Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Automated quality enhancer for fast neural networks on mobile devices

Daniel Pilz

Course of Study:	Informatik
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Johannes Kässinger, M.Sc.
Commenced:	June 15, 2022
Completed:	December 15, 2022
Matriculation Number:	2952745

Abstract

While neural networks nowadays are widely used in different areas and applications, many usages of such neural networks are hindered by lacking computation power, especially on smaller devices like mobile devices. For this reason, it is very attractive to find ways to enable such devices to use neural networks under their restricted hardware conditions. There are different ways to approach that goal, but I will focus on an approach that enhances a given neural network to perform better on the same device and data without any distribution on other devices or reduction of potential redundancy. To achieve a better performance, I will create a supporting structure consisting of at least one supporting network processing input data instead of the original network. To decide which network processes which input, I will create a deciding neural network which will allocate each input value to exactly one network. This requires more computation in the training process because I will have to search the supporting and the deciding network and train them accordingly, but will reduce the error in the output of the whole network construction compared to the original network.

Contents

List of Abbreviations

DNN Deciding Neural Network. 9

DNNs Deciding Neural Networks. 10

MAE mean absolute error. 21

MNN Main Neural Network. 9

RSD Reduced Super Decider. 10

SD Super Decider. 10

SDS Super Decider Structure. 24

SNN Supporting Neural Network. 9

SNNs Supporting Neural Networks. 10

1 Introduction

Over the course of the last five to ten years, the topic of machine learning got more and more attention both in science and in everyday applications. While highly specialized and unique applications in science or big companies can rely on huge computation resources provided by stacking of GPUs and CPUs to push machine learning models to the limits, there are also applications that use machine learning while being deployed on end user devices. Combined with the growing amount of mobile devices many people rely on, like smartphones, tablets and smartwatches, there is a need to optimize applications for these devices. Machine learning is a computation-heavy task and mobile devices are limited in terms of computation power. This is why machine learning isn't easy to realize on mobile devices with the same efficiency. Using neural networks requires more computation the more layers and nodes the network has. Therefore, the limitations on mobile devices lead to neural networks that have less layers and nodes. While smaller networks require less computation power, they also tend to provide less accuracy. To address this issue, I want to enhance a given neural network without just adding layers and nodes to the existing network.

It is possible to enhance networks that can process some areas of the input with a sufficient accuracy and are very inaccurate in other areas. The algorithms presented in this work aim to enhance the accuracy for the input data areas where the initial network performs with bad accuracy. Networks that have a constant deviation in accuracy for all input data are less likely to profit from these approaches.

The original framework presented in Chapter 2 enhances a neural network that behaves as described above. The initial network is called the Main Neural Network (MNN). The MNN is enhanced by creating a structure consisting of multiple neural networks. The goal is to use the MNN for all input values where it performs sufficiently accurate and to use other networks for input data points where the MNN is not accurate enough. The MNN is complemented by Supporting Neural Network (SNN) and Deciding Neural Network (DNN). Each DNN is tied to exactly one SNN. On input of a data point, a DNN can either assign an input data point to its corresponding SNN or pass it to the next DNN. If an input data point is assigned to an SNN, this SNN computes the label to the input feature. Each input data point is fully processed by either exactly one SNN or the MNN. The MNN is the default processing network which processes all input features

that were not assigned to any SNN. The result of the framework is a structure called binary tree structure (BTS).

I create a new structure called Super Decider structure (SDS). This structure contains the MNN and n Supporting Neural Networks (SNNs). To decide which of these networks computes the label for an input data point, I use a special deciding network called Super Decider (SD). On input of a data point, the SD computes the index of the network which is chosen to compute the prediction for that data point. This solves the problem of the long inference time the BTS needs since each data point has to be processed by multiple Deciding Neural Networks (DNNs) before reaching an SNN or the MNN. The SDS has a constant inference time because every input data point is only processed by the SD and by the chosen SNN or MNN. my next step is to reduce the SD in size. This means I reduce the amount of layers the SD has as well as their dimensions to create a Reduced Super Decider (RSD). The RSD is negligibly worse in accuracy than the regular SD, but has an even lower inference time. I also developed new ways of training the SNNs before adding them to the structure. For the cover method, each SNN is trained with all input values from an input value area where no other network in the structure performs accurate enough. This means that if an input value is processed with an error below the threshold, it is never used as training data for subsequent networks. The mixed method combines the prediction training method and the cover method and merges the training data from both approaches to aggregate a new set of training data for each SNN. These new training methods allow more flexibility and enhance the accuracy of the SNNs. my last approach deletes the MNN from BTS. Instead, I add more SNNs to substitute the MNN. This results in a structure containing more of the specialized SNNs. This solves the problem that the original BTS still assigns most of the input data to the MNN, which was not accurate enough in the first place. Besides that, I implemented the framework for a new use case. This includes new data sets and a new neural network that has to be enhanced. I show that the approach is not only feasible for the original network, but also for other use cases.

One important feature of all these algorithms is that they work automated. This means that after the configuration of the run, no more input is needed. All SNNs and DNNs are trained automatically with the training method that was chosen in the configuration. The amount of SNNs added to the structure is also determined automated. New SNNs and DNNs are added as long as they improve the accuracy of the structure. Also the SD and its reduced versions are created and trained automatically to decide between all processing networks in the structure at the time they are created. To decide which data is assigned to which network, I need a threshold value. This threshold value is chosen by linear search between a minimal and a maximal value. These limits are passed as configuration parameters before the run and have to be set manually. A data point is assigned to an SNN when it is the first network in the structure that can compute a prediction for that data point with an error lower than the current threshold.

With the approaches in this work, I achieve multiple results. First of all I reduce the inference time considerably by creating a structure with an SD or RSD. These structures are only negligibly worse in accuracy than the original structure. Combined with the constant inference time for any amount of SNNs in the structure, it enables me to create larger structures with more SNNs without an increased inference time. I also improve the accuracy of the structure with the new training methods. I show that the SNNs can perform with a better accuracy if trained with the new training methods. The evaluation of the approach without the MNN shows that this approach results in a structure that performs with a better accuracy than the original one. I also show that the general idea of enhancing a neural network by creating a structure as described works not only for the original use case. The results for the second use case I implemented are similar to the improvements achieved in the original use case. This means that this approach of enhancing a given neural network is not a customized improvement for one use case, but seems to have a certain generality.

In Chapter 2, I present related work from the topic of machine learning as well as a work from the university of stuttgart, which lays the foundation of this work. I then give some information about the technical background in Chapter 3 before stating the problem in Chapter 4. After that, I present the algorithms in Chapter 5 and their implementation in Chapter 6 before showing the results of the work in Chapter 7. In Chapter 8, I summarize the work and then give an outlook for future work on the topic in Chapter 9.

2 Related Work

Since neural networks grew in importance in recent years, there has been more and more research concerning the topic of improving their performance with different approaches.

In [TMK17] the authors establish a “Distributed Deep Neural Network over the Cloud, the Edge and End Devices“. They face the problem of many small end devices that collect sensor data and a powerful neural network distributed in the cloud, where the data is processed. This means the collected data has to be processed by the end devices directly, which don’t have the computation power for a good neural network, or all data has to be passed to the cloud, which often leads to high communication cost due to bad latency. To address these problems, the authors distribute one deep neural network on a distributed computation hierarchy. This means the neural network is trained once and each device taking part in the computation therefore is contributing to the same network instead of having to aggregate the solutions of many different networks. Multiple end devices can work together to compute a solution by aggregation of their intermediate results. If this result isn’t of sufficient precision it is passed to the next layer, but it’s not longer raw data that has to be passed, but already processed data, which saves a significant part of the communication cost. This means we can avoid communication between end devices and the cloud or even the edge if the end devices themselves can produce a good solution and we can improve the communication if it is needed.

In [LBG+16] the authors develop “a software accelerator for low-power deep learning inference on mobile devices“ to approach the problem of running powerful neural networks on mobile devices with little computation power without draining the battery too fast through GPU-usage or requiring a constant connection to a cloud to run the neural network there. Their framework DeepX decomposes neural networks at runtime, so no additional training step is needed. It consists of two techniques: The Runtime Layer Compression tries to reduce redundancy by creating and integrating smaller layers between highly connected layers of the original neural network without affecting the quality of the output and is based on approaches of single value decomposition such as in [XLG13]. This technique reduces the parameters and computation cost of these layers. Deep Architecture Decomposition is a technique that finds building blocks in the neural network which it can allocate on certain computation devices to maximize the usage

of computation resources. Building blocks can be as small as the sum of computation steps needed to compute the solution of a single node in a layer. This technique needs a second component that processes the inference with the distributed building blocks and aggregates all different computations into the final solution of the computation.

In [TMW+18] an “adaptive deep learning model selection on embedded systems“ is developed. The selection is performed by a series of small neural networks, where each selecting network decides if the next processing neural network is chosen or not. If the network is not chosen to process the data, the next selecting network will evaluate the decision for the next network. The selection is made between certain pre-trained models that have been added to a pool of potentially fitting models. The pool consists of models that cover the set of input data as good as possible. After pool of models is created, the selecting networks are trained.

Difference to my approach

I don’t shift any computation to any other device, be it in a cloud or some other network. All the computation is done on the original device, which means I am not dependent on a constant connection to these other devices. This eliminates a big constraint for the approach. On top of that, I don’t want to optimize the network during runtime, but instead I focus on improving the network in the training phase. This gives me the advantage of almost no constraints considering the time, since training the network can be done in advance and is not time-critical. Furthermore, my goal is to enhance a neural network that has already been optimized for its intended use. This means I don’t actually optimize the given network by making this network better, but by constructing a structure of additional neural networks. These networks will allocate and process the input partially to use the original network if it performs well on the input value and assign other input values to supporting neural networks which perform well for some specific input value areas. This means I create an enhanced structure of multiple neural networks, which replaces the one original network instead of enhancing one neural network. In [TMW+18] there is a somewhat similar approach with multiple networks and a choice between these networks for each input value, but in contrast to that approach, I train the supporting networks specifically for the task I need them for instead of relying on pre-trained models, so I have specialized networks for different input value areas.

State of the art

The work in this thesis expands what Daniel Mantsch presented in [Man22].

This bachelor thesis from the University of Stuttgart created a framework to enhance a given neural network. I summarize the thesis to lay the foundations for the approaches presented in this work, for more details concerning [Man22] please see the work itself.

Design of the framework

Let X be a set of $m \in \mathbb{N}$ data points with $l \in \mathbb{N}$ features and Y be the corresponding set of m predictions of dimension $k \in \mathbb{N}$. Let A be a neural network which computes the prediction $\hat{y}_i$ for the data point x_i. Let the error e_i be $\frac{|\hat{y}_{i,0}-y_{i,0}|+...+|\hat{y}_{i,l-1}-y_{i,l-1}|}{l}$ and the overall error e^A produced by A be $\frac{e_0+...+e_{m-1}}{m}$. The framework enhances the neural network A in terms of accuracy, i.e. the error e^A is reduced. The result of the framework is a structure consisting of multiple neural networks, where the prediction $\hat{y}_i$ for each data point $x_i \in X$ is computed by exactly one neural network. The final structure consists of two different types of neural networks, the deciding networks and the processing networks. I will call this final structure BTS. A is the first network added to the structure and is called the MNN. It is one of the processing networks as it computes a prediction $\hat{y}_i$ on input of a data point x_i. The other processing networks are called SNNs. They also compute a prediction $\hat{y}_i$ on an input x_i. The difference between the MNN and the SNNs is the way they get trained.
For each SNN that is added to the structure there is also a deciding neural network (DNN) added.

