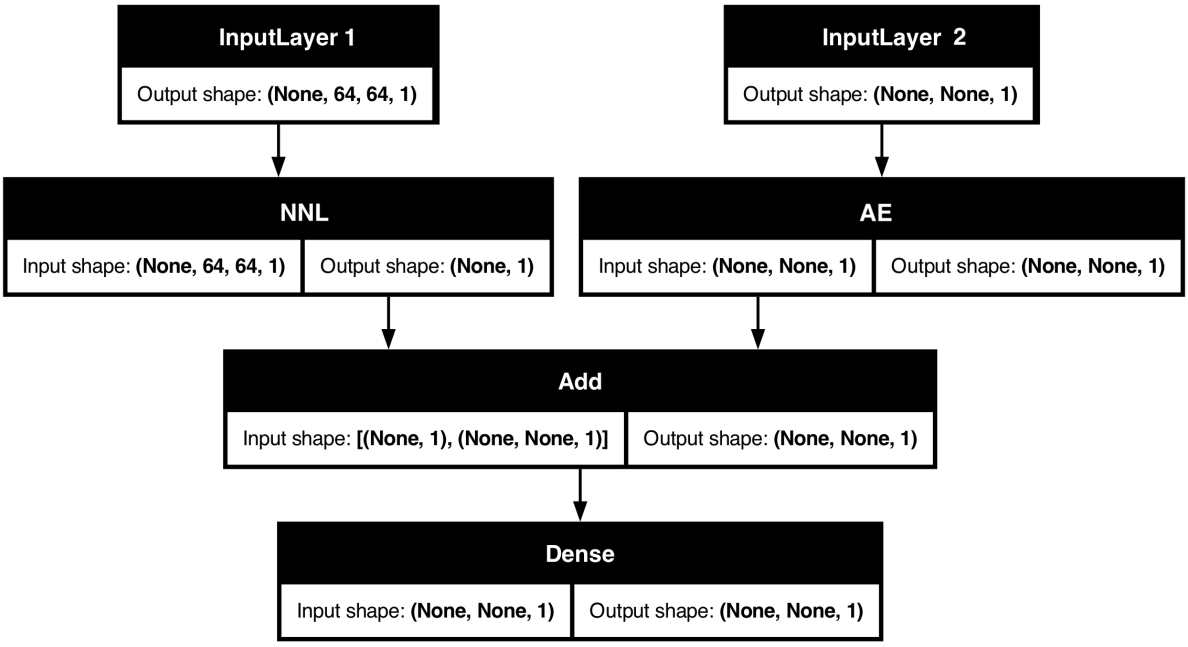


Figure E.4.: Architecture of the used MF-TLNN designed by [10].

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 64, 64, 1)	0	-
input_layer_2 (InputLayer)	(None, None, 1)	0	-
NNL (Functional)	(None, 1)	23,619,825	input_layer_1[0][0]
AutoEncoder (Functional)	(None, None, 1)	25	input_layer_2[0][0]
add (Add)	(None, None, 1)	0	NNL[0][0], AutoEncoder [0][0]
output_layer (Dense)	(None, None, 1)	2	add[0][0]

Total params: 23,619,852 (90.10 MB)
Trainable params: 23,619,612 (90.10 MB)
Non-trainable params: 240 (960.00 B)

Figure E.5.: Total amount and distribution of parameters in the used MF-TLNN designed by [10].