On an input x_i a DNN will not compute a prediction $\hat{y}_i$, but computes a binary value $\hat{z}_i$ which is used to assign the input x_i to the corresponding SNN or not. If the first option is chosen, the SNN computes the prediction $\hat{y}_i$ for x_i. If the input is not assigned to that SNN, x_i is passed to the next DNN which processes the input the same way, but of course for another SNN. If no DNN assigns x_i to its corresponding SNN, this data point will be computed by the MNN as a default.
All networks have the same size and form, except for the output layer of the DNNs. The output layer is a layer with dimension l for the processing networks and of dimension 2 for the DNNs. The amount of layers and nodes of all neural networks is given by the form of the MNN, which is copied for all other networks except of the output layer. Note that the structure resulting from running the framework is bigger than the original MNN. The final structure in the end is an unbalanced binary tree as shown in Figure 2.1, with each node being a neural network. The processing networks are leaf nodes, with the

MNN being the leaf with the longest distance to the root in the structure. The nodes for the deciding networks have two children: one processing network and the next deciding network. The only exception is one DNN that has two processing networks as child nodes, the last SNN and the MNN itself. This is because there is no DNN for the MNN, since it is the default choice for all input features that were not assigned to any SNN. The structure is constructed in multiple iterations. In each iteration one SNN and one DNN is added. I call a structure with n pairs of SNN and DNN BTS^n. The only exception is the first iteration, in which the MNN is also trained and added. I will now explain in detail what happens in each iteration.

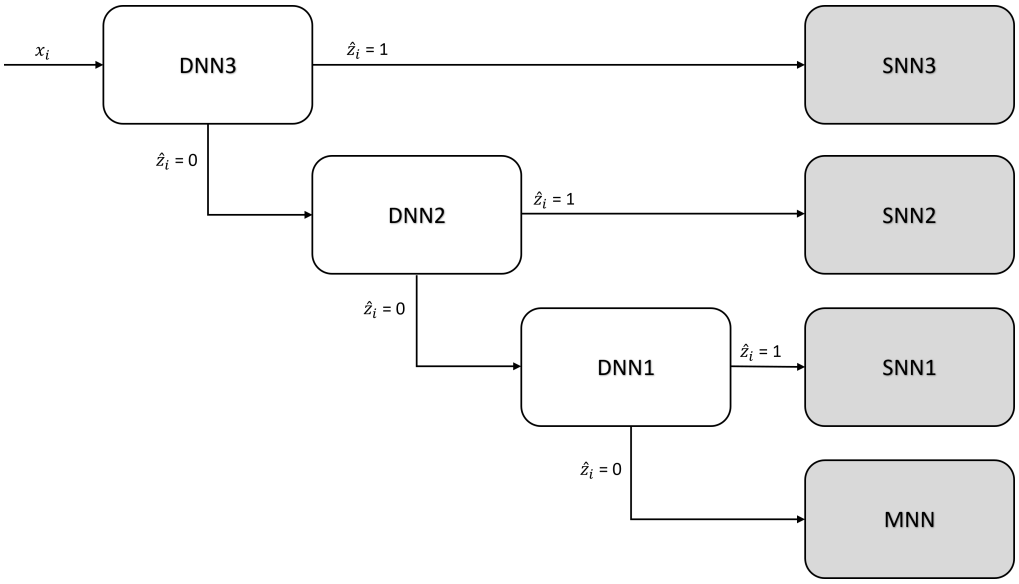

Figure 2.1: illustration of the binary tree structure

Iteration of the framework

For the training of SNNs and DNNs, a threshold value t is needed. To achieve the best results, multiple values for t are evaluated in each iteration. Let these values be $t_0, ..., t_s$ and let the set containing them be T. They are chosen by a linear search between the minimal value t_{min} and the maximum value t_{max}. The values t_{min} and t_{max} are read from the configuration file. The configuration parameters have to be filled in manually into the configuration file. The file is loaded at the start of every run to configure the conditions of the run. The values $t_j \in T$ with $j \in \mathbb{N}, j \leq s$ are iteratively chosen. For each value a DNN and SNN is trained and evaluated. This evaluation is performed using the whole structure so far plus the newly created DNN and SNN. The best pair is saved and compared to the ones that are evaluated after it. If a newly trained pair of DNN

and SNN performs better than the current best pair, the new pair is saved as best. After training a pair of DNN and SNN for every value $t_j \in T$, the best-performing pair of DNN and SNN is added to the structure.

As already mentioned, the first iteration is an exception as there is the extra step of creating and training the MNN and adding it to the structure. In all subsequent iterations, the MNN is not trained again, but simply loaded from the structure. The training data for the MNN is the set (X_T, Y_T), where $X_T \subset X$ and $Y_T \subset Y$. X_T and Y_T are randomly chosen from X, Y such that $x_i \in X_T \iff y_i \in Y_T$ and contain 80% of the values from X and Y. After creating and training (in the first iteration) or loading (in all following iterations) the MNN, the DNN is trained. The training data for the DNN has to be computed before training the network. Let this new training data be $(X_{\text{DNN}}, Z_{\text{DNN}})$ where X_{DNN} is a subset of X_T. The DNN's output is not a prediction $\hat{y}_i$, but a binary value $\hat{z}_i$. This means $Z_{\text{DNN}} = z_i \mid z_i \in 0, 1$. If $z_i \in Z_{\text{DNN}} = 1$ on the input data point $x_i \in X_{\text{DNN}}$, x_i is assigned to the corresponding SNN. Otherwise, x_i is passed to the next DNN (or the MNN if this is the last DNN in the structure). To compute $z_i \in Z_{\text{DNN}}$ for all x_i in iteration r, the data points x_i are processed by the structure BTS^{r-1} to compute predictions $\hat{y}_i$. The first DNN is added in iteration 1, so the structure BTS^0 contains only the MNN. In any subsequent iteration r, there are the MNN and $r-1$ DNNs and SNNs in the structure. If the error e_i produced by the BTS^{r-1} on input x_i is greater than t_j, z_i is set to 1. Otherwise, z_i is set to 0. Remember that the creation of the training data and the training of the DNN are performed for each value $t_j \in T$ separately.

After the DNN has been trained using the newly computed set $(X_{\text{DNN}}, Z_{\text{DNN}})$, the SNN is created and trained. Let the training data for the SNN be $(X_{SNN}, Y_{SNN}) \subseteq (X_T, Y_T)$. It holds: $x_i \in X_{SNN} \iff y_i \in Y_{SNN}$ and $x_i \in X_{SNN} \iff \hat{z}_i = 1$, where $\hat{z}_i$ is the output computed by the DNN on input x_i. This means that the SNN is trained with the data points (and the corresponding predictions) that the DNN assigns to that SNN. After the SNN has been trained, it is evaluated how well the structure $\overline{BTS^r}$ performs. $\overline{BTS^r}$ is the structure BTS^{r-1} with the newly trained pair of DNN and SNN. Note that both networks are not actually added yet to BTS^{r-1}, this happens later when it is clear which DNN and SNN are the best. In the evaluation, the mean absolute error $e^{\overline{BTS^r}}$ for all $x_v \in X_V = (X \setminus X_T)$ is computed. If the result of the evaluation is better than the current best result, it is saved as best result along with the DNN and SNN. After having the DNN and SNN trained and evaluated for all $t_j \in T$, the best-performing pair of DNN and SNN is added to the structure. The pseudo-code Algorithm 2.1 illustrates the procedure of one iteration of finding new networks.

Algorithm 2.1 find new networks (r, BTS^{r-1})

for $t_j \in T$ **do**
  if r = 1 **then**
    Create and Train MNN using (X_T, Y_T)
  else
    Load MNN
  end if
  compute $(X_{\text{DNN}}, Z_{\text{DNN}})$ by evaluating BTS^{r-1} on input X_T
  train DNN using $(X_{\text{DNN}}, Z_{\text{DNN}})$
  compute $(X_{SNN}, Y_{SNN}) : (x_i, y_i) \in (X_{SNN}, Y_{SNN}) \iff \hat{z}_i = 1$
  train SNN using (X_{SNN}, Y_{SNN})
  result ← evaluate($\overline{BTS^r}$)
  if result < best result **then**
    best result ← result
    best DNN ← DNN
    best SNN ← SNN
  end if
end for
add best DNN and best SNN to the structure

Training method for the SNNs

I described the way the training data for the SNNs is aggregated previously in this chapter. I call this the prediction method. The author implemented another method for the training of the SNNs, but did not show the results when using this method. I call that training method the original approach from now on, while the training method that was described in my summary and in [Man22] itself is called the prediction method. The original approach does not use the newly created DNN to compute the training data (X_{SNN}, Y_{SNN}). Instead, only the structure BTS^{r-1} is used to compute the training data for the r-th SNN. To do so, the data set X_T is fed into the structure BTS^{r-1}. It holds: $x_i \in X_{SNN} \iff e_i^{BTS^{r-1}} > t_j$ and $y_i \in Y_{SNN} \iff x_i \in X_{SNN}$. This means that only those data points are used for training the new SNN, that can not be accurately processed by BTS^{r-1}. I chose to evaluate the original method to compare it to the other methods in Chapter 7 although I did not implement it on my own.

3 Technical Background

Tensorflow

Tensorflow was developed at Google and released in 2015. It was presented in [AAB+16]. Tensorflow is a framework that can be used in machine learning to express and execute algorithms. It enables using machine learning on heterogeneous distributed systems without the need to adapt the algorithm to the underlying system. Tensorflow uses so-called tensors to pass the data between different components of the computation. Tensors are arrays arrays that are configurable concerning the element type they can hold. To execute an algorithm with Tensorflow, the framework generates a directed acyclic graph. The nodes of the graph are operations with inputs and outputs. The edges of the graph are either dataflow edges or controlling edges. Dataflow edges represent a tensor flowing from one node's output to another one's input. Controlling edges represent dependencies between nodes. If a controlling edge goes from one node to another, the first node has to finish its execution before the latter one can activate.

Keras

Keras is a library used to create, train and evaluate neural networks. It is written in python and can be run on top of the Tensorflow framework. The developers describe it as simple, flexible and powerful [Cho+]. By this properties, Keras enables the user to quickly set up neural networks, train them and use them. I use Keras to define and train the neural network models. To do so, I simply call the methods to create a model, add layers and train the model with the training data I prepared. I present more implementation details in Chapter 6.

Yaml

Yaml is an acronym for either 'yet another markup language' or 'yaml ain't markup language' released in 2001. It is a language used for data-serialization and configuration files with the goal of still being readable for humans. To reach this goal, yaml uses indentation like in python for nested values instead of brackets or tags. For more information about yaml see [BEN]. I use yaml for the configuration file. To configure the framework, I created a config.yaml file. Here, I define key-value-pairs for all configuration parameters I want to have. The yaml-file is easily loadable into a python program. Since the config file already has a dictionary structure, I can easily convert it into a dict in the framework. This configuration dict has the same key-value-mapping I defined in the config file and I can use the parameters in the framework.

4 Problem Statement

Let X be a set of m data points with k features. Let Y be the corresponding set of m predictions of dimension l. Let $y_i \in Y$ be the prediction for $x_i \in X$ and let $y_{i,j}$ be the value for the j-th dimension for the prediction y_i. Let A be a neural network which takes the data points x_i as input and computes the prediction $\hat{y}_i$ and let t be the threshold for the maximum error $e_i = \frac{|\hat{y}_{i,0} - y_{i,0}| + ... + |\hat{y}_{i,l-1} - y_{i,l-1}|}{l}$. Let $e^A = \frac{e_0 + ... + e_{m-1}}{m}$ be the overall error produced by A on input of all x_i.
It holds: $\exists\, u, v : \forall\, j$ with $u \leq j \leq v : e_j \geq t$. The accuracy of A can be improved with the framework from [Man22], resulting in a structure BTS consisting of one MNN, n SNNs and n DNNs.

I aim to improve the framework by addressing multiple issues. The first issue is the inference time of BTS. As stated in [Man22], most of the data points are assigned to the MNN. Every data point that is assigned to the MNN is processed by $n + 1$ neural networks: All DNNs plus the MNN. This takes a lot of time even for the standard value of $n = 3$, which was tested in [Man22] and the inference time gets worse if more pairs of SNNs and DNNs are added.

The next issue is the suboptimal training of the SNNs. As described in Chapter 2, the original framework trains each SNN with the data that the corresponding DNN would assign them in a real run. However, the DNNs also produce an error which leads to some wrongly assigned values. This means that the SNNs are trained with data that is not from the area they are supposed to specialize in and in contrast, some of the input from exactly these areas is missing in the training data of the SNNs. Another issue is the high usage of the MNN when evaluating BTS. As stated in [Man22], most of the data points are assigned to the MNN. Since this is the network that I want to improve because it is not accurate enough, I create a new structure $NM - BTS$. This structure consists only of SNNs and DNNs, without the MNN. The missing MNN is compensated by adding more SNNs. By deleting the MNN, $NM - BTS$ can make better use of the specialized SNNs to achieve a lower overall mean absolute error (MAE) e^{NM-BTS}. The last issue I addressed is the limitation to exactly one use case in [Man22]. The author states that the framework was tested for another network, but couldn't achieve any results.

5 Design

This chapter expands on the brief introduction from Chapter 1. I refer to the framework from [Man22] as initial framework.

5.1 General details

The framework works in an automated way once it is configured by the user. This means that the framework selects the training data for any SNN or DNN automatedly without any user input needed. The optimal size of the structure, i.e. how many extra networks are added, can also be determined in an automated way. To do so, it is possible to add more networks until they don't improve the structure anymore. The SD is automatically trained to decide between all processing networks that are in the structure. To configure a run, the user has to determine the training method for the SNNs as well as a minimum amount of SNNs and DNNs to be added to the final structure and the limits of the threshold value. For more details on the implementation of the configuration see Chapter 6.

5.2 Super Decider

This approach enhances the structure BTS resulting from the initial framework by adding a single deciding network, the so-called SD. The SD replaces all DNNs in the structure. The initial framework created a structure consisting of exactly one MNN and n DNNs and SNNs respectively. Recall that each DNN is tied to exactly one SNN and could only predict in a binary way: Assigning an input value to this SNN or passing it to the next DNN. This can lead to a lot of neural networks processing the same data before reaching a result. So I substitute the n binary deciding networks with the SD, which will predict the processing network for each input value. On input of a data point, the SD computes the index c_i of the processing network the input data is assigned to. The SD consists of one input layer, one output layer and four layers in between, three Dense Layers and one Dropout Layer.

To reach this new structure SDS, I use the initial framework to create the structure BTS, which is the result from the framework presented in [Man22]. Then I create a list of all processing networks, called the net-list. The SNN closest to the root in the structure is added first to the net-list. The last network in the netlist is the MNN because it has the highest distance to the root. After creating the net-list, I create the training data for the SD. In order to do so, I partition X into the training data X_T and the validation data X_V, as it is done in every iteration of the initial framework. I then feed every input data point $x_i \in X_T$ to the structure. Let c_i be the index of the processing network which x_i gets assigned to. I save c_i by creating an array of size $n+1$ filled with zeros and assign 1 at position c_i of the array.

After this procedure, I have a training data set (X_T, D) with $D = \{d_i \mid d_i \in \{0,1\}^{n+1}\}$ being a set of arrays, where $d_i \in D$ marks c_i. I then use this training data to train the SD and create a new structure consisting of the SD and the net-list. I call this structure the Super Decider Structure (SDS). Note that it consists of the SD, the MNN and $n > 0$ SNNs. On input of a data point $x_i \in X$, SDS first computes the index $\hat{c_i} \in \mathbb{N}, \hat{c_i} \leq n+1$ of the processing network chosen to process x_i and then computes the prediction $\hat{y_i}$. When evaluating the structure, I feed the input data points $x_v \in X_V$ to the SD to get the index of the processing network x_i is assigned to. I then feed x_v to the chosen network to get the prediction $\hat{y_v}$ computed by my structure.

The following Algorithm 5.1 is a pseudo-code of the algorithm which is described above. Figure 5.1 shows the structure resulting from the SD-approach.

Algorithm 5.1 Super Decider

```
get BTS as result of the initial framework
Create net-list
for x_i ∈ X_T do
    Initialize d_i as 0-filled array of size n + 1
    Compute c_i using BTS
    d_i[c_i − 1] ← 1
end for
train_SD(X_T, D, n + 1)
```

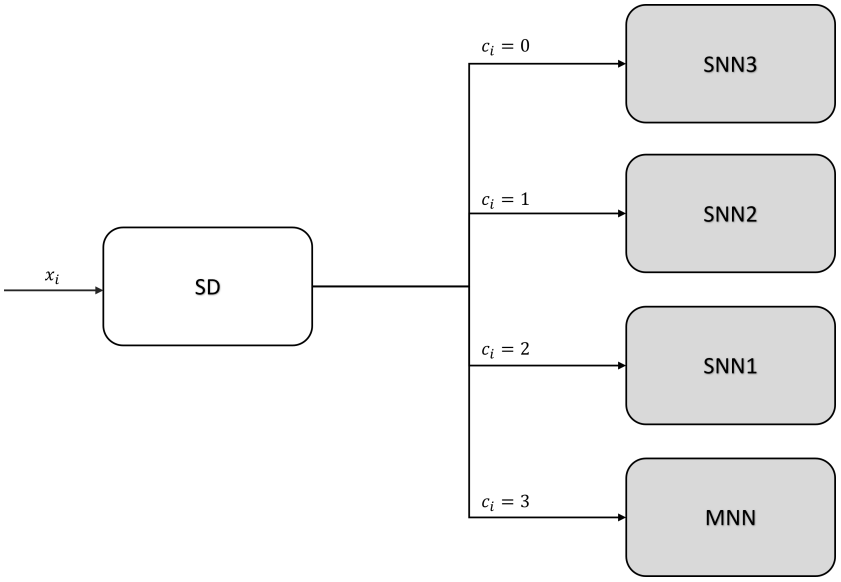

Figure 5.1: illustration of SDS

5.3 Reduced Super Decider

For this approach, I substitute the SD previously introduced with a remarkably smaller version, the RSD. The RSD is also a special version of a deciding network. It works just like the SD when fed with an input data point, but I expect it to be faster than the first version due to the smaller size. The RSD shall still perform accurate enough concerning the choice of the networks. I train the RSD with the same training data (X_T, D) as the SD. The net-list used for the SD is also used for the RSD, since the processing networks in the structure don't change.

The first layer I remove is the Dropout Layer. It is used to prevent overfitting, but in the first tests I quickly observed that the SD has no problems in this regard. After that, I create multiple variations of the RSD, each with different amounts of layers and nodes. All of these variations have same-sized input and output layers as the original SD, but differ in the way the layers between these two are arranged. This is why I only mention the middle layers when describing the layers of the RSD-variations in the following. The network uses the sequential form of layering, so all the layers come one after another. The input dimension of each layer is the output dimension of the predecessing layer.
The first variation of the RSD consists of one Dense Layer with output size 200. The second variation consists of two Dense Layers, each with an output size of 100. The third RSD has three Dense Layers with an output size of 50 each. The fourth version has two Dense Layers with output size 50.

The pseudo-code from Algorithm 5.1 works for the RSD-approach as well because the approaches differ only in the fact that a neural network with a different structure (layers, nodes, output sizes) is created. This doesn't change the preparation of training data and net-list, which is covered in the pseudo-code.

5.4 Training method: Cover

This approach modifies the way I generate the training data for the supporting networks. For this approach, I alter the framework from [Man22]. This means that the result of this approach is a structure just like the output from the initial framework. I call this structure $Cover - BTS$. The $Cover - BTS$ differs from the BTS only in the data used for the training of the SNNs. This means that $Cover - BTS$ can be further enhanced by substituting the DNNs with an SD. Recall that for the prediction method, each SNN was trained with exactly the input values the structure would assign to them if this was a real run. In contrast to that, I introduce a method I call the cover-method. The goal of this method is to cover each input data point x_i with a processing network that computes a prediction $\hat{y}_i$ with $e_i < t_j$.

For this method, I modify the sequence of each iteration of the initial framework after training the DNN. After the DNN has been created, I select the training data $(X_{\text{SNN}}, Y_{\text{SNN}})$ for the SNN. Let P be the set of all processing networks previously added to the structure. It holds: $x_i \in X_{\text{SNN}} \iff y_i \in Y_{\text{SNN}}$ and $x_i \in X_{\text{SNN}} \iff \forall p \in P : e_i^p > t$, where e_i^p is the error e_i produced by the network $p \in P$. After that, the SNN is trained with the selected data. In the first iteration, this approach leads to an SNN trained with all data points that the MNN can't process accurate enough. For the following iterations, the new SNN is only trained with data points that no other processing network can process with a sufficient accuracy.

The pseudo-code Algorithm 5.2 shows how I find new networks when using the cover-method for the training of the SNNs.

Algorithm 5.2 find new networks using the cover-method

for $t_j \in T$ **do**
  if $r = 1$ **then**
    Create and Train MNN using (X_T, Y_T)
  else
    Load MNN
  end if
  compute $(X_{\text{DNN}}, Z_{\text{DNN}})$ by evaluating BTS^{r-1} on input X_T
  train DNN using $(X_{\text{DNN}}, Z_{\text{DNN}})$
  compute $(X_{\text{SNN}}, Y_{\text{SNN}}) : (x_i, y_i) \in (X_{\text{SNN}}, Y_{\text{SNN}}) \iff \forall p \in P : e_i^p > t_j$
  train SNN using $(X_{\text{SNN}}, Y_{\text{SNN}})$
  result $\leftarrow$ evaluate($\overline{BTS^r}$)
  if result < best result **then**
    best result $\leftarrow$ result
    best DNN $\leftarrow$ DNN
    best SNN $\leftarrow$ SNN
  end if
end for
add best DNN and best SNN to the structure

5.5 Training method: Mixed

This is another approach of modifying the selection of training data for the SNN. As suggested by the name, it is a combination of the prediction method and the cover-method. This means I will generate a set of training data $(X_{\text{SNN}}, Y_{\text{SNN}})$ as described in chapter 7.3 and then combine it with the set generated with the prediction method. The newly generated data is cleared of all duplicates that were part of both training sets. The SNN is trained with the new, combined set. This overhead is not a big disadvantage because it occurs in the training phase, where I can spend a lot of time to train the networks. When using the trained networks to compute the predictions for the input data, the overhead is not an issue anymore.

The pseudo-code Algorithm 5.3 illustrates the process of finding new networks when using the mixed-method to gather the training data for the SNNs.

Algorithm 5.3 find new networks using the mixed-method

for $t_j \in T$ **do**
  if $r = 1$ **then**
    Create and Train MNN using (X_T, Y_T)
  else
    Load MNN
  end if
  compute $(X_{\text{DNN}}, Z_{\text{DNN}})$ by evaluating BTS^{r-1} on input X_T
  train DNN using $(X_{\text{DNN}}, Z_{\text{DNN}})$
  compute $(X_{\text{SNN}}, Y_{\text{SNN}}) : (x_i, y_i) \in (X_{\text{SNN}}, Y_{\text{SNN}}) \iff \forall p \in P : e_i^p > t_j \wedge \hat{z}_i = 1$
  train SNN using $(X_{\text{SNN}}, Y_{\text{SNN}})$
  result $\leftarrow$ evaluate($\overline{BTS^r}$)
  if result < best result **then**
    best result $\leftarrow$ result
    best DNN $\leftarrow$ DNN
    best SNN $\leftarrow$ SNN
  end if
end for
add best DNN and best SNN to the structure

Instead of combining the prediction and the cover method, it is also possible to combine two other training methods. For the second test case, I came up with an alternative version of the mixed training method that I call mixed*. This training method is a combination of the cover method and the original method. So the training sets $X_{\text{SNN}}, Y_{\text{SNN}}$ for the mixed*-method are a combination of the corresponding sets computed with the

cover method and the original method described in Chapter 2. I evaluate this special training method only for the second test case.

5.6 Substitution of the Main Network

For this approach I delete the MNN from BTS. Instead of the MNN, I add more of the specialized SNNs and their respective DNNs. This approach can be combined with the new training methods and with the SD presented in this work. To implement this approach, I use the original framework to get the BTS. I then delete the MNN from the structure and call this new structure $NM - BTS$. After that, I start a new iteration of the initial framework to find another SNN and DNN. For this iteration, only the new $NM - BTS$ is used. The training of the DNN and the SNN works as usual. It is worth noting that the training data sets $(X_{\text{SNN}}, Y_{\text{SNN}})$ will contain a lot more data since the MNN, which many input data points were assigned to, is missing now. After the iteration, I only add the best DNN and SNN if $e^{NM-BTS} < e^{\overline{NM-BTS}}$, where $\overline{NM - BTS}$ is $NM - BTS$ with the new DNN and SNN. This means I only add new networks to the structure if the overall MAE e^{NM-BTS} of the structure improves by doing so. If the new DNN and SNN are added, I start a new iteration of finding networks for the structure. This means I add new networks as long as I can find networks that improve the accuracy of the structure $NM - BTS$.

The following pseudo-code Algorithm 5.4 illustrates the process of deleting the MNN from BTS and adding net networks as compensation.

Algorithm 5.4 Substitution of MNN

```
1: get BTS as output from original framework
2: delete MNN from BTS to create NM − BTS
3: best result ← evaluate(BTS)
4: while terminated = 'false' do
5:     run for-loop from algorithm 3.1
6:     result ← evaluate(NM − BTS)
7:     if result < best result then
8:         best result ← result
9:         add DNN and SNN to NM − BTS
10:    else
11:        terminated ← true
12:    end if
13: end while
```

6 Implementation

In this chapter, we present the implementation of the key elements for the different approaches from Chapter 5. Since all approaches are extensions of the initial framework from [Man22], we will only show the parts of the code where the framework is extended to implement the new approaches.

6.1 General Details

To run the framework, it is necessary to load the configuration file. This file is a yaml-file where the user can set the parameters for the run. The configuration file is organized as a list of key-value-pairs. A configuration parameter is passed by setting the value for the corresponding key in the yaml-file. When running the framework, the configuration file is loaded into a dict named 'conf_dict' is created. This dict then holds all key-value-pairs from the configuration file.

The structures are stored as a recursive dict. For the structures without SD, each dict has four keys: last, net, decider and next_dict. 'last' is a boolean that indicates if the dict has a successing dict or not. 'next_dict' holds the successing dict if there is one. If not, the value is empty. The key 'decider' has a DNN as value. This DNN computes the label $\hat{z}_i \in 0, 1$ on input x_i. If $\hat{z}_i = 0$, x_i is passed to the successing dict. If $\hat{z}_i = 1$, the data point x_i is assigned to the network that is stored as value for the key 'net' of the same dict. Note that the dict holding the MNN has no value for the key 'decider'. This means that there is no dict with a DNN that has no successor.

6.2 Super Decider

For the Super Decider approach, I only show how I create and train the SD.

```
def trainSuperDecider(val_x, dict_, count_input_nodes, count_supporting_networks,
    trainingset_name, shapey, missing_index, reduced_decider=False):

   print("Train NN to predict error")
   networkcount = count_supporting_networks+1
   pred.optimizer_predict_generate_all(val_x, dict_, count_input_nodes, networkcount,
       shapey)
   tr_data, val_data, tr_x, tr_y, val_x, val_y = modify.shuffle_and_separate_tr_data\
      (os.path.join('../data', 'stack_predict' + '.csv'),count_input_nodes,networkcount,
          trainingset_name+"_predict")

   if reduced_decider:
      superDecider = create_and_optimize_predict_reduced(tr_x, tr_y, val_x, val_y,
          count_input_nodes, networkcount, missing_index) # , model_name)
   else:
      superDecider = create_and_optimize_predict(tr_x, tr_y, val_x, val_y,
          count_input_nodes, networkcount) #, model_name)

   return superDecider
```

Listing 6.1: method to prepare creation and training of the Super Decider

First, I set the amount of processing networks the SD can assign the input features to and save it as "networkcount ". I do this by incrementing the amount of SNNs by one to include the MNN.
I then use BTS, which is stored in the parameter "dict_", along with the given data points, the amount of networks and the shape of the output, to predict the chosen network for each input data point by calling 'optimizer_predict_generate_all'. This method will generate an array where the i-th row holds the input data point x_i in the first k columns and an integer value in the next n columns, where $n = networkcount$. This integer value is 0 for all columns but one, where it is 1. The column where the entry is 1 can be used to compute c_i by subtracting k from the column index, since the first k entries only hold the input value and don't contribute to the network counting. After filling this array for all input features, this array contains (x_i, c_i) in the i-th row. It is filled by feeding all x_i one after another into BTS and tracking the network that is chosen to process x_i.

This array is the training data for the Super Decider and is saved as a .csv file. I then load this file and call 'shuffle_and_separate_tr_data' , which will shuffle the training data and separate some of the data as validating data. After having received the training data by the function, I call 'create_and_optimize_predict' to create and train the Super

Decider network.

```
def create_and_optimize_predict(tr_x, tr_y, val_x, val_y, count_input_nodes,
    networkcount,):

   model =
       tf.keras.models.Sequential([tf.keras.layers.Flatten(input_shape=(count_input_nodes,)),
                                tf.keras.layers.Dense(200, activation='relu'),
                                tf.keras.layers.Dense(200, activation='relu'),
                                tf.keras.layers.Dropout(0.2),
                                tf.keras.layers.Dense(200, activation='relu'),
                                tf.keras.layers.Dense(networkcount, activation='softmax')])
   model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mse', 'mae'])

   idx_lst = modify.part_data(5, tr_x.shape[0])
   modify.cv_train(model,tr_x,tr_y,idx_lst,20,32)
   model.evaluate(val_x, val_y, verbose=0)

   model.summary()
   return model
```

Listing 6.2: method to create and train the Super Decider

I create the network with the help of the keras API. You can see the layers added to the model in the code. I then prepare the data for cross-validation training in five folds and call the 'cv_train'-method. Here the network gets trained by calling the 'fit'-method from the keras API. The last step is evaluating the new Super Decider and then returning it.

6.3 Reduced Super Decider

The RSD is a variation of the Super Decider approach where I reduced the amount of layers and also used smaller layers. As you can see in Listing 7.1, the preparation method of the RSD is the same as for the regular SD, but I call another method to actually create and train the RSD.

```
def create_and_optimize_predict_reduced(tr_x, tr_y, val_x, val_y, count_input_nodes,
    networkcount, missing_index):
   print(str(networkcount))

   if missing_index == 1:
      model =
          tf.keras.models.Sequential([tf.keras.layers.Flatten(input_shape=(count_input_nodes,)),
                                tf.keras.layers.Dense(200, activation='relu'),
                                tf.keras.layers.Dense(networkcount, activation='softmax')])
```

```
elif missing_index == 2:
    model =
        tf.keras.models.Sequential([tf.keras.layers.Flatten(input_shape=(count_input_nodes,)),
                            tf.keras.layers.Dense(100, activation='relu'),
                            tf.keras.layers.Dense(100, activation='relu'),
                            tf.keras.layers.Dense(networkcount, activation='softmax')])
elif missing_index == 3:
    model =
        tf.keras.models.Sequential([tf.keras.layers.Flatten(input_shape=(count_input_nodes,)),
                            tf.keras.layers.Dense(50, activation='relu'),
                            tf.keras.layers.Dense(50, activation='relu'),
                            tf.keras.layers.Dense(50, activation='relu'),
                            tf.keras.layers.Dense(networkcount, activation='softmax')])
if missing_index == 4:
    model =
        tf.keras.models.Sequential([tf.keras.layers.Flatten(input_shape=(count_input_nodes,)),
                            tf.keras.layers.Dense(50, activation='relu'),
                            tf.keras.layers.Dense(50, activation='relu'),
                            tf.keras.layers.Dense(networkcount, activation='softmax')])

model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mse', 'mae'])

idx_lst = modify.part_data(5, tr_x.shape[0])
modify.cv_train(model,tr_x,tr_y,idx_lst,20,32)
model.evaluate(val_x, val_y, verbose=0)

model.summary()
return model
```

Listing 6.3: method to create and train the Reduced Super Decider

In this method, I differ between four cases to create various neural networks. I call the training method for the RSD in a loop with all four indices one after another to create all different versions and compare them afterwards. Besides the creation of the neural network dependant of the given index, this method works the same way as the method from Listing 7.2.

6.4 Training Method for SNN: cover

For this approach, I change the way the SNN is trained by computing a different training set (X_{SNN}, Y_{SNN}). To do this, I first introduce a new option in the config file: 'training_mode'. I have three options for training_mode: prediction, which is the normal one described in Chapter 2, cover, which is presented here, and mixed, which is a combination of the two. If the training mode is set to 'cover', I will aggregate a new set

of data points and predictions used to train the SNN.
I compute (X_{SNN}, Y_{SNN}) after training the MNN, which means the data is computed for every threshold value I test. This is important since the threshold value is what determines which input features are part of X_{SNN}.

```
if conf_dict['training_mode'] != 'prediction':
  cover_data_x, cover_data_y = cError.generate_cover_data(np.vstack((val_x, train_x)),
      np.vstack((val_y, train_y)), main_model, model_mode, threshold_value)
```

Listing 6.4: Introduction of new variables for the training data

This data is passed to the training method of the SNN later. To compute the new training data, I use both the validation and the training data from the original approach. Note that the 'main_model' that is passed as parameter here can be the MNN , but can also be the whole structure BTS consisting of MNN, SNN and DNN that shall be expanded with another SNN and DNN. (X_{SNN}, Y_{SNN}) is computed by the 'generate_cover_data'-method:

```
def generate_cover_data(val_x, val_y, model, model_mode, threshold):
   data_x = []
   data_y = []
   candidates = np.full(np.shape(val_x)[0], 1)
   temp_model = model

   if model_mode == 'optimized':
      is_last = False
      while not is_last:
         result = temp_model['net'](val_x)
         for i in range(len(result)):
            if candidates[i] == 0:
               continue
            absolutes = 0.0
            for j in range(np.shape(val_y)[1]):
               absolutes = absolutes + modify.calc_mae_error(result, val_y, i, j)

            if absolutes / np.shape(val_y)[1] < threshold:
               candidates[i] = 0
         is_last = temp_model['last']
         if not is_last:
            temp_model = temp_model['next_dict']
      for i in range(len(candidates)):
         if candidates[i] == 1:
            data_x.append(val_x[i, :])
            data_y.append(val_y[i, :])

   else:
      result = model(val_x)
      for i in range(len(result)):
```

```
            absolutes = 0.0
            for j in range(np.shape(val_y)[1]):
                absolutes = absolutes + modify.calc_mae_error(result, val_y, i, j)

            if absolutes / np.shape(val_y)[1] > threshold:
                data_x.append(val_x[i, :])
                data_y.append(val_y[i, :])

    data_x = np.array(data_x)
    data_y = np.array(data_y)

    return data_x, data_y
```

Listing 6.5: method to aggregate the training data for the cover-approach

If I pass only the MNN as 'main_model', the model mode will not be 'optimized', and I can simply evaluate the MNN for all input features $x_i \in X_t$ to see if $e_i > t$. If this is the case, $(x_i, y_i) \in (X_{SNN}, Y_{SNN})$. These are stored in 'data_x' and 'data_y', so x_i and y_i are added to the corresponding arrays. This is covered by the final else-branch.

In iteration $r > 1$, the model mode will be 'optimized'. This means I have to evaluate the MNN and every SNN for every input data point and measure the error that occurs. I do this by loading the next model from the structure if the current model is not the last one. The models are stored in a dictionary. The key 'net' has the SNN for the current node of BTS as value. For every model $p \in P$, I evaluate the model with the training data X_T and measure the absolute error e_i^p for each $x_i \in X_T$. If this value is above the threshold value, I save the data as part of (X_{SNN}, Y_{SNN}). Before I add the values, I check if they are already included to prevent the occurence of duplicate values.
After evaluating every input feature with every network, I return the arrays 'data_x' and 'data_y' containing X_{SNN} and Y_{SNN}.

6.5 Training Method for SNN: mixed

For this approach, I combine the two previous training methods for the SNN: I aggregate all input features which no other network can process with an MAE below the threshold and add all features that the DNN will assign to the new SNN. I prevent any duplicate values occuring in the process from being added twice. To do so, I have to set the training mode to 'mixed'. I then call the method to aggregate $(X_{SNN}, Y_{SNN}$ after training the DNN, so I once again aggregate the training data once per threshold value.

```
if conf_dict['training_mode'] == 'mixed':
  cover_data_x, cover_data_y = cError.merge(data_x, data_y, cover_data_x, cover_data_y)
```

Listing 6.6: call to merge the training sets

I pass the two data sets from the other training methods: 'data_x' and 'data_y' contain (X_{SNN}, Y_{SNN}) computed by the prediction-approach, while 'cover_x' and 'cover_y' contain (X_{SNN}, Y_{SNN}) from the cover-method. This means that I have to compute the training sets from both approaches to get the new training sets.

```
def merge(data_x, data_y, b_x, b_y):
   partition_x = []
   partition_y = []
   for i in range(len(b_x)):
      partition_x.append(b_x[i, :])
      partition_y.append(b_y[i, :])
   for i in range(len(data_x)):
      in_list = False
      for k in range(len(b_y)):
         if np.array_equiv(data_x[i], b_x[k]):
            in_list = True
            break
      if not in_list:
         partition_x.append(data_x[i, :])
         partition_y.append(data_y[i, :])

   partition_x = np.array(partition_x)
   partition_y = np.array(partition_y)

   return partition_x, partition_y
```

Listing 6.7: method to aggregate the training data for the mixed-approach

I first add data points and predictions from one training set before adding all features from the other set while preventing duplicates from being added. This is important because the better the DNN performs, the bigger the intersection between the two sets will be. After merging the two training sets into one, I continue with the usual iteration by creating the SNN, training it with the new training data.

6.6 Substitution of the Main Network

After one full run of the original framework, I drop the the MNN from the structure by traversing through the tree of networks and deleting the MNN, which has to be the last entry in the tree.

```
deleted_main = False
no_main_dict = dict(dict_models)
current_dict = no_main_dict['next_dict']
while not deleted_main:
  next_dict = current_dict['next_dict']
  if next_dict['last'] == True:
    current_dict['last'] = True
    current_dict['next_dict'] = None
    deleted_main = True

    else:
    current_dict = next_dict
```

Listing 6.8: deletion of the MNN

First I copy the structure stored in 'dict_models'. I then traverse the structure until I find the dict with the value 'True' for the key 'last'. After that, I change the values of the predecessor of the last dict by setting the value for 'last' to 'True' and deleting the link to the former last dict. Now the dict with the MNN is no longer part of the structure. Note that I now have a last dict which contains a DNN and an SNN. The DNN from this dict will never be called, because this dict has no successor it could pass an input feature to.

After deleting the MNN from BTS, I aggregate the training data (X_T, Y_T) and the evaluation data (X_V, Y_V). This data is used to evaluate the new structure $NM - BTS$ by calling 'compute_metrics_no_main'. The resulting MAE is saved as 'last_best_stats'. It will be used to check if the newly found networks improve the structure. I then continue by running an iteration to find another pair DNN and SNN.

```
continue_search = True
train_x, train_y, val_x, val_y = modify.prepDataCords(conf_dict['data_location'],
    conf_dict['data_x_shape'][0], conf_dict['data_y_shape'][0], trainingset_name)

erg_no_main, no_main_mae = eval.compute_metrics_no_main(val_x, val_y, no_main_dict)
last_best_stats = erg_no_main

while continue_search:
  best, best_stats = exe.search_new_networks2(no_main_dict, False,
      search_multiple_supporting_networks, False, iteration, run_path, no_main_dict, )
```

```
main_model = best[0]
support_model = best[1]
decision_model = best[2]
search_multiple_supporting_networks = best[3]

if best_stats < last_best_stats:
  last_dict = dict(no_main_dict)
  no_main_dict = {
    'last': False,
    'net': support_model,
    'decider': decision_model,
    'next_dict': dict(last_dict)
  }
  last_best_stats = best_stats
  overall_stats.append(best_stats)
else:
  continue_search = False
```

Listing 6.9: search networks to compensate the missing MNN

With every call of 'search_new_networks2' I find one DNN and one SNN. I then check if the structure with the new networks $NM - BTS'$ performs better than the old structure $NM - BTS$ did. If this is the case, I add the DNN and the SNN to the structure by adding a new dict with the new networks and stay in the loop to start another iteration. If the new networks do not improve the structure, I set 'continue_search' to False and stop searching for more networks.

7 Evaluation

To evaluate the implemented approaches, we used two different test cases. Each test case consists of one set X of input data points and one set Y of corresponding predictions. Additionally, each test case includes the shape of a neural network, i.e. the amount of layers and the input and output dimensions of each layer. This shape is used for the MNN and the SNNs in the structure I create.

For each approach, I ran the framework multiple times and aggregated the results into one average result. By doing so, I tried to balance the variation between the different runs. To evaluate the performance of the different structures, I measured the mean absolute error as well as the time needed for computation. Note that I did not measure the time it took to search or train the networks, but only the time it took the final structures to compute the predictions $y_v \in Y_V$ on input of X_V.

7.1 Test Case: Muscle Activation

The first test case uses data sets from the PerSiVal project from the university of Stuttgart. Each data point $x_i \in X$ from this test case has $k = 4$ features: the angle of a human arm, the angular speed, the angular acceleration and the angular weight. The data points are mapped to predictions with dimension $l = 5$. Each dimension holds the activation value for a different muscle. The given neural network A consists of six layers. The input layer is a Flatten layer with input shape $k = 4$. I then have two Dense layers with output size 200, one Dropout Layer and two more Dense Layers, with the last layer being the output layer with output size $l = 5$. This shape will be used for all neural networks in the structure, with the exception of the SD and RSDs. The shape of these networks is described in Chapter 6. All runs were made with $t_{min} = 0,001$ and $t_{max} = 0,035$. The stepsize for the linear search between those values was $0,0025$. All values $x_i, y_i \in \mathbb{R}$.

7.1.1 Super Decider and Reduced Super Decider

To evaluate the performance of the SD, I created the structure as described in Section 5.2. To test the different variations of the RSD, I created multiple structures with the different version of RSDs as presented in Section 5.3. For the evaluation of the SD and the RSDs, I always created a structure with $n = 3$ SNNs (and also 3 DNNs in case of the BTS-structure). I then measure the MAE of the different structures as well as the MAE of the initial structure BTS with n DNNs and n SNNs. The MAE is measured as the average error over all $e_i, 0 \leq i < m$. I also measure the inference time for all these structures. The inference time is the time the structure needs to compute a set of predictions Y' on input of a set of data points X'. On the average of twelve runs with different training methods for the SNNs, I have the results listed in Table 7.1.

	MNN	BTS	SD	RSD1	RSD2	RSD3	RSD4
MAE	0,0126	0,007	0,0071	0,0071	0,0071	0,0071	0,0072
% of MNN	100	56	56	56	56	57	57
inference time in s	18,32	80,1	41,82	33	36,58	39,51	36,17

Table 7.1: test case 1: accuracy and inference time of different SD-approaches

The BTS-structure has an average MAE of less than a third of the average MAE of the MNN (58%). The SD has a slightly higher average MAE compared to the BTS-structure, but the difference is so low that the average MAE is still only 59% of the MNN's average MAE. I also observed that there was only little variance in the SD's performance. This means that the difference between the structure with SD and the BTS-structure was always around the same, so the SD performed very consistent. Figure 7.1 shows the plot of the MAE computed by BTS and the plot of the MAE computed by the structure with an SD. Both structures were created in the same run. On the y-axis, the error e_i is shown. In each of the graphs, the data points are sorted after one feature. The plots show that the difference between the original structure and the structure using the SD concerning the MAE is very small. Apart from some few points that are different, the error for the input data points is mostly the same for both structures. With these observations, I can conclude that the SD works pretty accurate and I can use a structure with the SD, the MNN and the SNN instead of the BTS-structure without having a noteworthy loss in accuracy.

All versions of the RSD have MAE-values very similar to the regular SD. They only differ after the third decimal. All RSD versions have the same percentage value when comparing their MAE values to the MAE of the MNN. I think that a difference in the MAE at the fourth decimal and the ratio to the MNN's MAE being equal is a good result

concerning how much smaller the RSD networks are. Figure 7.2 shows the plot of the predictions computed by the structure with the regular SD on the left and the predictions computed by a structure with RSD4 on the right. The graphs look very similar, but the predicted error is higher for some data points when computed by the structure with the RSD4. This shows that the regular SD performs slightly better, but there are no groups of input data points that are remarkably worse with the RSD4 than with the SD. The graphs for the other versions of the RSD look very similar to Figure 7.2. To conclude the evaluation of the RSD versions, I think the difference to the regular SD is low enough that all tested RSD versions can substitute the regular SD. Since RSD4 is the smallest network, I prefer this version above the other ones besides its slightly worse performance.

Concerning inference time, I clearly see that the BTS-structure takes very much time to compute the predictions. The MNN alone takes 18,32 seconds to evaluate. With 80 seconds, the evaluation of the BTS-structure takes more than four times as long as the evaluation of the MNN alone. The structure with the SD taxes 41,82 seconds to evaluate. This is approximately half as long as the BTS-structure, but still more than twice as long as the MNN alone. I can improve the inference time further by using the RSDs. For these networks, I can clearly see that the evaluation takes longer the more layers a network has. RSD3 has the most layers from all RSD versions and with an inference time of 39,51 seconds, its structure is only a little faster than the one with the regular SD. The fastest RSD is RSD1. The structure using it takes 33 seconds to evaluate. This is less than twice the amount of time the MNN needs. The structure with RSD4 has an inference time of 36,17 seconds. While this is slower than the one with RSD1, it is still around half as fast as the MNN alone and more than twice as fast as the BTS-structure.

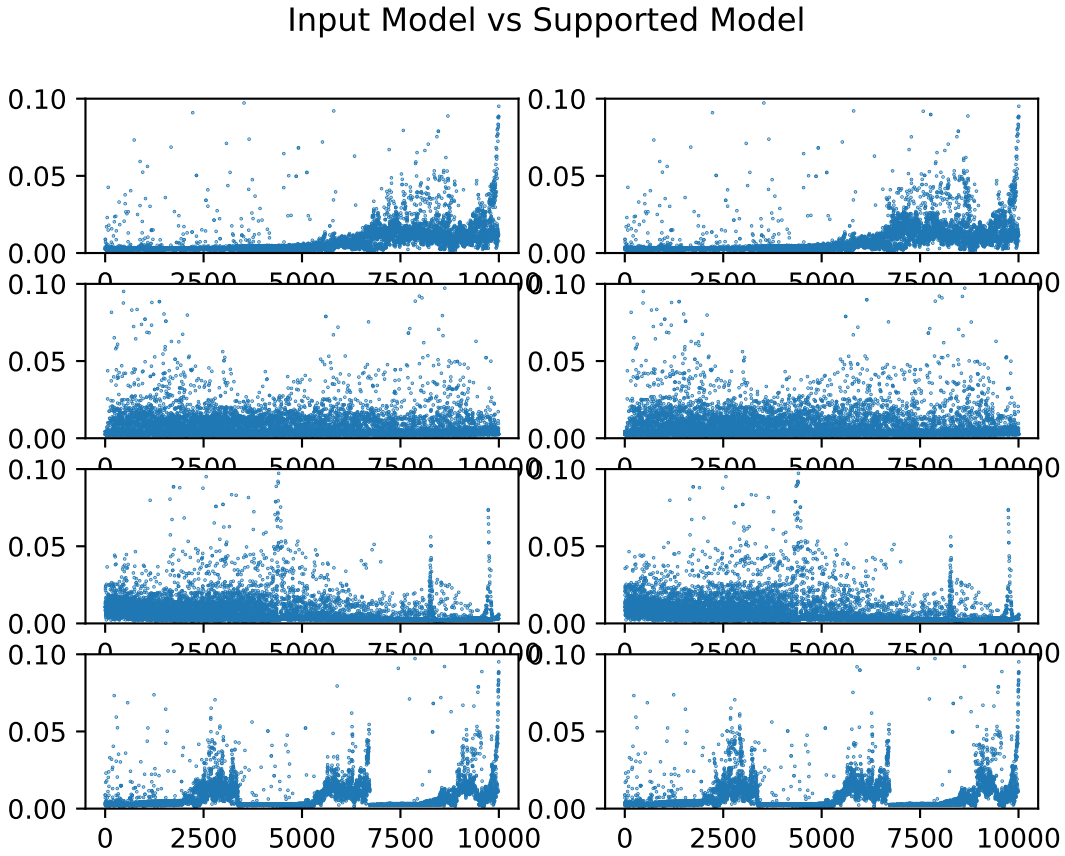

Figure 7.1: left: plot of BTS, right: plot of the structure with SD

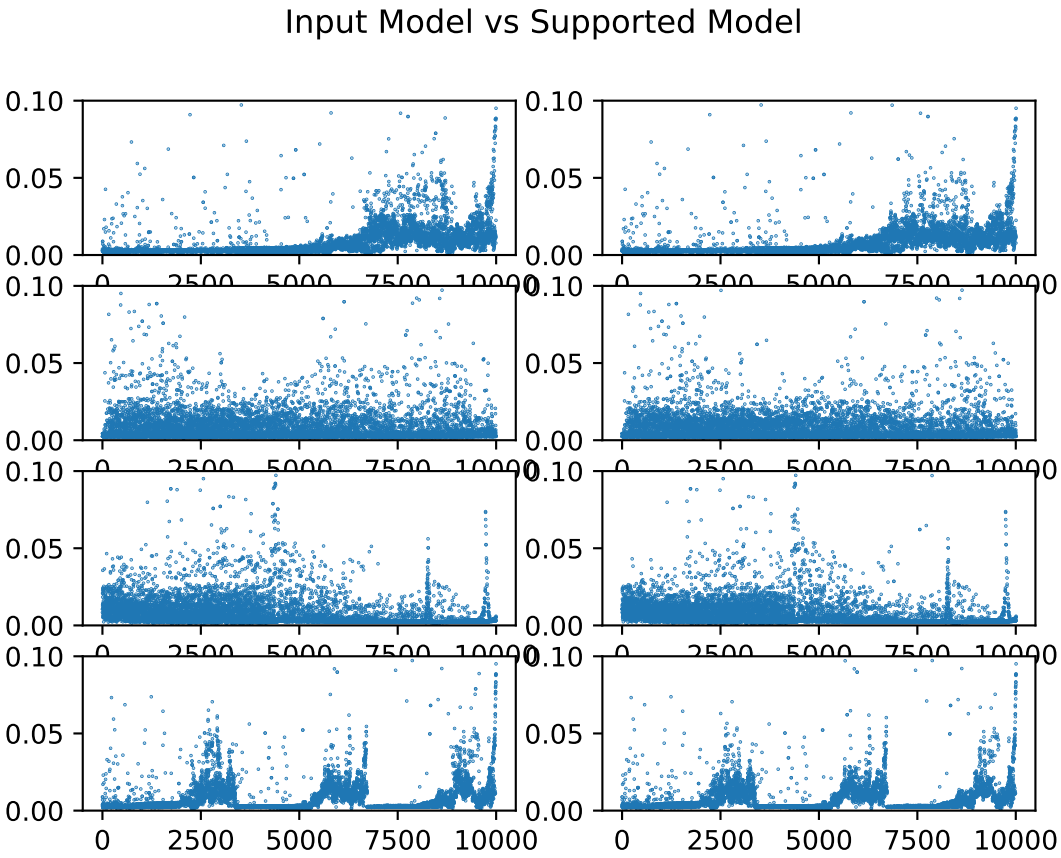

Figure 7.2: left: Structure with regular SD, right: Structure with RSD4

7.1.2 Training methods for the SNNs

I tested the different training methods for the SNNs I described in Chapter 5. For each training method, I created a structure where the SNNs were trained with this method. I did multiple runs for each training method and measured the MAE produced by the structure for each run. After that, I computed the average MAE for each training method. I set $n = 3$ for all test runs to make it easier to compare the individual runs. In a real application, I would only add more networks to the structure if they improve the overall error. To see if this would result in structures with less networks, I also measured the error of the structure after adding each network. Table 7.2 shows the average MAE over all runs for each training method. The columns '1 SNN' and '2 SNN' show the MAE of a structure with one and two SNNs respectively. The column 'MNN' shows the error producey by the MNN and the column 'BTS' shows the error of the final structure with three SNNs. It still holds that there are as much DNNs as SNNs in the structure at any time I measure the error. Note that the MAE produced by the MNNs differ between the training method. This is not caused by the training methods, but the normal variation when training neural networks due to differently chosen training and evaluation data sets for each run.

As mentioned in Chapter 2, the original training method is the one that was implemented for the initial framework from [Man22]. The prediction method is the training method described in Chapter 2. The mixed method is the combination of the prediction method and the cover method. Table 7.3 shows the percentage value of the average MAE produced by the different structures compared to the average MAE produced by the MNN. I compared the average MAE produced by a structure with the average MAE produced by the MNN in runs using that training method.

	MNN	BTS	BTS^2	BTS^1	SDS	RSD4
Original	0,0129	0,0074	0,0076	0,0078	0,0075	0,0076
Prediction	0,0125	0,0072	0,0073	0,0079	0,0074	0,0075
Cover	0,0129	0,007	0,0072	0,0074	0,0071	0,0071
Mixed	0,0135	0,0074	0,0077	0,009	0,0075	0,0077

Table 7.2: MAE for different training methods

The results show that all training method presented in Chapter 2 result in a lower error when adding more SNNs and DNNs. In fact, $n = 3$ is the optimal size for this test case. I tested if the structure gets better if I add even more SNNs and DNNs, but in almost all test runs the addition of a fourth SNN and DNN made the error bigger instead of smaller. The structures created with the original method get consistently more accurate

	BTS	BTS^2	BTS^1	SD	RSD4
Original	57	59	61	57	57
Prediction	58	58	63	59	60
Cover	55	56	57	55	55
Mixed	55	57	60	55	56

Table 7.3: percentage values for different training methods

with every SNN added. The first structure has an average MAE of $0,0078$, which is 61% of the average MAE of the MNN. With $n = 3$ SNNs, the structure improves the accuracy of the MNN by more than 40%. When compared to the original method, the prediction method performs with approximately the same accuracy, but it has a bigger increase for the second SNN. The slight difference in the percentage value is also based on the slightly better performance of the MNN in the runs for the prediction method. The final structure with $n = 3$ pairs of SNNs and DNNs has an average MAE of less than 60% of the MAE of the MNN, meaning it is almost twice as good as the MNN. I can conclude that the prediction method results in a structure that improves the accuracy compared to the MNN a lot. Also it is not remarkably worse than the one created with the original training method.

When using the cover method, the average MAE for BTS^1 is better than for any other training method. Compared to the MNN, it almost halves the MAE with an absolute value of $0,0074$. This value is 57% of the average MAE of the MNN. The optimal amount of SNNs is $n = e$ when using the cover method. The final structure achieves an average MAE that is 45% lower than the average MAE of the MNN. When comparing this value to the values achieved with the original method, the structure created with the cover method has the better average MAE with a value of $0,007$. Since the average MAE of the MNN is the same for both training methods, this shows that the cover method performs slightly better than the original method. Figure 7.3 shows a comparison of the MAE computed by the MNN on the left side and the MAE computed by the SDS using the cover method on the right side. For the graphs in the first row, the data points were sorted after the angle of the arm. The plots illustrate that the MNN's accuracy gets worse as the angle of the arm gets steeper. The structure with the SD can lower the error for those data points with a steep angle remarkably by using SNNs which were trained with data points from that area. There are similar areas in the graph for different orderings as well. Both plots are similar for some areas, e.g. for the first 5000 data points when sorting the data points after the angle of the arm. In other areas, the SDS performs with a higher accuracy than the MNN. One such area are the last 1500 data points when sorting after the first feature. This illustrates the improvement in the areas where the SNNs are specialized in.

The mixed training method achieves the second-best result out of all training methods. For this method, the increase in accuracy with more networks added is bigger than for the cover method. This lead to the same improvement of 45% for both these methods when using a structure with $n = 3$ SNNs when the mixed method was three percentage points worse than the cover method for $n = 1$.

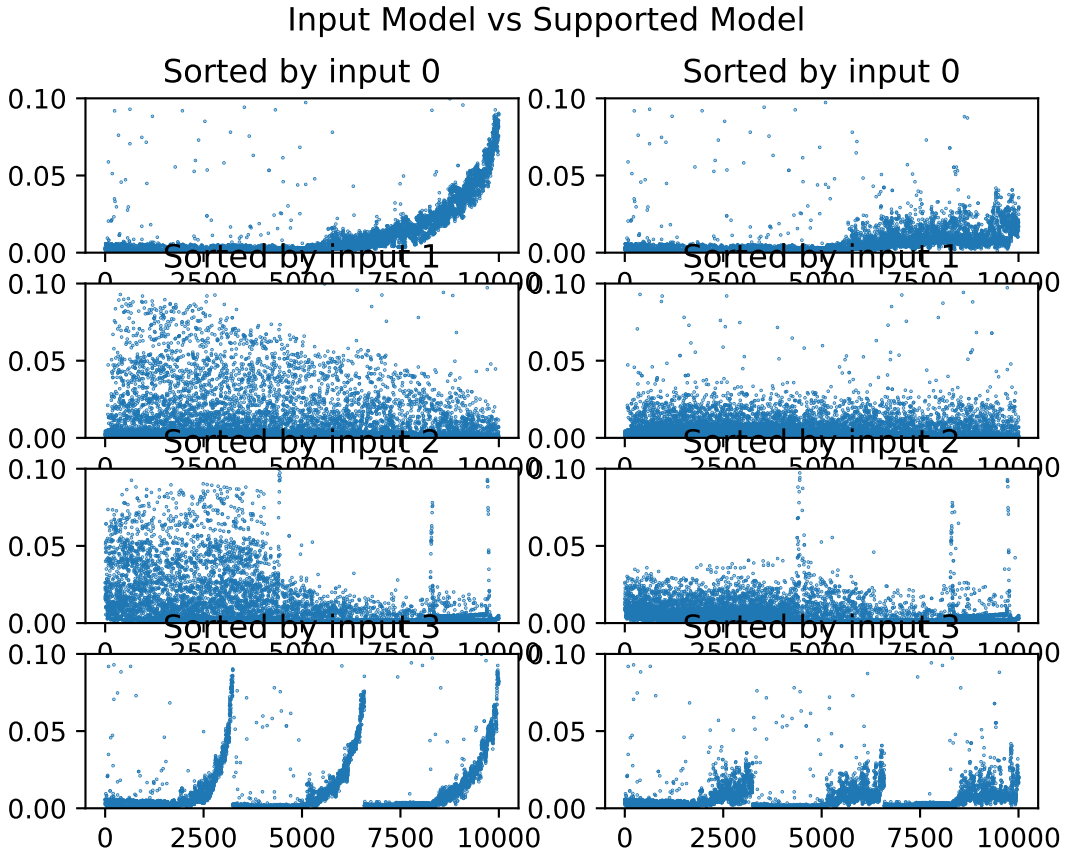

Figure 7.3: left: plot of the MNN, right: plot of the SD for the cover method

7.1.3 Substitution of the MNN

Once again, I did multiple runs for this approach and aggregated the values into one average MAE and one percentage value compared to the accuracy of the MNN alone. The SNNs for this approach were trained with the original training method. Table 7.4 shows the values for the different structures. I show the values for all structures BTS^n with $n \leq 5$ because every structure without MNN had at least five pairs of SNN and DNN. The column 'BTS' shows the values for the final structure. The amount of SNNs in

the final structure varied from five to nine. Note that I deleted the MNN after the third iteration, so the column 'BTS^4' shows the value for the first structure without the MNN.

	MNN	BTS	BTS^5	BTS^4	BTS^3	BTS^2	BTS^1
MAE	0,0119	0,0064	0,0065	0,0077	0,0071	0,0072	0,0075
% of MNN	100	54	55	65	59	60	63

Table 7.4: test case 1: MAE and percentage value for structures without MNN

The values show that the average MAE for the structure BTS^4 is worse than for BTS^3. This is because BTS^4 is the first structure without the MNN, so the fourth SNN has to substitute the MNN compared to BTS^3. I also see that the average MAE gets remarkably better for $n > 4$. The average MAE of BTS^5 is 0,0065. This is not only a better value than BTS^3, which is the most accurate structure that still contains the MNN. With a percentage value of 54% compared to the MNN, it is also better than what the original framework achieved with exactly the same training method. The final structure is slightly better than BTS^5. On average, it contains between six and seven SNNs. Since three SNNs were added before the deletion of the MNN, I need around three or four more SNNs to compensate the missing MNN. With these results, the approach of substituting the MNN with more SNNs is a good alternative. It produces better results for this test case than any structure could do with the original approach of having one MNN and multiple pairs of SNN and DNN. The SD can also be combined with this approach to neglect the downside of the longer inference time due to more binary deciding networks.

7.2 Test Case: Muscle Deformation

This test case uses the data of the muscle activation as input data points. This means I now have $k = 5$ features for all input data points $x_i \in X$. The input is used to compute values for the muscle deformation. These predictions $y_i \in Y$ have a dimension of $l = 30$. The given neural network A has a shape of seven Dense layers sequentially arranged. The output dimension of the layers start with five and increase up to 30 for the last layer. This means that each layer expands the output dimension by a few steps until the dimension of $l = 30$ is reached. I use this shape for all processing networks in the structure. The DNNs are created with the shape described for the first test case in Section 7.1. The SD and RSDs have the shape described in Chapter 6. All runs were made with $t_{min} = 1,85$ and $t_{max} = 2,25$. The stepsize for the linear search between those values was $0,025$. It is worth noting that the given neural network produces a

more consistent error throughout the input data than the neural network for the first test case. The areas where it performs with good accuracy are not as well separated from the areas where it doesn't perform as accurate. This is visible in the plots that will be shown later in this chapter. Once again, all $x_i, y_i \in \mathbb{R}$.

7.2.1 Super Decider and Reduced Super Decider

As for the first test case, I aggregated multiple test runs for different training methods with this test case to evaluate the performance of the structure with the SD. Since the optimal amount of networks in the structure varies more for this test case, I didn't fix the size of the structure as I did for the first test case. Table 7.5 shows the average MAE and the percentage values for the different structures.

	MNN	BTS	SD	RSD1	RSD2	RSD3	RSD4
MAE	1,52	0,9	0,94	0,94	0,94	0,94	0,93
% of MNN	100	59	62	62	62	62	61
inference time in s	1,017	2,546	1,9022	1,464	1,646	1,756	1,6

Table 7.5: test case 2: accuracy of different SD-approach

The values show clearly that the SD as well as all version of the RSD work very well for this test case. While the regular SD has an average MAE that is only 0,004 worse than the average MAE of the structure with the normal DNNs, the RSD4 is even better than the regular SD. All version of the SD, no matter ob regular or reduced, are within three percentage points to the normal BTS-structure when comparing the average MAE to the one of the MNN. The inference time was measured for structures with $n = 3$ SNNs. Remember that for higher values of n, the inference time of the original structure will increase, while the structures with any version of an SD have a constant inference time. The original structure has an average inference time of around $2,5$ seconds. This is approximately 2,5 times higher than the inference time of the MNN with $1,017$ seconds. The regular SD can decrease the inference time to under 2 seconds, which is about 30% faster than the original structure. The improvement with the reduced versions are even higher. The fastest RSD is the first version, as it already was in test case 1. Its inference time of approximately $1,5$ seconds is more than one second and almost 50% faster than the original structure. With that inference time it is not even 50% slower than the MNN alone. The slowest RSD still has an inference time of $1,756$ seconds, which is more than 30% faster than the structure without SD. All versions of SD and RSD result in very similar values. This shows that I can once again use the structure with the SD or even with RSD4 instead of the BTS-structure to speed up the evaluation while barely losing accuracy.

Figure 7.4 shows a comparison between the MAE of a final structure created with the prediction method and the MAE of the structure with SD. In each row, the data points are sorted after one input feature, from the first feature in the first row to the last feature in the last row. The plots show that there is only a negligible difference in the glsmae of the SD compared to the original structure. Figure 7.5 compares the structure using the SD with the structure using RSD1. Once again, there is almost no visible difference between the plots. The plots for the other versions of the RSD look very similar. This supports the conclusion from the absolute values that both the SD and the RSD bring no noteworthy decrease in accuracy compared to the original approach.

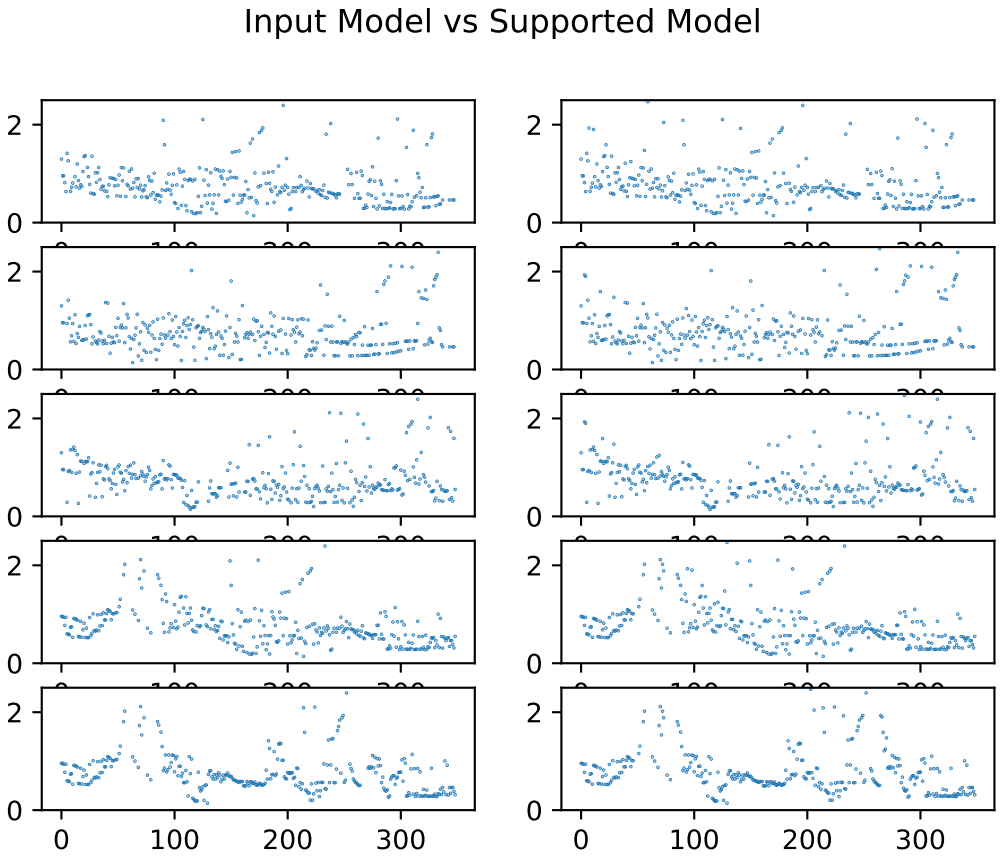

Figure 7.4: left: plot of the final structure, right: plot of the SD

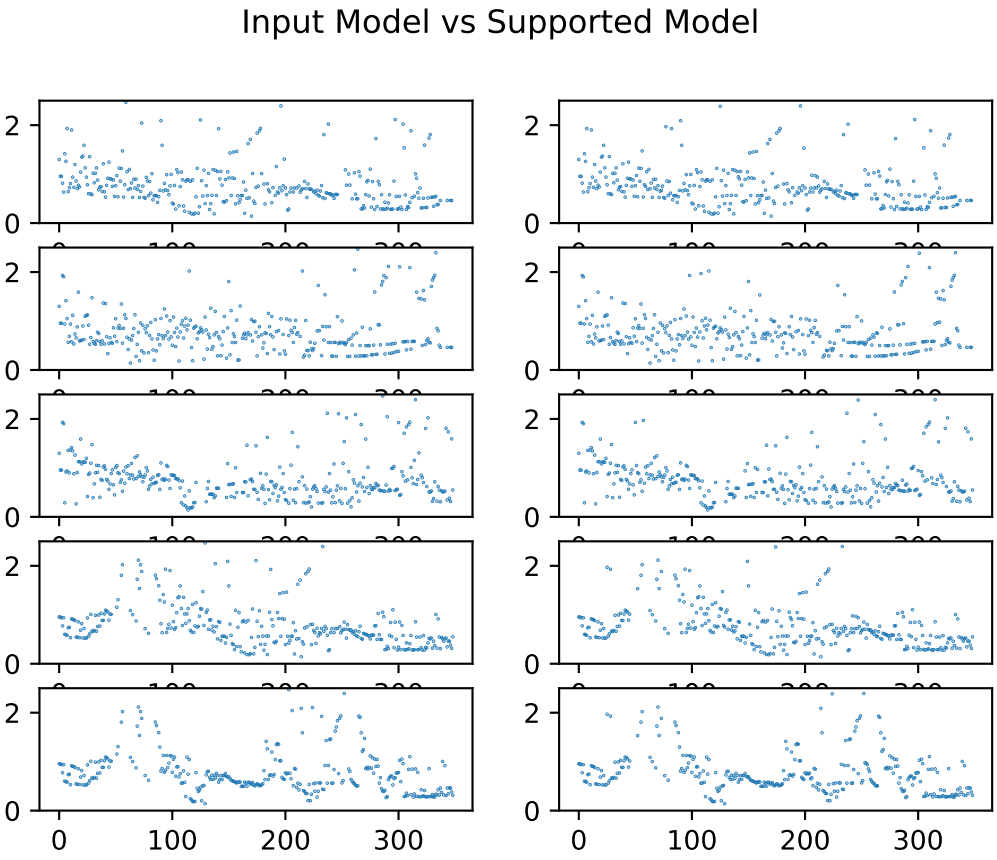

Figure 7.5: left: plot of the regular SD, right: plot of RSD1

7.2.2 Training methods for the SNNs

To evaluate the different training methods, I proceeded as in Section 7.1.2. Once again, I don't fix the amount of networks in the structures. Instead, I always add one pair of SNN and DNN to the structure. After that, I only add more of these pairs when the overall accuracy of the structure gets better by adding a new pair. This works much like adding more networks when substituting the MNN as described in Section 5.6. When evaluating the different training methods for this test case, I also combined the original method and the cover-method to develop the mixed* method. Table 7.6 shows the average MAE for the different structures. The column 'BTS' shows the average value for the whole structure. It still holds that there are as many DNNs as SNNs in a structure with no SD or RSD when measuring the error.
Table 7.7 shows the percentage values of the average MAE produced by structures with different training methods in comparison to the average MAE produced by the MNN.

	MNN	BTS	BTS^3	BTS^2	BTS^1	SD
Original	1,58	1,3	-	1,3	1,33	1,35
Prediction	1,53	0,9	1,12	1,13	1,15	0,95
Cover	1,48	1,78	-	-	1,78	1,85
Mixed	1,57	1,3	-	1,41	1,3	1,31
Mixed*	1,53	0,71	-	0,71	1,24	0,73

Table 7.6: test case 2: MAE for different training methods

	BTS	BTS^3	BTS^2	BTS^1	SD
Original	83	-	83	84	86
Prediction	60	74	74	76	62
Cover	120	-	-	120	124
Mixed	83	-	83	90	83
Mixed*	46	-	46	81	47

Table 7.7: test case 2: percentage values for different training methods

For the original method, all runs only added $n = 2$ pairs of SNN and DNN. I created some structures where I manually forced a third pair. These structures performed worse than the structures with $n = 2$, so I didn't add them into the table. The results show that while the original method is able to improve the accuracy of the MNN, it has not the same effect as it had in test case 1. With an average MAE of $1,3$, the final structure improves the average MAE only by 17%. The prediction method had at least $n = 3$ pairs

of SNN and DNN in every run. The maximum value for n was five. This is the reason why the average MAE for the final structure of the prediction method is lower than for BTS^3. In runs where more than three pairs of SNN and DNN were added, these extra pairs brought a remarkable improvement in accuracy. The final structure for the prediction method has an average MAE of $0,9$. This is 40% lower than the average MAE of the MNN. With this result, the final structure created with the prediction method achieves almost the same improvement as for the first test case, where it was 42% better than the MNN. Unfortunately, the structures created with the cover method were not able to improve the accuracy for this test case. The structures created with the mixed method for this test case never had more than two SNNs. The average MAE of the final structure is $1,3$. This is the exact same value that was achieved with the original method. With this MAE, the MNN is improved by only 17% when using the mixed method. When using the mixed* method, the optimum value for n is 2. The structure BTS^1 produces an average MAE of $1,24$, which is 20% better than the average MAE of the MNN. With the addition of the second SNN, the structure achieves an average MAE of $0,71$. This improves the accuracy of the MNN by almost 60%. With this value, the mixed* method achieves the best results of all training methods for the second test case.

Figure 7.6 shows plots of the MAE from a run with the mixed* method. On the left side, the MAE of the MNN is shown. On the right side, the MAE of the structure with the SD created by using the mixed* method is shown. In each row the x-data is sorted after one input feature, from the first feature in the first row to the fifth feature in the last row. The plots illustrate the improvements achieved with the mixed* method

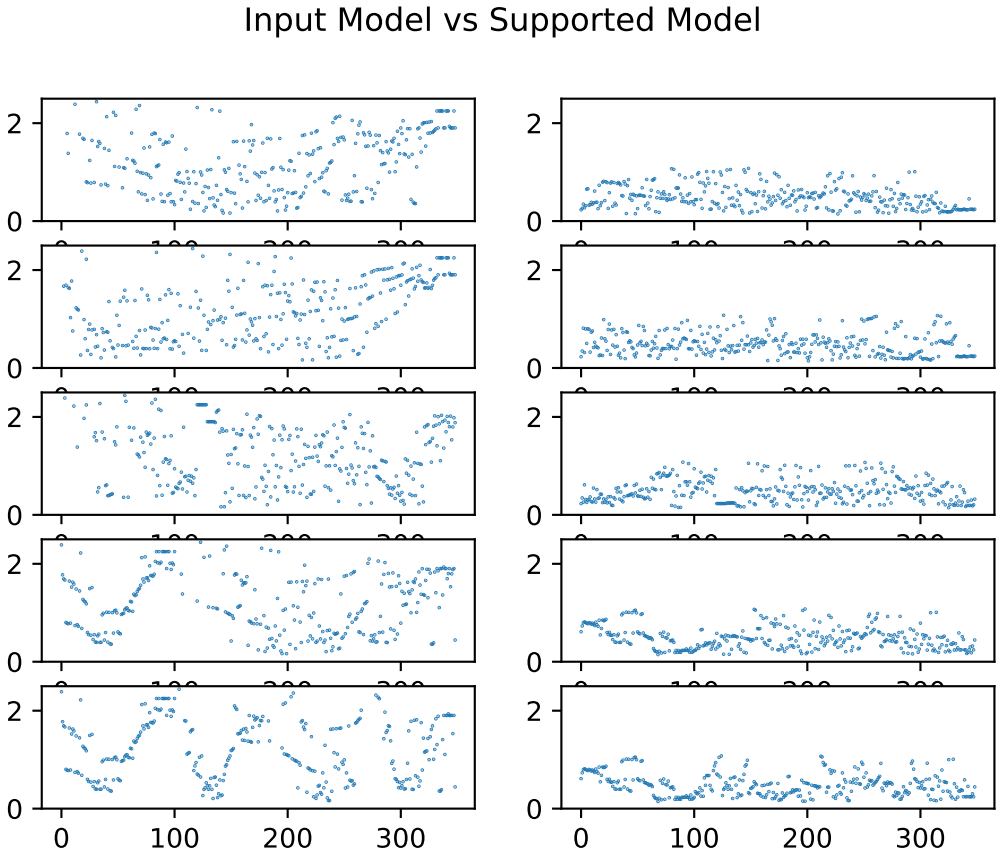

Figure 7.6: left: plot of the MNN, right: plot of the SD for the mixed* method

7.2.3 Substitution of the MNN

For the approach of substituting the MNN with extra SNNs, I created a structure using the prediction method and deleted the MNN after the second iteration. This means that BTS^3 is the first structure without a MNN. Once again, I evaluated this approach by running it multiple times and aggregating the results into one average value.

	MNN	BTS	BTS^3	BTS^2	BTS^1
MAE	1,56	1,08	1,09	1,06	1,06
% of MNN	100	69	70	68	68

Table 7.8: test case2: MAE and percentage value for structures without MNN

The results for this approach show that substituting the MNN with more SNNs does not improve the accuracy when compared to the normal approach. While the accuracy of the final structure is still around 30% better than the accuracy of the MNN alone, the normal approach with the prediction method achieved an improvement of 40%. After deleting the MNN, the structure gets only slightly worse, but the following SNNs can not improve the accuracy enough to achieve the same improvement than the normal approach has with the prediction method.

8 Conclusion

The goal of the work was to develop algorithms to improve a given neural network. The work [Man22] from Daniel Mantsch already did a first step in that direction by creating a framework that improved a given neural network by adding SNNs and DNNs. To improve the results from [Man22], I developed new methods to decrease the inference time of the structure that resulted from the framework or increase the accuracy even more. As I showed in Chapter 7, I achieved both goals. Furthermore, I wanted to show that this approach of improving a given neural network does not only work for the use case presented in [Man22], but also for other neural networks. With the implementation of the framework for the second test case, I were successful in this concern.

The approach of adding a SD performed with a negligibly worse accuracy than the original structure. In return, the structure with the SD needs only around half the time to compute predictions for a set of input data points compared to the original structure with $n = 3$. The runtime for the structure with SD is independent of n, while the original structure has a longer inference time for higher values of n. This enables me to create bigger structures with more SNNs without increasing the inference time of these structures. I can improve this result with the implementation of an RSD. While the accuracy is worsened by 1% compared to the regular SD, the fastest version of the RSD is more than 20% faster than the regular SD. This means that a structure with an RSD enables me to improve the accuracy of the MNN remarkably while also being considerably faster than the initial structure resulting from [Man22]. Considering these results, the implementation of one single deciding network instead of n DNNs was a success. There is no reason to use a binary tree structure containing DNNs instead of using a version of an SD in future implementations.

I also introduced new training methods for the SNNs to improve the accuracy of the structure. The results for these training methods differed between the test cases. For the first test case, which was also used in [Man22], the cover method and the mixed method produced the most accurate structures. The MAE produced by the structures created with these training method was in average 45% better than the MNN alone. Considering the two training methods from [Man22], the original method performed slightly better than the prediction method. Both these methods improved the average MAE of the MNN by around 30%. The results show that I was able to improve the

accuracy of the framework from [Man22] with the introduction of new training methods, although the improvement is not that big. For the second test case, the results were a bit different. First of all, I was able to show that the general approach of enhancing a given neural network with an automated structure consisting of SNNs and DNNs works for different use cases. Combined with the fact that the given neural network in the second test case had differently distributed areas of input values where it did not perform accurate enough, this shows that the approach can be viable for different types of neural networks. For this test case, the new mixed* training method performed best with an improvement of over 50% compared to the MNN alone, followed by the prediction method with an improvement of 40%. The other methods performed worse than for the first test case. This shows that the idea of the mixed training method can achieve good results and be even better than the original training methods, although it was the alternative mixed* approach that brought the best improvements for the second test case. Overall, the introduction of the new training methods improves the accuracy of the original framework.

The last approach is the substitution of the MNN. The goal was to improve the accuracy of the structure even more by adding more specialized neural networks to substitute the MNN. This approach worked slightly better than the original approach for the first test case. Nonetheless it was able to improve the accuracy of the original framework for the first test case. For the second test case, the approach without MNN performed worse than the original approach with the same training method. While the approach without MNN proved it can increase the accuracy of the original framework in some cases, the results for the second test case show that it is not generally superior.

All in all, I found a way to reduce the inference time of a structure as it is described in [Man22] remarkably. Combined with the results from the different training methods for the SNNs and the approach of substituting the MNN with more SNNs, I enhanced the framework from [Man22] both in inference time and in accuracy. Additionally, I increased the flexibility of the framework since different approaches vary in efficiency for different use cases. By introducing new training methods and the possibility of substituting the MNN, it is more likely that the enhanced framework can improve the accuracy of a network in future use cases.

9 Outlook

With all the different approaches I introduce in this work, there are possibilities that sound promising, but that I couldn't test. I briefly explain them in this chapter. First of all, it sounds interesting to create a structure consisting of MNN and n pairs of SNN and DNN where not all SNNs are trained with the same training method. If for example the mixed method produces a very good result for the first SNN and the cover method achieves a consistent increase of accuracy when more SNNs are added, I could train the first SNN with the mixed method and all subsequent SNNs with the cover method to see if the resulting structure achieves an even better accuracy. Another interesting idea is to reduce the processing networks in size. Since the reduction of the neural network achieved good results for the SD, I think it is possible to increase the inference time of a structure even more if I could also reduce the processing networks. The loss of accuracy I expect should be compensable by adding more SNNs. Since the structures with SD don't take more time to compute with more SNNs in the structure, this idea seems promising. Because the mixed* method worked well for the second test case, it might be worth testing out in other use cases too. Concerning the implementation, there are possibilities to improve the framework. One option is to automatically copy the structure of a given neural network instead of coding that structure manually. This way, it would be possible to put in a neural network and the sets X and Y and the framework would automatically enhance the accuracy without the need to adjust the code manually. Another option is to load already trained networks into the structure. I could load the MNN into the structure instead of training it in every run of the framework. I also could save the best-performing SNNs and try to combine all these saved networks into one new structure to see if this structure achieves better results. Both these approaches would reduce the time needed to create the structure.

Bibliography

[AAB+16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. DOI: 10.48550/ARXIV.1603.04467. URL: https://arxiv.org/abs/1603.04467 (cit. on p. 19).

[BEN] O. Ben-Kiki, C. Evans, I. döt Net. *YAML version 1.2*. https://yaml.org/spec/1.2.2/. Accessed: 2022-12-03 (cit. on p. 20).

[Cho+] F. Chollet et al. *Keras*. https://keras.io. Accessed: 2022-12-03 (cit. on p. 19).

[LBG+16] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, F. Kawsar. "DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices." In: *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 2016, pp. 1–12. DOI: 10.1109/IPSN.2016.7460664 (cit. on p. 13).

[Man22] D. Mantsch. "Automated Quality Enhancer for fast Neural Network Inference on Mobile Devices." In: (2022) (cit. on pp. 15, 18, 21, 23, 24, 26, 31, 46, 57, 58).

[TMK17] S. Teerapittayanon, B. McDanel, H. Kung. "Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices." In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 328–339. DOI: 10.1109/ICDCS.2017.226 (cit. on p. 13).

[TMW+18] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, Z. Wang. "Adaptive Deep Learning Model Selection on Embedded Systems." In: *SIGPLAN Not.* 53.6 (June 2018), pp. 31–43. ISSN: 0362-1340. DOI: 10.1145/3299710.3211336. URL: https://doi.org/10.1145/3299710.3211336 (cit. on p. 14).

[XLG13] J. Xue, J. Li, Y. Gong. “Restructuring of Deep Neural Network Acoustic Models with Singular Value Decomposition.” In: *14TH ANNUAL CONFERENCE OF THE INTERNATIONAL SPEECH COMMUNICATION ASSOCIATION (INTERSPEECH 2013), VOLS 1-5*. Ed. by F. Bimbot, C. Cerisara, C. Fougeron, G. Gravier, L. Lamel, F. Pellegrino, P. Perrier. Interspeech. 14th Annual Conference of the International-Speech-Communication-Association (INTERSPEECH 2013), Lyon, FRANCE, AUG 25-29, 2013. Int Speech Commun Assoc; europa org; amazon; Microsoft; Google; TcL SYTRAL; European Language Resources Assoc; ouaero; imaginove; VOCAPIA res; acapela; speech ocean; ALDEBARAN; orange; vecsys; IBM Res; Raytheon BBN Technol; voxygen. 2013, pp. 2364–2368. ISBN: 978-1-62993-443-3 (cit. on p. 13).

All links were last followed on December 03, 2022.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature