Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Surrogate models for black-box optimization problems

Leonard Popp

Course of Study:	B.Sc. Computer Science
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	Peter Domanski
Commenced:	August 28, 2022
Completed:	March 17, 2023

Abstract

Finding the optimum of a black-box function is a challenging task. Optimizing these functions is often time consuming and computationally expensive. In such situations, surrogate models are often used. Surrogate models are mathematical models that approximate the behavior of a black-box function using sample data. There are several works that present methods to optimize these surrogate models. However, the influence of the different surrogate models and their properties on the solution of the underlying optimization task remains unclear. In our work, we focus on a learn-to-optimize approach, where an optimization algorithm is trained using reinforcement learning. We study different surrogate models and analyze their properties. As surrogate models, we use well-known machine learning algorithms such as k-nearest neighbors, decision trees, and deep neural networks. We optimize these models to fit our dataset and compare them using different metrics. We evaluate the models' ability to predict the maximum of our function using Monte Carlo sampling. We then analyze the influence of their different properties in the learn-to-optimize process. Finally, we identify the best model for our task and propose ideas for further improvement based on our observations.

All this is done using real data from performance tuning of semiconductor circuits in post-silicon validation. Performance tuning is a complex optimization task to determine the optimal configuration of various circuit parameters to achieve maximum performance. Semiconductor circuits are used in a wide range of applications, including computing and power electronics. Therefore, their optimization is of general interest, which underlines the importance of the topic.

Contents

1 Introduction **9**

2 Related Work **11**

3 Fundamentals **13**

3.1 Deep Learning 13

3.2 Optimization Theory 16

4 Methodology **23**

4.1 AutoTune 23

4.2 Important Libraries 24

4.3 Analysis Design 25

5 Experiments **31**

5.1 Comparison of models without tuning 31

5.2 Hyperparameter Optimization 33

5.3 Comparison of models with tuning 50

5.4 Surrogate-based Monte Carlo Sampling 53

5.5 Learn-to-Optimize 58

6 Future Work **61**

7 Conclusion **63**

Bibliography **65**

Acronyms

DNN Deep Neural Network. 15

DT Decision Tree. 18

ET Extra Trees. 19

FOM Figure of Merit. 23

GBT Gradient Boosting Tree. 19

HSIO high-speed input/output. 11

KNN K-Nearest Neighbours. 18

MLP Multi-layer Perceptron. 13

MSE Mean Squared Error. 21

NN Neural Network. 13

PSV Post-Silicon Validation. 9

R^2-score Coefficient of determination. 25

RBF Radial Basis Function. 11

ReLU Rectified Linear Unit. 20

RF Random Forest. 19

RL Reinforcement Learning. 23

SGD Stochastic Gradient Descent. 19

Sk-learn Scikit-learn. 17

SSR Subset Selection Regression. 11

SVM Support Vector Machine. 11

TF TensorFlow. 24

1 Introduction

Optimization algorithms are of great importance for current developments in many scientific domains. However, we do not always know a closed-form expression of the objective we like to optimize. In these cases, a gradient is difficult and costly to obtain. Furthermore, if the objective function is non-differentiable, derivative-based algorithms such as gradient descent cannot be applied. To optimize those functions, so called zero-order algorithms exist which only require evaluations at arbitrary points of the function. Sometimes the evaluation of the black-box function can be computational expensive or even unfeasible if we only have access to sampling data of the objective. One approach to those black-box optimization problems is surrogate-based modeling and optimization. Surrogate models are trained on the given sampling data and approximate the objective using machine learning algorithms. They try to mimic the behaviour as close as possible with the advantage of being computationally cheaper to evaluate. Once a surrogate model is accurately trained, it can be used as an replacement for the true optimization objective with the benefit of less costly evaluations. The optimization algorithm then utilizes the surrogate model for function evaluations at specific points and the search for the optimum value. However, the found solution and the optimization process therefore depend on the different surrogates and their individual properties.

A real-world application, including complex optimization tasks, is the performance tuning of semiconductor circuits in Post-Silicon Validation (PSV). It is a very complex optimization task to identify a optimal configuration of the circuits parameters in varying conditions reaching maximum performance and pass the specification goals of circuits. As a semiconductor circuit is a black-box function and computational expensive to evaluate, surrogate models often are necessary or already in use. Designing a surrogate model in this context can be an interesting challenge as the data consists of measurements from different circuits and features with various data types including integer, categorical and continuous values. Our research focuses on an already existing project called AutoTune [Dom22; DPLR21; DPRL21] which uses a learn-to-optimize approach to solve the optimization task. However, it is unclear how the different surrogate models and their properties influence the learn-to-optimize process.

In our research, surrogate models are indispensable as our solution works with a data set of circuit measurements. We will study different machine learning models and their properties. We will optimize them by hyperparameter optimization for our specific problem. Further, we will explore the trade-off between different properties like model complexity, quality of approximation and time consumption. Then we will study the influence of the different properties on the training process of learn-to-optimize, asking the question if different surrogate models will lead to faster convergence in the training process or different solutions. Finally, we will suggest potential surrogate models tailored specific to our use case and the scenario of learn-to-optimize.

2 Related Work

Using surrogate models in black-box optimization has been object of different research papers. In [VDHL17] Vu et al. survey methods currently used in black-box optimization, concentrating specifically on surrogate-based methods. Those methods appear to be very successful at finding global solutions. They describe the common approach of iteratively constructing and improving surrogate models using evaluations on points of the black-box function determined by the previous surrogate. In comparison to our approach/scenario they are able to evaluate the black-box function to further improve their surrogates. Our surrogate models will be solely based on our dataset as we cannot access the devices anymore after the initial data generation using automatic test equipment. Furthermore, they present some of the most popular models polynomials, Radial Basis Functions (RBFs), Kriging and Support Vector Machines (SVMs). They introduce different merit functions, which are used to decide which point(s) should be evaluated next and which areas should be searched. Summarized, their approach focuses on iteratively creating surrogate models with as few as possible evaluations. These surrogate models are designed to only approximate the black-box well in the neighbourhoods of optima.
In [CSM14] Cozad et al. propose a new methodology for developing accurate surrogates with reduced difficulty and complexity to improve the tractability of the optimization model. They focus on surrogates with the final purpose of algebraic optimization and therefore aim on finding surrogates with smaller and compact forms opposed to the common idea of developing models that are as accurate as possible disregarding their resulting complex functional form. Their methodology uses a linear combination of simple basis function forms like polynomial or multinomial to model the unknown functional forms with a lower complexity. Further, they use a reformulation of the best subset method to find the most accurate model while avoiding the costly combinatorics of full enumeration. Finally, they implemented the proposed methodology in a software package called ALAMO.
In [KB20] Kim et al. again focus on the trade-off between surrogate complexity and accuracy. Popular methods such as Kriging and RBFs have been successfully applied in many areas but tend to be rather complex and cannot be easily globally optimized. Their approach uses Subset Selection Regression (SSR) techniques to balance both accuracy and complexity of the surrogate functions. This involves selecting a subset with only the most informative features to generate a sparse model. The elimination of informative features can lead to an increase in prediction accuracy while the reducing of the number of variables can lower the computational cost. They compare several SSR algorithms and Kriging on constrained and unconstrained benchmark problems. They conclude that SSR algorithms show promising performance, especially for low-dimensional problems, degrading as the dimension of the problem increases. In addition, the Kriging-based approach was able to solve the most number of problems while requiring significantly longer computational time to optimize the model and the least computational time for parameter estimation.
In [RCV+17] Rangel Patiño et al. propose surrogate-based optimization as an improvement to the very time-consuming process of physical layer tuning in high-speed input/output (HSIO) links. They analyze different surrogate modelling techniques, like Kriging and SVMs, and design of experiments

to find the best optimizing approach for a receiver equalizer. Furthermore, they validate the model performances with actual measurements to verify the accuracy of the models and compare the models to find the most accurate surrogate model. Finally, they perform an optimization on an HSIO link using their best surrogate model to get the optimal tuning settings. Their results demonstrate the high effectiveness of their approach.

In [LM16] Li et al. present a method using reinforcement learning to learn an optimization algorithm. This method is very similar to the approach AutoTune uses. They aim on automating the design of optimization algorithms to outperform the algorithms designed by human experts. They later show through comparison that their autonomous optimizer method converges faster and reaches better optima than hand-engineered optimizers.

In [DPRL21] Domanski et al. propose a methodology for a learn-to-optimize approach in PSV. This methodology was implemented in the software module called AutoTune [Dom22], which is the software module for our experiments on the influence of surrogate models on solving the optimization task. The package will be further explained in the later Section 4.1.

3 Fundamentals

3.1 Deep Learning

3.1.1 Neural Networks (NNs)

NNs are a subset of machine learning algorithms. They have become increasingly popular in recent years due to their ability to model complex, non-linear relationships between input and output variables. NNs are trained on large amounts of data resulting in a model able to afterwards approximate functions computationally cheap. This perfectly fits to our scenario of building an approximation based on sampling data. In our experiments we will use a subclass of NNs called multi-layer perceptrons as surrogates.

Multi-layer Perceptron (MLP)

MLPs consist of one input layer, one or more hidden layers and an output layer, each comprised of different number of artificial neurons. Each neuron is connected to each neuron of the next layer resulting in a densely connected graph. Each of those connections is then associated with a weight w_i and each neuron with a bias b. To predict a value the input values are each assigned to one input neuron. Each neuron of the next layer then determines its output by multiplying the inputs from the previous layer x_i with their respective weights and then summing them. To add non-linearity to the network the sum is then passed through an activation function $a(x)$, which determines the output of the neuron. The calculation can be seen in Equation (3.1) with n being the number of neurons from the previous layer.

$$\text{output} = a(\sum_{i=1}^{n}(w_i \cdot x_i)) \tag{3.1}$$

The output data is then passed to the next layer as input data, and the process is iterated until the final output layer is reached. Hence the name feedforward network. Example MLPs with illustrations can be seen in Section 3.2.2.

To change the output or prediction of the MLP to fit our function, we need to adjust the weights and biases of the model. This is done by feeding the network a sample from our training data and calculating the error between the predicted output and the desired output, using a pre-defined loss function. Using backpropagation, the error is then propagated backwards through the network adjusting the weights and biases based on the error using an optimization algorithm, like gradient descent. This process is repeated multiple times with different training samples, updating the weights and biases, until the error is minimized.

3.1.2 Training

Training, Test and Validation Accuracy

Training a machine learning algorithm is the process of using a set of examples, called the training set, to fit the parameters of the model. These parameters highly depend on the different algorithms as each algorithm is built, trains and predicts different. A models score on the training set is called training score. To evaluate a model a testing set (also hold-out set) is used, which is disjoint to the training set, and therefore consists of samples the model has never seen before. Our goal in machine learning is to create a model that is able to generalize our problem and capture the underlying pattern of our data. Therefore, the test score evaluates how the model would perform on unseen data while a high training score could result of the model being could at memorizing the data and not at generalizing it. In some scenarios a third set called validation set, which is disjoint to the training and test set, is used. It is used in hyperparameter optimization or early stopping as a method to validate the model and compare different models. As the validation set therefore has an influence on the choice of models the testing set is needed as a final evaluation. In early stopping for example the training is stopped depending on the model's performance on the validation set. This can lead to some overfitting to the validation set as the model is optimized for a good performance on it.

Overfitting

Overfitting is a concept in machine learning, which occurs when a statistical model has a notably higher training accuracy than test accuracy. As the test accuracy represents the estimation quality of the model on unseen data, this means that the model is not able to generalize the data very well. It occurs when the model captures the training data's noise or irrelevant information instead of the underlying pattern, resulting in an overly complex model that performs well on the training set but poorly on the test set. The model then fits too closely to the training set and becomes "overfitted". Ideally, a model should perform equally well on both seen and unseen data and therefore have the same training accuracy and test accuracy. Overfitting can result from too long training or the nature of the model. Various methods can be used to mitigate overfitting, such as early stopping or ensemble techniques.

Cross-validation

Cross-validation is a technique aiming on improving model validation by providing a more accurate estimate of model prediction performance. It is used in a prediction problem where a model is trained on a dataset of known data (training set) and tested on a dataset of unknown data (testing set). Because those datasets are often sampled at random, the data the model is fit on and the data the model is evaluated varies. This can lead to small variations in the models performance. Cross-validation tries to improve the reliability of the test score by running multiple fits and evaluations. The dataset is divided for each iteration into different portions of testing and training data and averaging the test score measures. We are using a method named k-fold cross validation which is explained in the following steps:

1. Split the dataset into K equal folds
2. Use one fold as the testing set and the union of the other folds as training set. This ensures that there are no intersections between the testing set and the training set.
3. Train the model and measure the testing accuracy
4. Repeat steps 2 and 3 K times with each subsample used exactly once as the testing set
5. Use the average testing accuracy as the estimate of model prediction performance

Early Stopping

In machine learning, early stopping is a method to mitigate overfitting by interrupting the training process of iterative training methods before the model starts to get overly complex. During iterative training methods the model is updated each iteration to better fit the training data. Up to a point, this generally improves the model's performance on unseen data outside of the training set as the model gets better at generalizing the problem. However, past that point training the model further on the dataset can lead to a worse test score as the model starts to capture and remember noise. Generally we want to stop training at this point to have a model performing well on the test dataset. Early stopping uses the validation dataset to score the model each iteration and to stop training once the validation score has not been improving for a specified number of epochs. Model checkpoints can be further used to avoid overfitting by saving the models parameters/weights each iteration and loading the parameters of the model with the best validation score after training.

3.1.3 Hyperparameter Optimization

Hyperparameter optimization is the problem of choosing the optimal hyperparameters for a machine learning algorithm regarding a specific use case. Hyperparameters can be thought of as the configuration of a machine learning model. Depending on the model, they are used to control the learning process, the architecture or behaviour of the model. However, they are set before the training process and cannot be derived from the training data. Examples for hyperparameters of Deep Neural Networks (DNNs) are the number of hidden layers, the activation functions or the learning rate. By contrast, the values of other parameters, like weights or biases, are derived during training. Especially deep neural networks heavily depend on a wide set of different hyperparameters controlling the networks's architecture, generalization and optimization [FH19]. Hyperparameter optimization returns a set of hyperparameters corresponding to the model with the minimal value of a predefined loss function. Depending on our efforts in hyperparameter optimization, this is the best model we have found at solving the underlying machine learning task. Cross-validation (Section 3.1.2) is used to provide a more accurate estimate of the model's performance. In this specific case, cross-validation usually tests the selected model afterwards with a unbiased split of the dataset. Many machine learning algorithms are configurable and have numerous hyperparameters. Therefore hyperparameter optimization quickly faces the curse of dimensionality problem with large sets of hyperparameters. In most cases, an exhaustive optimization is impossible and we need to balance between available resources, computational time and the final performance. For this reason, several hyperparameter optimization algorithms have been developed and used to address this tradeoff. Some of the hyperparameter optimization algorithms used in this thesis are Grid Search, Random Search, Bayesian Optimization, Successive Halving, and Hyperband.

3.2 Optimization Theory

3.2.1 Overview of most common (zeroth order) optimization algorithms

Grid Search

Grid Search is the most straightforward algorithm of the presented ones. It is an exhaustive search through all possible combinations of pre-defined parameters and returns the best combination. Alternatively it is also possible to provide a fixed number of iterations or a grid size specifying a grid with uniform spacing over the different parameters. The advantages of grid search are its simplicity and its ability to guarantee finding the global optimum in scenarios where all possible parameters are checked. Grid Search brings the advantage of being easy parallelizable due to its simplicity. However, it quickly suffers from the curse of dimensionality and becomes unfeasible in scenarios with too many possible configurations. In such situations grid search either is extremely unpractical or needs guided assistance limiting the parameters to specific search areas.

Random Search

Random Search is arguably even more simple than grid search. It searches through a fixed number of random parameter combinations and returns the best combination. The greatest advantage of random search is its simplicity which makes the algorithm really easy to parallelize and to use. The only parameter needed to start a random search is its number of iterations and the algorithm can be run on problems independent of their dimensionality and search space size. Although, due to its random nature it is not able to guarantee a optimal solution opposed to grid search. Therefore, for a long time random search was seen as inferior to grid search. But as Bergstra et al. have shown in [BB12] random search can be superior in different scenarios due to its flexibility and its randomness. For problems with high dimensionality, where searching all possible configurations is unfeasible, grid search uses a uniformly spread grid and therefore often fails at finding a good optimum. According to Bergstra et al. this grid search failure is the rule rather than the exception in high dimensional hyperparameter optimization [BB12]. This especially occurs when different parameters aren't as important to tune as others. They provide a simple probabilistic explanation for why random search with 60 trials is likely to find the close-to-optimal region of hyperparameters, assuming it occupies at least 5% of the grid surface. This probability can be calculated using the Equation (3.2), where n is the number of random draws, and solved for n to determine the number of draws needed to achieve at least a 95% probability of success. This calculation can be done with different occupation percentages of the close-to-optimal regions and different probabilities of success. It will be used to determine the number of iterations for our later random search.

$$1 - (1 - 0.05)^n > 0.95 \tag{3.2}$$

Bayesian Optimization

Bayesian Optimization is a strategy for global optimization of unknown objective functions also called black-box functions. The analytical expression of these function is unknown. Therefore, these continuous functions are typically difficult to evaluate because their derivatives are very costly to evaluate or not accessible at all. The Bayesian strategy is to place a prior probability distribution (prior) over the function which records our beliefs about the behaviour of the function and use function evaluations to update our beliefs continuously. In more detail, a few initial samples are used to update the prior and form the posterior distribution. Using the posterior distribution, an acquisition function is constructed. This acquisition function acts like a guide, determining the next optimal query point to evaluate. Acquisition functions balance the exploration-exploitation trade-off by balancing between exploring unexplored regions of the input space and exploiting areas where the function is expected to perform well. The black-box function is then evaluated at the query point and the resulting sample is used to update our prior. This process is repeated for a given number of iterations. In our case Gaussian process regression is used to update the prior and define the posterior distribution over the function.
Hyperparameter optimization can be seen as such a black-box optimization problem. Therefore, we are using Bayesian Optimization with Gaussian Process as a Hyperparameter Optimization strategy implemented by KerasTuner [OBL+19] as it has been shown to obtain better results compared to other strategies like Grid and Random Search.

Successive Halving & Hyperband Algorithm

Successive Halving is an algorithm aiming to improve the performance of hyperparameter optimization by only focusing on the most promising parameters [JT16]. It starts with a large set of hyperparameter configurations, allocates a relatively low amount of resources and evaluates their performance. Then it removes the half of the configurations with the worst performances and repeats the process until only a single configuration remains. In each iteration, the algorithm allocates exponentially more resources to the configurations. The Scikit-learn (Sk-learn) implementation specifically does not half the parameters and instead takes a proportion, determined by the parameter "factor", of candidates that are selected for each subsequent iteration.

Hyperband extends the Successive Halving algorithm and calls it as a subroutine [LJD+17; LJD+18]. Hyperband focuses on the trade-off between multiple configurations with a shorter training times and a smaller number of configurations with a longer training times. It aims on speeding up random search through adaptive resource allocation.

We are using both algorithms for the Hyperparameter Optimization in Section 5.2.

Monte Carlo Sampling

Monte Carlo algorithms form a broad class of computational techniques that rely on repeated random sampling to obtain numerical results. In the context of optimization, such algorithms use repeated random samplings to find an optimal solution for a black-box function. The algorithm operates as follows: for a given number of trials, a set of samples is randomly drawn from the space of possible parameter combinations. For each trial, the sample with the best performance is saved.

After all trials, the best sample from each trial is compared to all others, and the best overall sample is returned as the solution. An alternative way to describe this algorithm is as a set of random searches, where the best solution across all searches is returned. While Monte Carlo algorithms are flexible and applicable to a wide range of problems, they can be computationally expensive and require a large number of samples to obtain accurate results.

3.2.2 Surrogate models in optimization

K-Nearest Neighbours (KNN)

The K nearest neighbour regressor is one of the more simple Sk-learn models. The training phase only consists of storing all the feature vectors and labels of the training samples. In the prediction phase the model calculates the weighted average of the k nearest training samples to the query point, k being a integer-valued hyperparameter of the model. Weights can be uniformly distributed or based on the distance to the query point. The distance is calculated using different algorithms like the Euclidean distance, Manhattan distance or Minkowski distance. The best choice of distance metric can differ depending on the problem.
The greatest advantage of KNN is its simplicity. It has a extremely short training time and is easy to hyperparameter optimize as the model mostly relies on the k parameter. However, the model takes very long to predict values and requires high memory depending on the number of samples used to train the model and their dimensionality. Furthermore, the model is sensitive to the scale of the data, irrelevant features and the feature data types. Due to the distance calculation of the model, it is not as suitable for some data types as other models. For example, for categorical features, it is often difficult to compute a useful distance.

Decision Tree (DT)

The DT regressor is another simple Sk-learn regression model. It builds a tree of decision nodes each having a condition dividing different training samples into sub-trees. The goal is to create a model that predicts a target variable based on simple decision rules deduced from the training data. To predict a target variable the algorithm starts at root node and based on the condition results of each node follows the path until it ends up in a leaf node. The algorithm then returns the mean of the target values of the samples in the leaf node.
Due to the simplicity of the DT model it generally performs very fast at training and predicting of data. It is an easy to use model, requires little data preparation and is configurable. However, they generally tend to not generalize the data well and overfit. But different hyperparameters can be used to avoid/minimize this behaviour. Sk-learn's default values for the parameters lead to fully grown and overfitting trees. Another disadvantage of DTs is that they can be unstable because only small variations in different training data sets can result in completely different trees. Even when using a best split strategy features are randomly permuted at each split. Therefore in our experiments we always choose the same random state of our DTs to ensure deterministic behaviour and a fair comparison. This problem is further mitigated by ensembles of DTs presented in Section 3.2.2.

Random Forest (RF), Extra Trees (ET) & Gradient Boosting Tree (GBT)

The RF model is a ensemble method consisting of multiple DTs. During training the model trains multiple DTs on either the whole training set or each on a random sample from the training set, depending on the hyperparameters of the model. The prediction of the RF model is given as the averaged prediction of the individual DTs.
The ET model works similar to the RF model with a minor difference in splitting nodes. It chooses a random split point for each feature while the RF model chooses the optimum split. This results in the ET model usually having a reduced variance and training time but a slightly higher bias. Additionally, the ET model doesn't use bootstrap samples for training per default while RF uses them. This again has the effect of reducing the variance and training time, sometimes at the cost of a slight increase in bias. Generally the reduce in variance is significant therefore yielding a better model. In practice both models often are better at generalizing the problem hence outperforming the standard DT model.

GBT is another Sk-learn ensemble method based on DTs. As well as the other methods GBT tries to train multiple weaker DT models and combines them to get better performance as a whole. Compared to the training of the previous ensemble models, machine learning boosting methods work iteratively to create an ensemble. It starts by fitting an initial model to the data. Then each iteration a new model is built that focuses on accurately predicting the spaces where the combined ensemble of all previous models performs poorly. The model is then combined with all previous models and expected perform better. Gradient boosting specifically uses the gradient of the prediction error. The process is further described in the seminal work [Fri01].
Instead of the standard Gradient Booster from Sk-learn we are using the histogram based estimator from the class `HistGradientBoostingRegressor` as it performs way faster on datasets with more than ten thousand samples.

Stochastic Gradient Descent (SGD)

SGD is a widely used optimization algorithm in machine learning, particularly in training deep neural network. In our case we refer to the `SGDRegressor` from Sk-learn. The model uses the stochastic gradient descent algorithm to train a linear regression model. SGD iteratively adjusts the model's parameters based on the error computed between the predicted values and the true values, also called loss.
The advantage of the SGD model is its efficiency and performance. It generally has a low training and prediction time, and is well suited for regression problems with a large amount of training data. However, due to its linearity it is not capable of learning complex tasks without a linear relationship. Additionally linear regression models are generally sensitive to outliers.

DNN

We have already established the fundamentals of neural networks for our purpose in Section 3.1.1. In the following we will show the architecture of two DNNs as an example.
Standard DNN: Our standard DNN consists of a input layer, two hidden layers and an output layer. An illustration can be seen in Figure 3.1. The first hidden layer has 64 neurons while the second hidden layer has 32 neurons and both are densely connected.

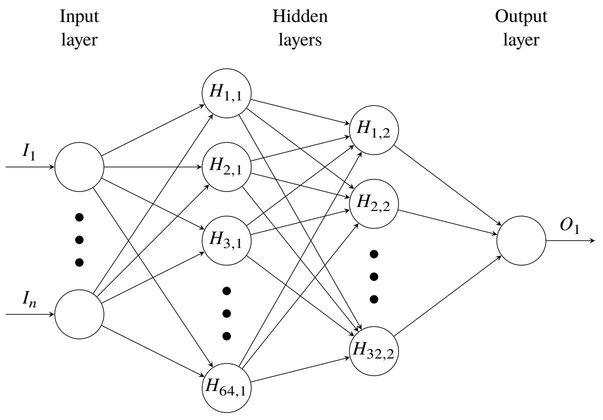


Figure 3.1: An illustration of the standard DNN model

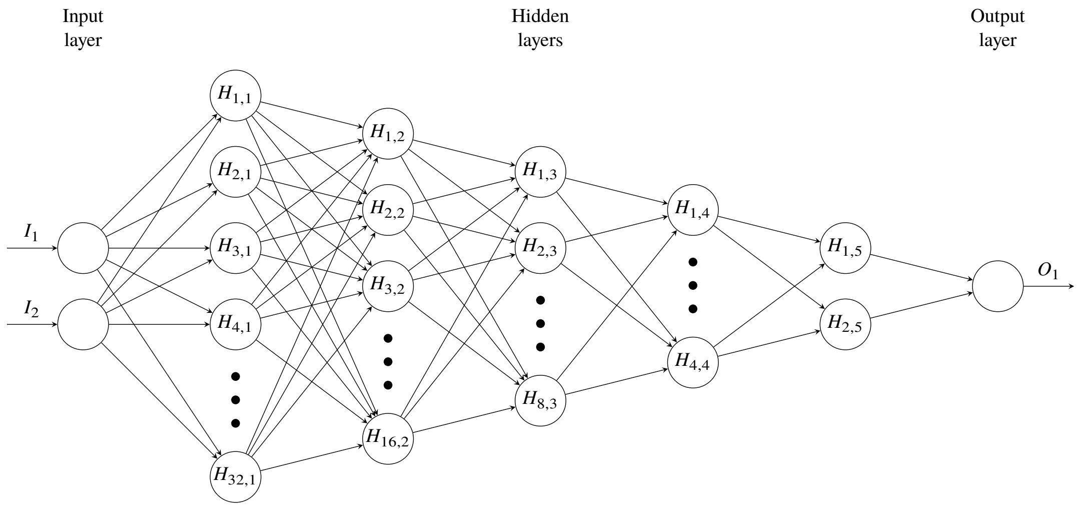


Figure 3.2: An illustration of the PSV DNN model with 2 input features

PSV DNN: The PSV DNN model is a little more complex. Depending on the number of input features it creates a DNN in a pyramid form while halving the number of neurons each hidden layer. The depth of the model is calculated as shown in Equation (3.4) with x being the number of input features. Figure 3.2 shows an illustration of the model with 2 input features.

$$(3.3)\quad \lceil \log_2(4 \cdot x + 3 \cdot x + 2 \cdot x + x + \frac{1}{2} \cdot x + 1) \rceil$$

As we explained in Section 3.1.1 the DNNs rely on a activation function and the training relies on a loss function and an optimizer. In practice, there are many different activation functions available, but in our experiments we will use the Rectified Linear Unit (ReLU) activation function for all of our hidden layers, independent of the model. The ReLU activation function has the advantage of being computationally cheap due to its simplicity, while still being effective. It can be expressed mathematically as follows for any given scalar input value x:

$$(3.4)\quad \mathrm{ReLU}(x) = \max(0, x)$$

For our output layer it is common to use a linear activation function for regression problems.
As loss function we will exclusively use the Mean Squared Error (MSE) loss function for all of our models and as optimizer we will use the Adam optimizer. The Adam optimizer is a gradient descent optimization algorithm and one of the most popular optimization algorithms for training deep neural networks due to its ability to converge quickly with high accuracy on a wide range of problems.
The greatest advantage of DNNs is their ability to model complex nonlinear relationships between inputs and outputs therefore being a well addition to our set of surrogate models. Furthermore they can learn from large amounts of data, are highly configurable and can handle noisy or incomplete data. However, they tend to be computationally more expensive than the other machine learning algorithms and are more difficult to train and optimize due to the many options in configuration and training.

4 Methodology

This thesis focuses on exploring different surrogate models in the setting of learn-to-optimize using the module AutoTune. Different surrogate models with different properties might have an impact on the training process of Reinforcement Learning (RL) in terms of better performance, faster convergence or different solutions/optima. We want to evaluate the impact of surrogate models with different properties. First, we implement different kind of models and perform a first (experimental) study to identify the most promising model types. In a second study, we use hyperparameter optimization to identify the most promising models for the subsequent RL task / optimization task. Then we will compare the models according to different criteria and choose the most promising and interesting ones. Those models will be further examined by running the optimization task and comparing the results. The final goal will be to suggest potential changes to a surrogate model, specifically tailored to the setting of learn-to-optimize, based on the results of our research.

4.1 AutoTune

AutoTune is a library that proposes a methodology for performance tuning of semiconductor circuits in PSV. PSV is a long and complex task in semiconductor testing. It takes a lot of time to try different combinations of parameters to find the optimal parameters for a given chip. Considering the number of chips to be validated and the number of parameters a chip can have, it easily becomes a very tedious problem.
AutoTune works with a dataset of circuit measurements and uses a learn-to-optimize approach based on RL combined with surrogate models to train a machine learning model that predicts the optimal configuration of a circuit under given circumstances. It aims to accelerate performance tuning by sacrificing the initial time to train the NN, with the advantage of predicting the optimal parameters and performance score (Figure of Merit (FOM)) faster and with less computational effort. Sacrificing this initial time is possible because we have enough time in PSV. Furthermore, this approach allows us to do on-chip tuning in near real time, since the resulting NN is fast and relatively small, making it executable on small devices. To train the NN, the reward needs to be calculated frequently to evaluate quality of the RL machine learning model. However, the circuits act as a black-box function making frequent measurements at arbitrary points very expensive if not infeasible. Due to this black-box behaviour, we do not know a closed-form expression of the circuits performance. Therefore, AutoTune uses surrogate models to approximate the black-box function. These models are trained using a dataset with circuit measurements and are less costly to evaluate.

This thesis focuses on this intersection between surrogate models and the learn-to-optimize task. To evaluate a surrogate model in the learn-to-optimize scenario, we first fit the model to the dataset. The model is then used in a RL loop to train the NN. Depending on the circumstances of the experiment, the NN then predicts the optimal configuration and the corresponding maximum performance score. These results are then evaluated.

The AutoTune library provides end-to-end solutions to the task of robust performance tuning in PSV. It is flexible (different datasets, feature types, surrogate models), easy to use (highly configurable, can be used with few lines of code for different applications), and fully automated. It works with dataframes from pandas[tea22] and models from Sk-learn[PVG+11] and TensorFlow (TF)[MAP+15].

4.2 Important Libraries

pandas:

A package we use for our data structure and data processing. It provides many helpful functions dealing with large datasets and for data manipulation[tea22].

Sk-learn:

Efficient algorithms and useful helper functions for machine learning. It includes a wide variety of classification, regression and clustering algorithms and machine learning models, regression being specifically important for this research. Example algorithms used are kNN, random forest and Decision Trees[PVG+11].

TF:

A complex interface used for building and executing machine learning algorithms. TF provides an easy to use high-level API, called Keras, for building Deep Neural Networks which became more important as the importance of machine learning increases. It is wide-spread and therefore many libraries exist that extend the functionality and make more specific tasks like hyperparameter tuning easy[MAP+15].

KerasTuner:

An easy-to-use framework for the hyperparameter optimization of TF models. It is part of the Keras API and comes with Random Search, Bayesian Optimization (Section 3.2.1) and Hyperband (Section 3.2.1) algorithms[OBL+19].

Machine Learning Algorithm	Library
KNN	Sk-learn
DT	Sk-learn
SGD (Histogram based)	Sk-learn
GBT	Sk-learn
RF	Sk-learn
ET	Sk-learn
default DNN	TF (see Section 3.2.2)
PSV DNN	TF (see Section 3.2.2)

Table 4.1: All regression algorithms used in later experiments and their corresponding library

4.3 Analysis Design

In the following, we analyze the machine learning methods / algorithms given in Table 4.1. To analyze the impact of surrogate models on the RL training process we implemented a broad variety of machine learning algorithms. Those algorithms have different properties and therefore can have a different impact on the training process of RL.
Kriging, also known as gaussian process regression and SVM are two commonly algorithms used as surrogate models. However, the Sk-learn implementation of the Support Vector Regression model was excluded for not being suitable for our dataset. The fit time complexity of the model is more than quadratic with the number of samples, making it unfeasible with our 100000 samples per device. The Sk-learn implementation of gaussian process is excluded as well because it builds a $n \times n$ matrix not fitting the memory of our hardware.

As previous research has stated, the accuracy of a surrogate model is the ”key to success”. With a surrogate model perfectly approximating the black-box function, in principle, it is not so difficult to find the optimal solution. To measure the accuracy of our surrogate models we used the MSE and the Coefficient of determination (R^2-score), which are explained in the following.

4.3.1 MSE

MSE is a quantitative measure of the quality of an estimator and is commonly used for regression problems as a loss function. It measures the average squared distance between the predicted values and the actual values. Therefore, it is strictly positive and a lower MSE indicates a more accurate model with a better fit. Since it is distance-based, it depends on the value ranges of the predictions or ground truth data. Because of this relationship, it is often not as intuitive to interpret as other metrics such as the R^2-score. The MSE is a useful metric as it gives greater weight to larger errors, making it a more sensitive measure than other metrics such as the mean absolute error (MAE).
It is defined as in Equation (4.1) with n being the total number observations, y being the vector of true or target values and $\hat{y}$ being the vector of the predicted values by the model.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.1}$$

Name of parameter	type	range of values	input or output
device	integer	$\{0, 1, ..., 8\}$	input
c1, ..., c4	float	$\{-1.0, ..., 1.0\}$	input
t1	integer	$\{2, 3, ..., 13\}$	input
t2, t3	integer	$\{0, 1, ..., 15\}$	input
t4, t5	integer	$\{0, 1, 2, 3\}$	input
t6, t7	integer	$\{1, 2\}$	input
y	float	$\mathbb{R}$	output

Table 4.2: All parameters included in our dataset with a corresponding range of values

4.3.2 R^2-score

The coefficient of determination (also known as R-squared or R^2-score) is a statistical measure, providing a measure of how well a regression model predicts/fits the data. More precisely, it measures the proportion of the variation in the dependent variable that can be predicted from the independent variable(s). The R^2-score normally ranges between between 0 and 1 but there are cases where it can yield negative values. The higher the R^2-score, the more accurate the estimator's predictions are considered, with 1 indicating that the model predictions perfectly fit the data. The R^2-score is often a more intuitively information than the MSE in regression analysis evaluation because it can be expressed as a percentage and doesn't have arbitrary ranges. Therefore, no knowledge of the value ranges of the predictions and ground truth data is needed to evaluate the R^2-score of a single model. It is defined as in Equation (4.2) with n being the total number observations, y being the vector of true values, $\hat{y}$ being the vector of the predicted values by the model and $\bar{y}$ is the mean of the true values.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{4.2}$$

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y \tag{4.3}$$

4.3.3 Default evaluation

In this section, we will describe our default process to evaluate the models and explain our evaluation metrics in detail. The dataset consists of eleven input values and one output value as listed in Table 4.2. We have 900000 circuit measurements equally divided among 9 different semiconductor circuits. These circuits have different performances due to variations in the manufacturing process. The c parameters are conditions while the t parameters are so-called tuning knobs influencing the performance of the circuit. The y parameter is our score parameter and is our scalar performance metric also called FOM. It is our target quantity and calculated from a set of measurements according to a given specification of the device. Therefore, we will be using regression machine learning models to approximate this score parameter.

The data is first split into one dataset for each device. As the devices differ from each other, we are training the machine learning models each on every device once. Per default, the input features are then scaled using min-max preprocessing.
To validate our model and generate the MSE-, R^2-scores, training and prediction times we are using k-fold cross validation with $k = 5$. The model is fitted to the data using the training set. For TF models 12.5% of the training set is taken as the validation set, resulting in a datasplit of 70% training, 20% testing and 10% validation data. Furthermore, during cross validation our TF models are trained for a maximum of 500 epochs/iterations with early stopping. Each iteration the model is fit once on the shuffled training set and evaluated on the validation set by calculating the validation loss. If the validation loss is the lowest out of all epochs of the model, a model checkpoint is created by saving the weights of the model. After 500 epochs or if the validation loss didn't improve for 50 consecutive epochs, the training stops. The weights of the last model checkpoint and therefore the best version of the model on the validation set is then loaded as the result of the training. The trained model then is evaluated on the training and the test set by calculating the MSE and R^2-score-score between the prediction and target values. The MSE and R^2-scores on the test and training set as well as the elapsed time required for the model to train and to generate a prediction for the test set is saved for later evaluation. This process is repeated 4 times like explained in Section 3.1.2 on cross validation.
After cross validating the model, the dataset is again split into a train, test and validation set. 70% of the data ends up in the training set, 20% ends up in the test set and the last 10% in the validation set. A new model is then fit on the training set. For the TF models the early stopping is disabled during this fit and a train history with all epochs and their train and validation losses is saved. Additionally, the model makes a prediction on the test set which is saved as well.

Lastly, the model's robustness is analysed. Our assumption is, that small changes on the input should only lead to small changes on the output value. A robust model therefore shows only minor changes in output when there are small changes in input. To evaluate the robustness of a model we will apply different noise levels on the continuous features in our dataset. We choose not to apply noise to the integer input features because it is not possible to make small changes to them, making the results difficult to evaluate. Regarding our dataset from Table 4.2, noise is only applied to the features c1, c2, c3, and c4.

In the following our noise evaluation is explained. First, 5000 samples are randomly chosen out of the test set and predicted by the model. Then multiple noise tests with different noise strengths are run. The strengths differ between 0.001 and 0.2 and are uniformly spaced resulting in 200 different noise strengths. The standard derivation of the feature data is multiplied with the noise strength and used as the standard derivation of a normal distribution. The noise is then generated by drawing random samples from this normal distribution and applied on the data.
The model predicts multiple scenarios based on the sample data. One scenario includes all features with noise applied and the other scenarios each include only a single feature with noise applied, while the other features remain unchanged. To evaluate the model's robustness the mean squared error is calculated between each prediction and the prediction on the dataset without noise.

4.3.4 Evaluation Metrics

After collecting data the following paragraph explains how, and according to which metrics, the surrogate models will be compared. The main properties that will be interesting for our further experiments are the quality of an estimator, performance and robustness. As the surrogates are trained on each individual device, all metrics are averaged over all device results.

Quality Of A Model

The quality of a model can be evaluated by different metrics. The MSE and R^2-score, on the test set, give a first impression of the models prediction quality. A lower MSE and a higher R^2-score determine a model whose predictions tend to be closer to the true values on average. The R^2-score is easier to interpret and if close to one indicates a overall good model. The MSE is better for a comparison between the different models as it tends to have a higher relative difference. Additionally, to ensure the model is able to generalize the data well, the difference between the MSE on the training set and MSE on the test set needs to be examined. This difference is used to examine if the model is overfitted on the data. Optimally the model does not overfit the data but has a good score on the testset. For further insights, the test set predictions can be compared between each model, showing the distribution and outliers.

The quality of an estimator can have different impacts on the later process of RL. First, an accurate estimator tends to better approximate the function and therefore might better approximate the optima of the function. This could result in a higher and most importantly accurate and more reliable maximum. Furthermore, a less overfit estimator generally indicates a smoother function which would help the RL at finding the maximum. Generally, a better quality of a model is desirable for our experiments.

Performance

To evaluate the performance of a model we have measured the training time, prediction time for the entire testset and, for TF models, the training epochs. In our use-case, the most important performance metric is the prediction time since the surrogate models are used to evaluate the quality of the RL machine learning model by calculating the reward. This reward is calculated very often during the training process of the RL model. Therefore, a short prediction time will quickly pay off, while a short training time will not have as significant an effect since models are trained only once. Nevertheless, a fast training time is still desirable. A lower prediction time could have an major impact on the reinforcement learning performance and therefore is an interesting comparison for analysis. To gain insight into the convergence of TF models, the number of training epochs can be compared between different models. However, a small number of epochs does not necessarily indicate a shorter training time, as more complex models may require longer processing time per epoch.

Robustness

The robustness of a model is evaluated using the results of the previous explained noise test. A model is considered to have a high level of robustness if it shows only minor changes in output when there are small changes in input. This generally indicates a smoother approximation of the function, potentially having a positive impact on the convergence of the RL process. A more robust model could therefore easen the learning process of the RL and is desirable.

5 Experiments

In this section, the models we introduced in Section 3.2.2 are evaluated in different experiments.

The following experiments were run on the following hardware:

- **CPU:** Intel Core i7-9700k 3.60Hz
- **GPU:** NVIDIA GeForce RTX 2070
- **RAM:** 32 GB

5.1 Comparison of models without tuning

To get a first overview over the different models all parameters of the models were kept at their default values The results of the evaluation are shown in Figure 5.1. In terms of estimator quality, as evaluated by the MSE and R^2-score, NNs seem to perform best, even with a standard configuration that is not specifically adapted to our problem. The PSV DNN model and the standard DNN model show similar accuracy/quality, although their architecture and hyperparameters are different. In addition, the difference between the training and test MSE indicates almost no overfitting, especially in comparison to the other models. Thus, NNs seem to be best at understanding the pattern in our data and approximating the black box function. But also the simpler models, such as RF or ET, achieve almost as good MSE and R^2-score performance as the NNs, with the advantage of being faster at training and predicting the data. This will be an interesting trade-off between quality and performance for later use in the learn-to-optimize approach. As expected, the simple estimators like KNN and DT tend to overfit much more, since they are very good at remembering the training samples in their default configuration. For example, KNN remembers the training samples, and with a default of $k = 5$, only 4 other neighbors are taken into account for averaging. With our training set size of 70000, this usually results in a very good prediction on the training points. The KNN and SGD models both perform relatively worse than the other models in their MSE and R^2-scores tests. To further illustrate the extent of this difference, Figure 5.2 shows the prediction graphs of the SGD and the standard DNN model. The first bisector is marked in red and represents the prediction of a perfect estimator, since the prediction would always be equal to the ground truth. All 20000 sample predictions of the test set are plotted in both plots. The difference between the estimators is clearly visible, as the predictions of the SGD model are widely scattered and not even aligned as a line, while the standard DNN clearly shows the tendency to align along the first bisector.

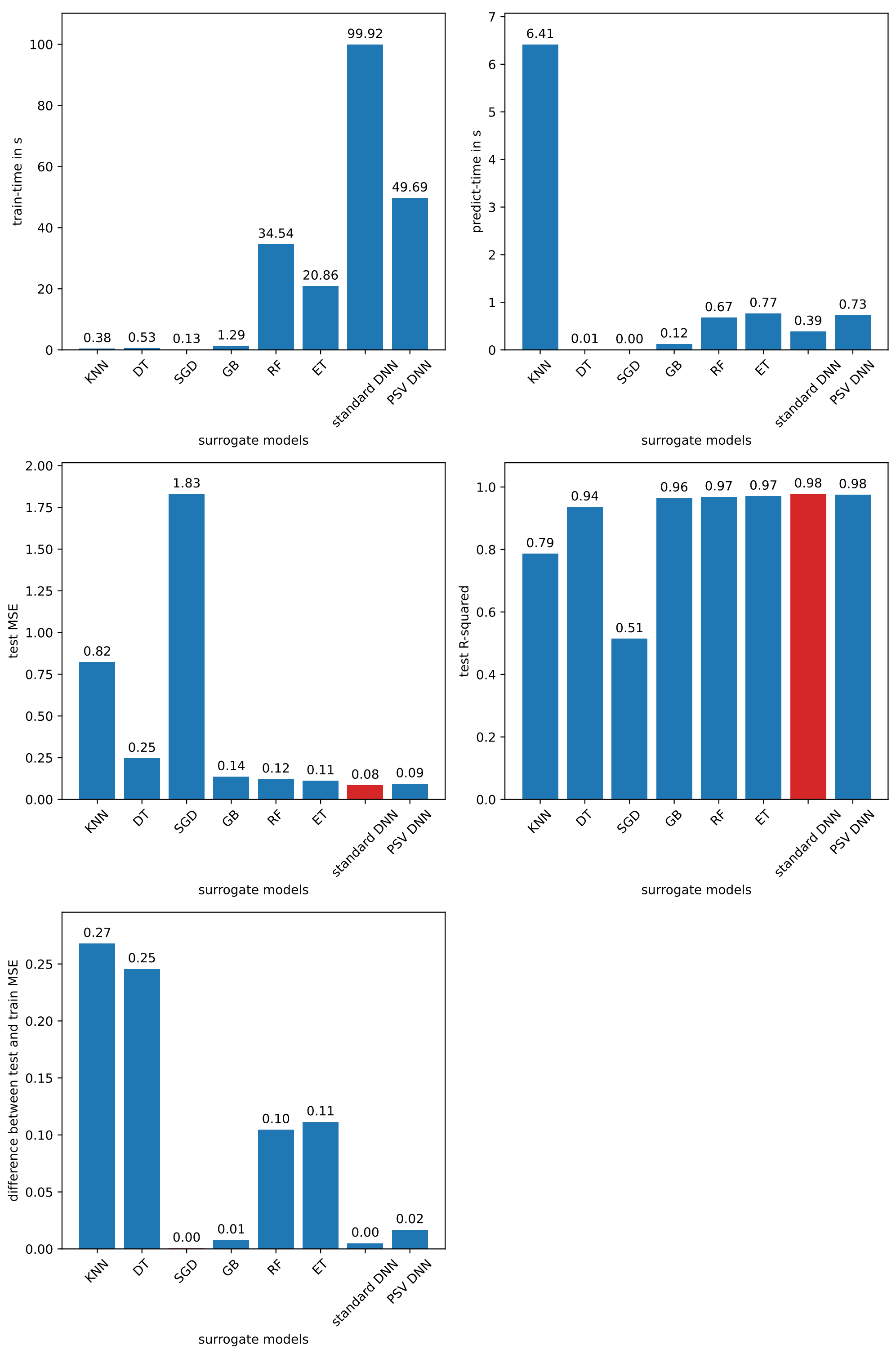


Figure 5.1: The results of the analysis of the models with their default parameters trained as explained in Section 4.3.3. All 5 metrics are shown in bar plots with the different surrogate models as x-axis and the metrics as y-axis. The best surrogate model in the corresponding metric is marked in red. The different bars are annotated with their rounded values.

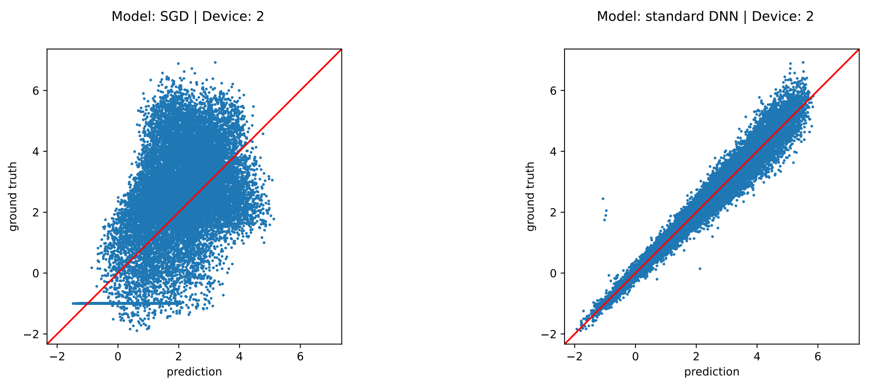


Figure 5.2: The testset prediction of the SGD and standard DNN model on device 2 plotted against the ground truth values. Each point represents one of the 20000 sample predictions. The first bisector is the perfect prediction and marked in red.

Parameter	Range of values
k (number of neighbors)	{1, 2, ..., 150}
leaf size	{1, 2, ..., 100}
p	{1, 2}

Table 5.1: The parameters and their corresponding spectrum for the KNN hyperparameter optimization experiment

5.2 Hyperparameter Optimization

For the parameter space of the hyperparameter optimizations, we always take meaningful large value ranges. These parameter spaces are densely sampled to avoid the grid search failure mentioned by [BB12]. This results in a larger number of possible hyperparameter combinations, making the search for the best hyperparameters less feasible, but allowing a more accurate representation of the space. Therefore, we prefer random search, Bayesian optimization, and Hyperband to exhaustive grid search, which would be infeasible.
To hyperparameter optimize the models, the data is prepared as described in the methodology section. Then for Sk-learn models the `RandomSearchCV` is used, while for TF models we use the KerasTuner module and its implemented algorithms. This hyperparameter optimization is done for each model on each device data. The resulting hyperparameter configurations are then analysed using the default evaluation process explained in Section 4.3.3.

5.2.1 KNN:

As KNN heavily changes it's prediction behaviour dependent on the k parameter it is intuitive to run a hyperparameter optimization on the model. The default $k = 5$ might not be suited for the complexity of this problem or the number of samples in our training set. The hyperparameter optimization was run with the parameter spectrum in Table 5.1. The parameter k (number of neighbors) defines

device	p	k	leaf size	model name
0	2	14	91	KNN 0
1	1	16	47	KNN 1
2	1	12	48	KNN 2
3	1	14	18	KNN 3
4	1	11	65	KNN 4
5	1	12	28	KNN 5
6	1	13	74	KNN 6
7	1	13	2	KNN 7
8	1	14	98	KNN 8

Table 5.2: The results of the hyperparameter optimization for the KNN model. For each device the optimal value of p, k and leaf size is shown.

number of neighbors used for the algorithm. The leaf size parameter is used for the ball tree oder kd tree and can affect the speed and memory consumption of the tree. As stated by Sk-learn the optimal value depends on the problem.[1] The p parameter influences the metric used for distance computation. If p is set to one the KNN model uses the Manhattan-Distance for distance calculation and if p is set to two the model uses the Euclidean-Distance.

We used the Random Search algorithm from Sk-learn [2] for this optimization with 90 iterations. Random search is much cheaper than standard grid search, and since we have a space of 30000 values for each device with cross validation, it adds up really fast. However Random Search with 90 iterations should be good enough for our case as we likely won't find the global optimum but with 90 iterations we will find one of the top 5% of hyperparameter combinations with a confidence of 99% [BB12].

Results: The Random Search results are listed in Table 5.2. All of the listed models are then run through the analysis and evaluation process explained in Chapter 4. The results are shown in Figure 5.3. At first glance, the KNN 2 model performs best on the problem according to MSE and R^2-score on the test set. But since all models except KNN 0 perform quite similarly in terms of MSE and R^2-score, it is good practice and our convention to take the least overfitted model. In this comparison, the training time of the models is not as important because the models don't differ that much in training time due to the nature of the KNN model. Another interesting observation is the difference in prediction time. It looks like the power parameter p has a significant influence on the time the model takes to score the test set, which makes sense since the parameter influences the distance calculation of the model. We will explore this influence of the p parameter later. In conclusion, the KNN 1 model seems to be the best of the 9 models because it overfits the least but still has a similar R^2-score as the best model. The model KNN 1 has a R^2-score of ≈ 0.813 while the best model KNN 2 has ≈ 0.815 which is an insignificant difference.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

[2] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

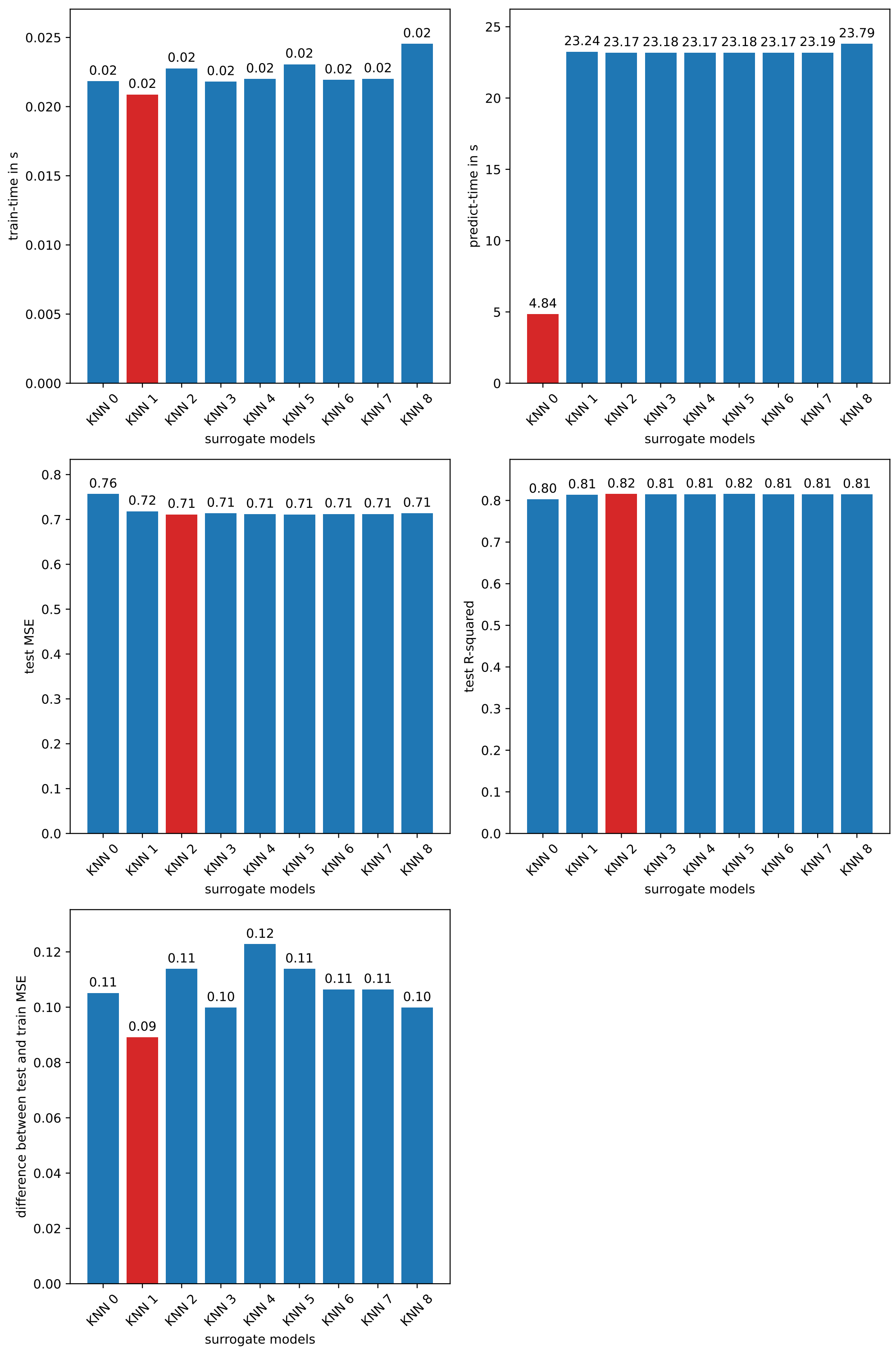


Figure 5.3: The results of the analysis of every optimal KNN model found by the hyperparameter optimization on each device. The models were trained as explained in Section 4.3.3. All 5 metrics are shown in bar plots with the different surrogate models as x-axis and the metrics as y-axis. The best surrogate model in the corresponding metric is marked in red. The different bars are annotated with their rounded values.

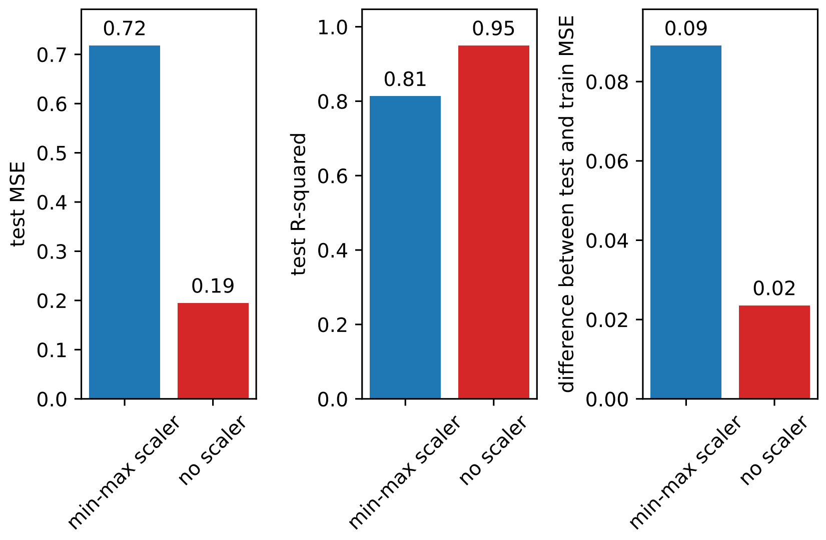


Figure 5.4: Comparison of the KNN 1 model with a min-max scaler and without a scaler according to 3 different metrics

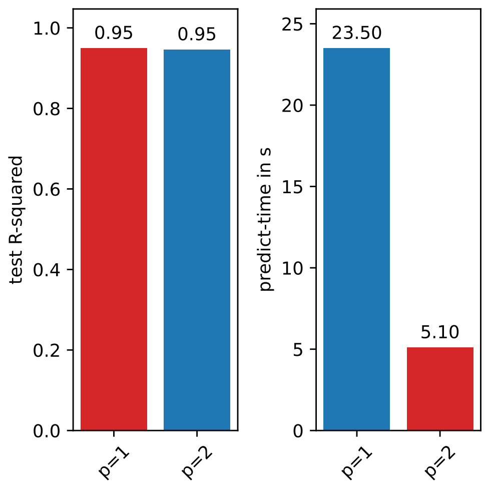


Figure 5.5: Comparison of the KNN 1 model with $p = 1$ and $p = 2$ according to 3 different metrics

Another parameter that can have a huge impact on the model performance is the scaling of the dataset. By default, a min-max scaler is used to scale the input values of the data set, but with a KNN model, this scaling can have a big impact on the distance calculation. As seen in Figure 5.4, the accuracy of the model increases significantly without a data scaler. Both R^2-score and MSE improve by significant amounts.

The chosen KNN 1 model without a scaler is then further examined for the impact of the hyper-parameter p. From our previous comparisons, we expect the model to lose a bit in accuracy or R^2-score, but show an increase in time to score the test set. In Figure 5.5, the evaluation meets

Parameter	Range of values
minimum samples per leaf	{1, 2, ..., 1000}
maximum leaf nodes	{None, 2, 3, ..., 100}
maximum depth	{None, 15, 16, ..., 45}

Table 5.3: The parameters and their corresponding spectrum for the DT hyperparameter optimization experiment

our expectations. The model only decreases minimal in R^2-score from ≈ 0.950 to ≈ 0.946 whilst having a massive decrease in score time from $\approx 23.50s$ to $\approx 5.10s$. The decrease in R^2-score is so small that it is insignificant therefore using the Euclidean-Distance seems way more promising.

The resulting optimal KNN model for our problem has the parameters
`p = 2, number of neighbors = 16, leaf size = 47` and uses no preprocessing on the input data.

5.2.2 DT:

The DT model is another simple model that relies on few hyperparameters that change the behaviour of the estimator. The hyperparameter optimization was run with the parameter space in Table 5.3. The maximum depth and maximum leaf nodes parameter are pretty simple and determine the maximum depth of the DT and the maximum leaf nodes the tree can have. The minimum samples per leaf parameter controls the minimum number of samples a leaf node must have. Nodes will only be split if the resulting leaves have at least the minimum training samples per leaf.[3] The maximum depth of the default DT when trained on our dataset varies between 33 and 42, depending on the device on which the estimator is trained. Therefore, a range between 15 and 45 was picked. “None” was included in the spectrum of maximum depth and maximum leaf nodes as it represents the default parameter. As the DT models train really fast and are really fast at predicting we can use Random Search with 460 iterations. This will result in us finding one of the 1% top parameter combinations with a confidence of 99% [BB12].

Results: The found hyperparameter combinations are listed in Table 5.4. Looking at the resulting hyperparameter combinations, it appears that the value spectrum of minimum samples per leaf is well chosen, as the values don’t cluster around the upper bound. This indicates that a local minima is inside our our value space. The number of maximum leaf nodes seems to work best with its default configuration “None” but one could increase the upper boundary of 100 in future research on the optimal configuration.

The results of the analysis can be seen in Figure 5.6. The results of our evaluation indicate that the train-time and score-time have no significance as they fall within a narrow range of values. In terms of MSE and R^2-score model 2 and 4 seem to be the best performing estimators on the dataset. Again we are choosing 2 as the better model as it has an almost similar MSE and R^2-score but overfits less. To set this improvement into perspective, in Figure 5.7 the comparison between the default model and the tuned model is shown in terms of the quality of the estimator, the MSE-score, and

[3]https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

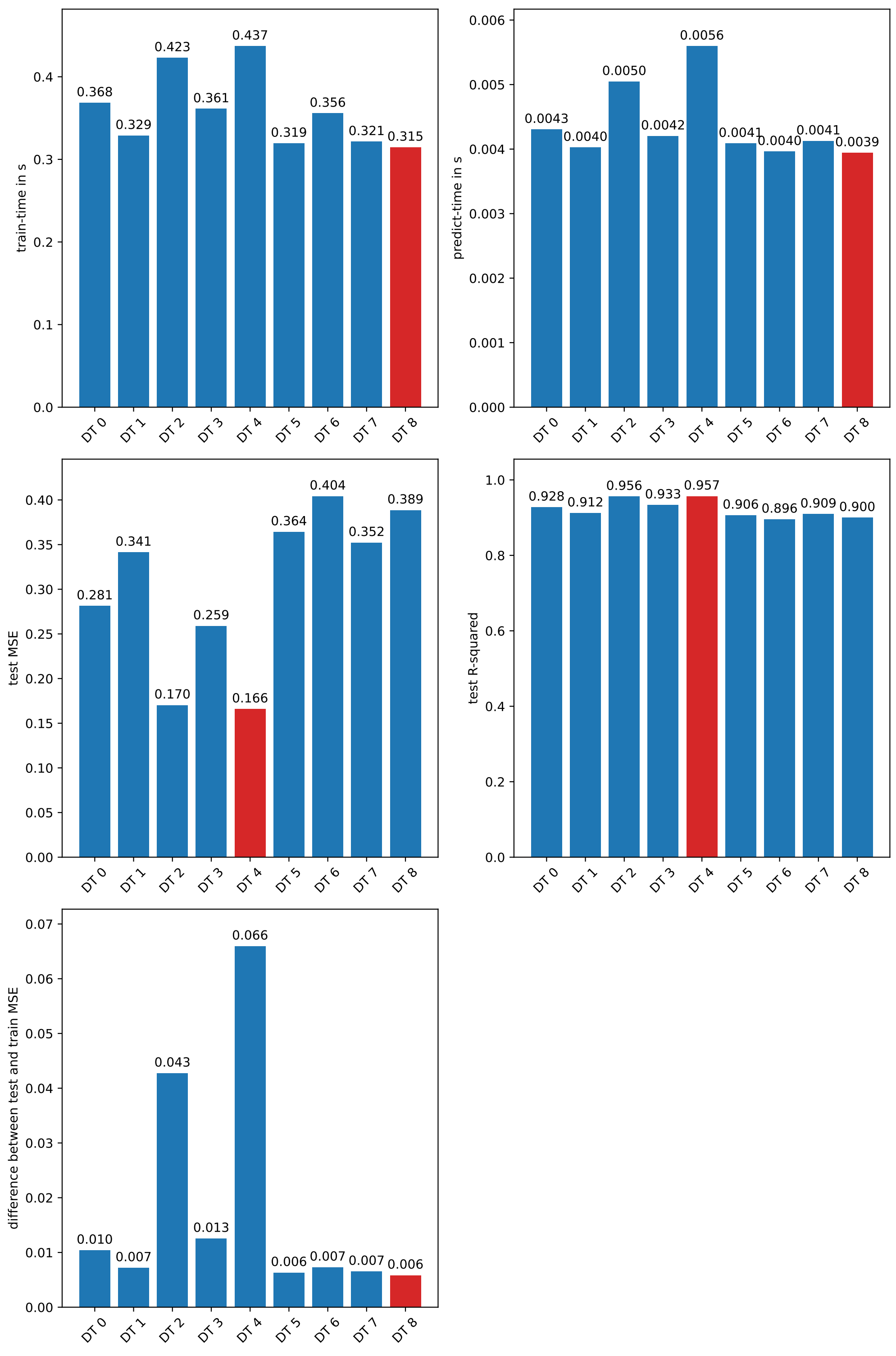


Figure 5.6: The results of the analysis of every optimal DT model found by the hyperparameter optimization on each device. The models were trained as explained in Section 4.3.3. All 5 metrics are shown in bar plots with the different surrogate models as x-axis and the metrics as y-axis. The best surrogate model in the corresponding metric is marked in red. The different bars are annotated with their rounded values.

device	min samples per leaf	max leaf nodes	max depth	model name
0	122	None	15	DT 0
1	230	None	15	DT 1
2	17	None	35	DT 2
3	96	None	29	DT 3
4	10	None	24	DT 4
5	285	None	22	DT 5
6	22	98	39	DT 6
7	254	None	35	DT 7
8	335	None	17	DT 8

Table 5.4: The results of the hyperparameter optimization for the KNN model. For each device the optimal value of minimum samples per leaf, maximum leaf nodes and maximum depth is shown.

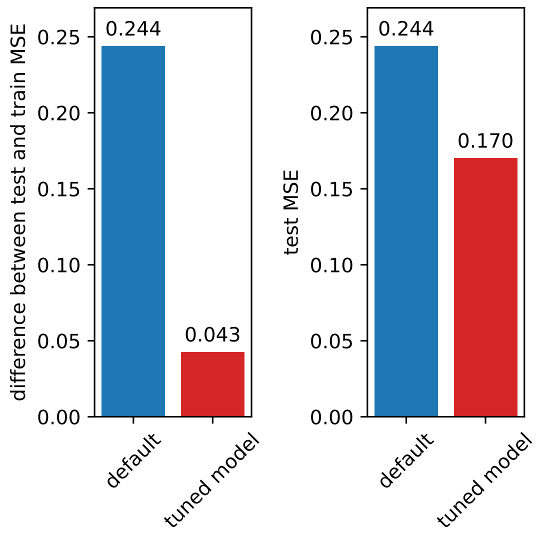


Figure 5.7: Comparison of the default and tuned DT model according to 2 different metrics

the overfitting of the estimator. The models difference between the training and test MSE improved drastically as well as the MSE on the test set prediction. Therefore, the tuned model is much better at generalizing the data and fits our problem better. We also conduct experiments if the Friedman MSE as criterion would make a difference opposed to the default MSE criterion but no significant difference was found. Both criterion's perform very similar in all metrics. Again, we tried the DT model without any preprocessing of the input data but the models quality and speed didn't improve any further. Therefore, the DT model is used with the default min-max scaling of the input data.

The resulting optimal DT model for our problem has the parameters `minimum samples per leaf = 17, maximum leaf nodes = None, maximum depth = 35` and uses min-max scaling on the input data.

Parameter	Range of values
maximum depth	{None, 10, 11, ..., 45}
minimum samples per leaf	{1, 2, ..., 1000}
number of estimators	{50, 60, 70, ..., 500}
maximum samples	{None, 0.01, 0.02, ..., 1.0}

Table 5.5: The parameters and their corresponding spectrum for the RF and ET hyperparameter optimization experiment

5.2.3 RF, ET and GBT:

RF and ET are both based on the DT model. Therefore, we can take a part of our previous experiments as foundation for the hyperparameter optimization of these models.

The selected hyperparameter spectrum is shown in Table 5.5. The parameters maximum depth and minimum samples per leaf are already explained in the previous Section 5.2.2. The maximum leaf nodes will not be a part of the hyperparameter spectrum as the DT tuning showed that it is best on its default configuration. The number of estimators parameter determines the number of trees the model uses and therefore is a parameter with a more coarse resolution as a small change will have an insignificant impact on the quality of the model. The parameter maximum samples controls the number of samples the estimator uses to train a single DT. In our case the float values represent percentages of the training data. Additional to the parameters above the bootstrap parameter of the ET model is set to `True` as it normally defaults to `False`. If bootstrap is set to `False` the model uses the whole dataset to build each tree while we want to use the maximum samples parameter to control the number of samples used.

Results: The found configurations for the hyperparameter optimization of the three models are shown in Table 5.6, 5.8 and 5.7. We used the same Random Search as in the KNN hyperparameter optimization with 90 iterations. The comparison of the tuned models is shown in Figure 5.8 and 5.9. As shown before in the DT tuning the best performing models RF 4, RF 5 and RF 6 tend to overfit the most. In the last comparisons we chose the least overfitted out of the best performing models but in this case training and scoring time have a noticeable difference. Therefore, even though it overfits the most, the parameters of RF 6 are chosen as best due to the quality and speed of the estimator. In our previous research we already discovered that using no preprocessing isn't beneficial for the DT model and therefore we will keep the default min-max scaler, as the RF and ET model are based on DTs.

The ET results are pretty similar to the RF results regarding the correlation between overfitting and quality of the estimator. We will again chose the parameters of ET 3 as it has the best quality predictions and is faster than other estimators with a similar performance.

Because the bootstrap parameter of the ET model is set to `False` per default it will be interesting to see if setting the parameter to `True` and using the number of maximum samples had a positive influence on the models performance. In Figure 5.10 one can see that setting bootstrap to `True` and therefore not using the whole training set for each DT has a positive influence on the model. The quality of the estimators is very similar as the MSE on the test set shows. The improvement on the

device	num of estimators	min samples per leaf	max samples	max depth	model name
0	80	28	0.76	17	RF 0
1	410	14	0.56	44	RF 1
2	110	16	0.51	23	RF 2
3	160	1	0.36	13	RF 3
4	420	10	0.44	28	RF 4
5	430	5	0.55	37	RF 5
6	190	3	0.5	40	RF 6
7	300	40	0.58	None	RF 7
8	150	24	0.36	28	RF 8

Table 5.6: The results of the hyperparameter optimization for the RF model. For each device the optimal value of number of estimators, minimum samples per leaf, maximum samples and maximum depth is shown.

device	num of estimators	min samples per leaf	max samples	max depth	model name
0	60	14	0.64	44	ET 0
1	240	1	0.45	30	ET 1
2	130	6	0.95	40	ET 2
3	110	1	0.78	30	ET 3
4	490	9	0.57	35	ET 4
5	240	2	0.74	16	ET 5
6	120	1	0.48	34	ET 6
7	440	39	0.45	19	ET 7
8	140	33	0.72	39	ET 8

Table 5.7: The results of the hyperparameter optimization for the ET model. For each device the optimal value of number of estimators, minimum samples per leaf, maximum samples and maximum depth is shown.

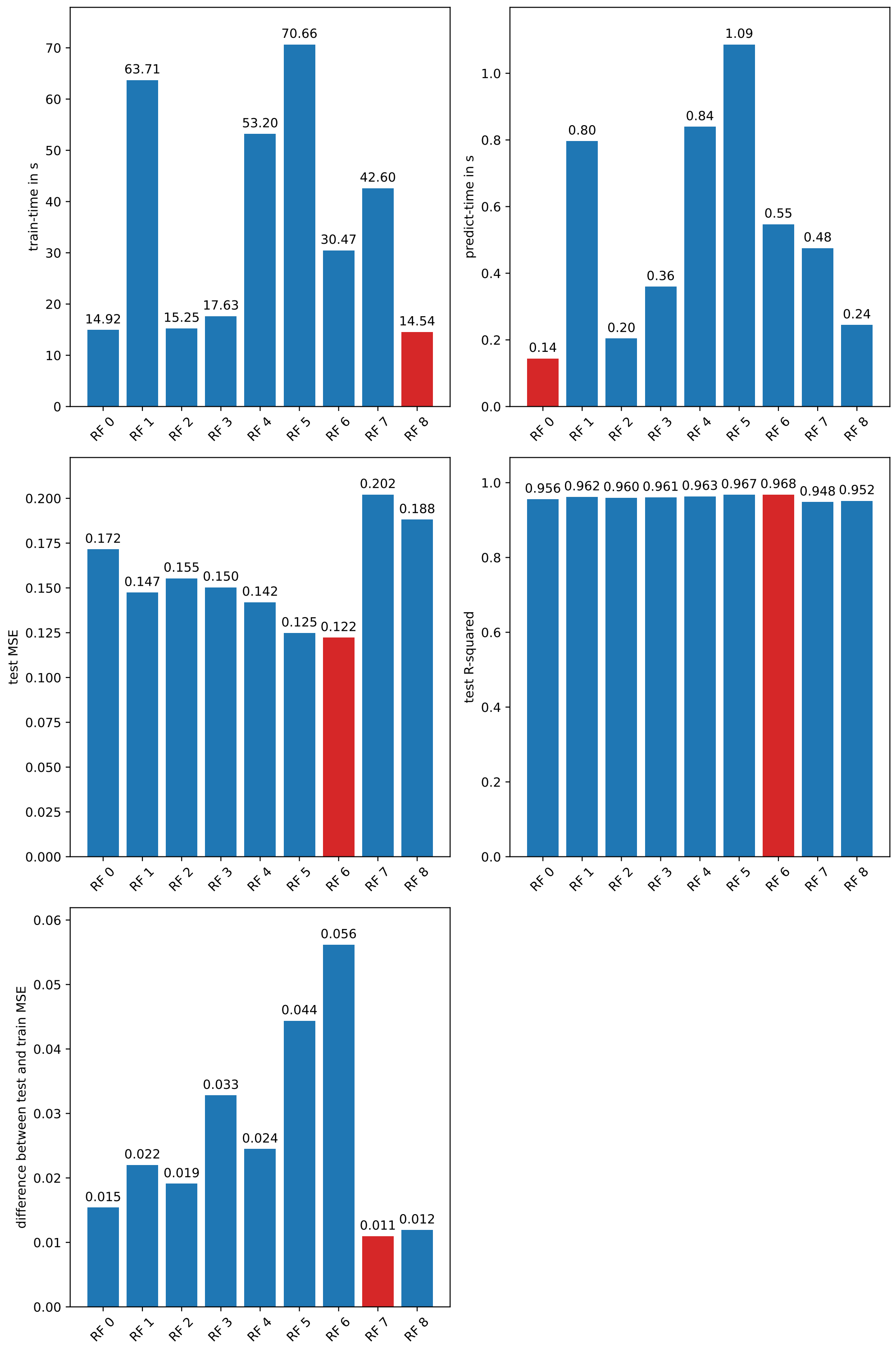


Figure 5.8: The results of the analysis of every optimal RF model found by the hyperparameter optimization on each device. The models were trained as explained in Section 4.3.3. All 5 metrics are shown in bar plots with the different surrogate models as x-axis and the metrics as y-axis. The best surrogate model in the corresponding metric is marked in red. The different bars are annotated with their rounded values.

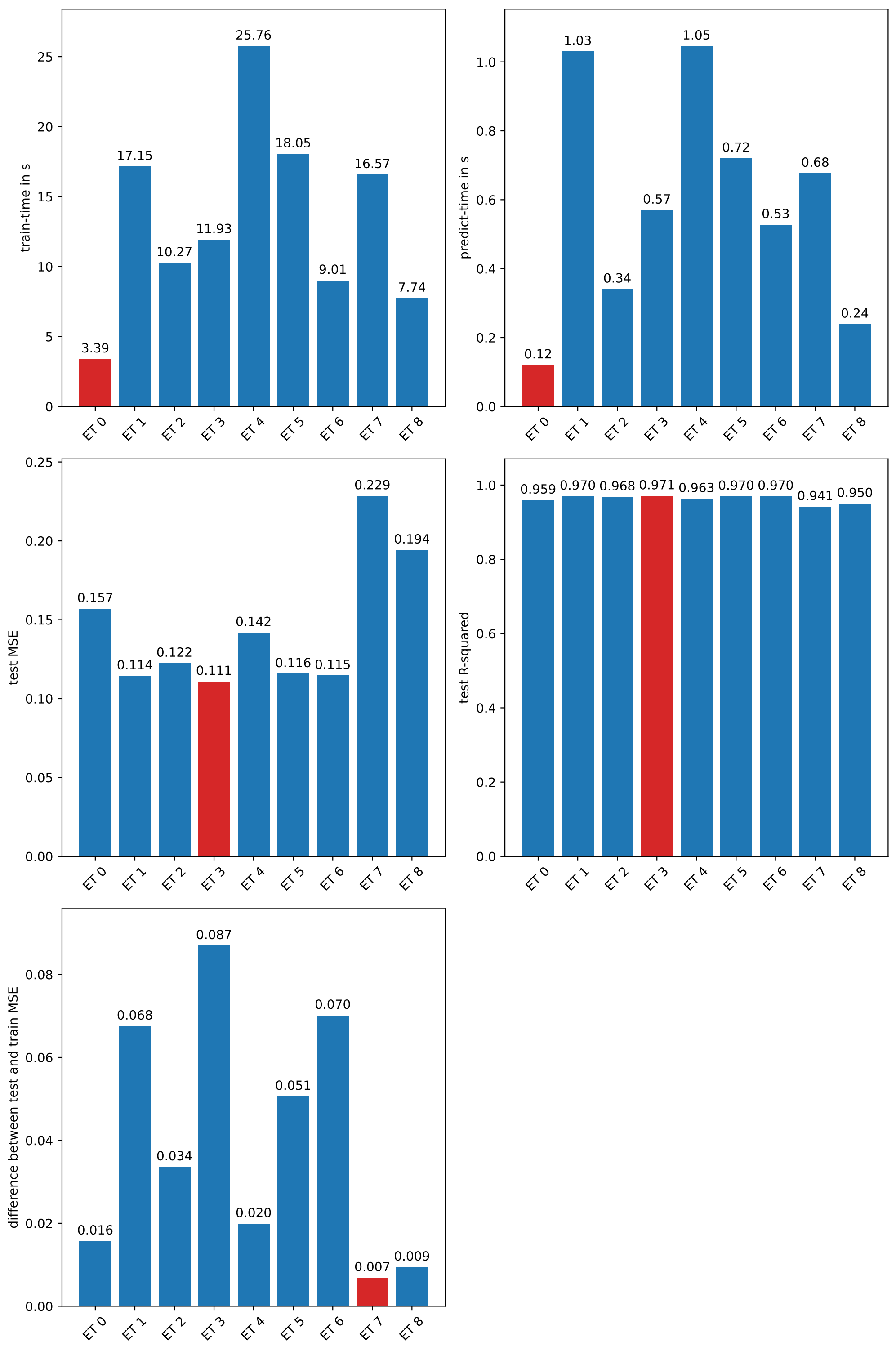


Figure 5.9: The results of the analysis of every optimal ET model found by the hyperparameter optimization on each device. The models were trained as explained in Section 4.3.3. All 5 metrics are shown in bar plots with the different surrogate models as x-axis and the metrics as y-axis. The best surrogate model in the corresponding metric is marked in red. The different bars are annotated with their rounded values.

device	min samples per leaf	maximum iterations	max depth	learning rate	model name
0	59	350	None	1.629	GBT 0
1	623	410	44	1.629	GBT 1
2	94	470	23	1.629	GBT 2
3	152	110	41	1.023	GBT 3
4	689	490	16	1.023	GBT 4
5	717	230	15	1.023	GBT 5
6	808	360	11	1.023	GBT 6
7	659	180	43	1.023	GBT 7
8	914	50	28	1.629	GBT 8

Table 5.8: The results of the hyperparameter optimization for the GBT model. For each device the optimal value of minimum samples per leaf, maximum iterations, maximum depth and learning rate is shown.

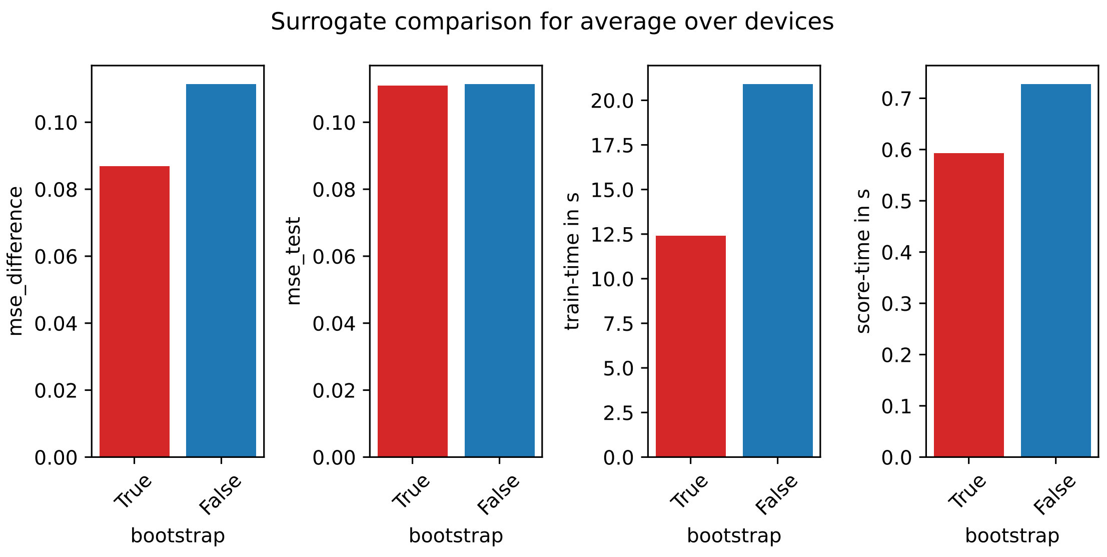


Figure 5.10: Comparison of the ET 3 model with bootstrap enabled and with bootstrap disabled according to 4 different metrics

Parameter	Range of values
number of layers	{1, 2, 3, 4, 5}
units in hidden layer i	{16, 32, 48, ..., 256}
include dropout layer	{False, True}
dropout rate	0.01-0.2 (sampling=”log”)[4]
learning rate	0.0001-0.01 (sampling=”log”)[4]

Table 5.9: The parameters and their corresponding spectrum for the DNN hyperparameter optimization experiment

training speed of the model is in line with our expectations as fitting each tree on only 78% of the dataset is faster than using 100% of the dataset for each tree. This normally results in smaller trees which has a positive impact on the scoring time of the model and on the overfitting of the model as the trees are less complex and better at generalizing the problem. As for GBT the selection is way more simple (see Figure 5.11. The MSE and R^2-score values are the best and very similar for the models 3, 4, 5, 6, 7 but in terms of train and score-time the third model performs the best. The only disadvantage is that the third model overfits the most out of the 5 best estimators but the MSE-difference is in such small ranges that this shouldn’t make a difference. To be precise it overfits 16,5% more than the most accurate model GBT 6 but has a 32,9% faster train time and 47,9% faster score time. Therefore the third models hyperparameter configuration is chosen as the best.
The resulting optimal models have the parameters

```
RF: number of estimators = 190, minimum samples per leaf = 3,
    maximum samples = 0.5, maximum depth = 40, bootstrap = True}
ET: number of estimators = 110, minimum samples per leaf = 1,
    maximum samples = 0.78, maximum depth = 30, bootstrap = True}
GB: minimum samples per leaf = 152, maximum iterations = 110,
    maximum depth 41, learning rate = 1.023}
```

and use min-max scaling on the input data.

5.2.4 Deep Neural Networks:

As a DNN heavily depends on various hyperparameters controlling the networks’s architecture, generalization and optimization and already performed well in our first comparison (see. Section 5.1), it seems promising to run a hyperparameter optimization on these models. The Hyperparameter Optimization was spread across 4 different Optimizations with different algorithms and algorithm parameters. After the 4 Optimizations the best two models of each Optimization are chosen and compared between each other.

The number of layers and units in hidden layer i are pretty intuitive and determine the number of hidden layer and their number of neurons/units. A dropout layer with the dropout rate is added after the first hidden layer if the include dropout layer parameter is set to `True`. The Dropout layer is used

[4]https://keras.io/api/keras_tuner/hyperparameters/#float-method

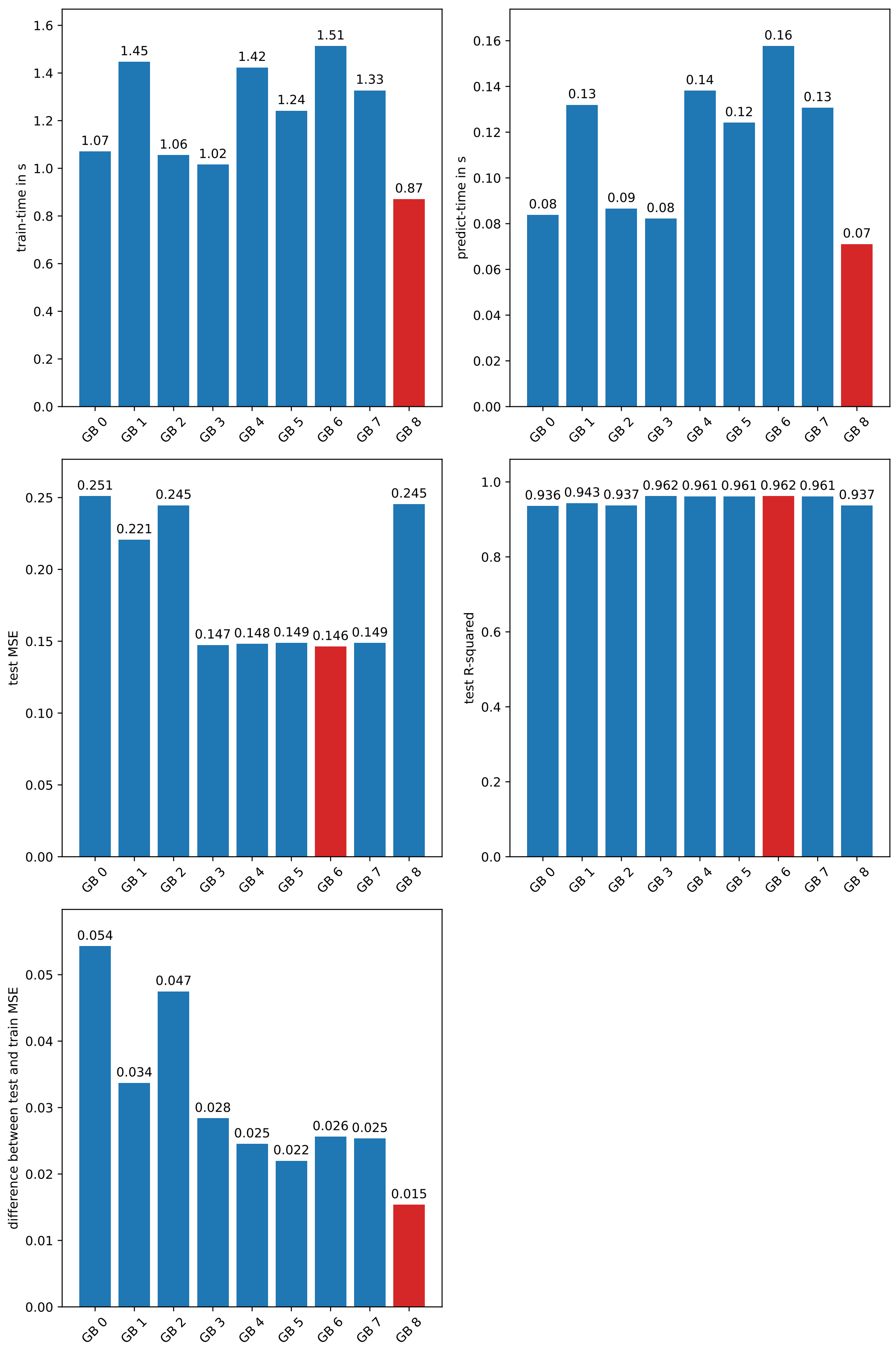


Figure 5.11: The results of the analysis of every optimal GBT model found by the hyperparameter optimization on each device. The models were trained as explained in Section 4.3.3. All 5 metrics are shown in bar plots with the different surrogate models as x-axis and the metrics as y-axis. The best surrogate model in the corresponding metric is marked in red. The different bars are annotated with their rounded values.

to prevent overfitting by randomly setting input units to 0 at the frequency of the dropout rate at each step during training. The remaining input units are then scaled by a factor of $1/(1 - \text{dropout rate})$ to keep the total sum of inputs the same. The learning rate parameter determines the learning rate of the adam optimizer.

For the hyperparameter optimization of the DNNs we used the library KerasTuner and the three different algorithms Random Search, Hyperband and Bayesian optimization. In the first and second optimization we used the Hyperband algorithm with the max epochs of 80 and 250. But throughout the optimizations Hyperband has a problem: As the Hyperband algorithm starts to train a wide variety of parameter combinations on a small number of training epochs, models that perform worse on a small number of epochs and converge slower are eliminated. This is problematic because more complex and larger models, in terms of neurons, converge faster and more simple models take more training to reach the similar or even better quality of prediction. The range of hyperparameter configurations that lead to simple models therefore is completely excluded but as the standard DNN model with a really simple architecture shows, those models can perform promising as well on our problem and tend to overfit less which is a admirable characteristic.

In the next optimization we therefore used Bayesian optimization with the parameter spectrum in Table 5.10. The Bayesian optimization as well showed some odd behaviour during the tuning. In Table 5.11 the results for the third tuning with Bayesian optimization are shown and it shows that it always seems to converge to maximum or minimum value of the parameter. Those results for optimal configurations deviate significantly from the results of the Hyperband optimization and the standard DNN model again shows a similar performance whilst having a completely different architecture. But an argument can be made that the hyperparameter spectrum we chose is too narrow and the optimum is outside of the maximum values of the hyperparameters. Therefore to further prove the assumption of Bayesian optimization being not suitable for our problem we did another optimization with a wider spectrum for the different parameters as seen in Table 5.12. The upper bound of the ranges are exceptionally high and unpractical for our problem as the models gets way to complex. The optimal hyperparameters found by Bayesian optimization are the following:

```
include dropout layer = True, dropout rate = 0.01, number of layer = 10
units in layer 0 = 1056, units in layer 1 = 16, units in layer 2 = 1056,
units in layer 3 = 16, units in layer 4 = 16, units in layer 5 = 16,
units in layer 6 = 16, units in layer 7 = 1056, units in layer 8 = 1056,
units in layer 9 = 1056
```

Bayesian optimization again chooses the minimum or maximum value of each hyperparameter. This results in DNNs with a really complex and unconventional architecture, which constantly varies between hidden layers with 1056 and 16 neurons. In Table 5.13 the resulting model is compared to our standard DNN defined in the beginning. The results prove our assumption of Bayesian optimization not being able to hyperparameter optimize the DNNs for our problem. The model is way to complex resulting in significant higher training and score time, greater overfitting and a negligible increase in MSE. In further experiments, both the Bayesian optimization for hyperparameter optimization of our DNNs and the over-complex models that are unsuitable for our task were excluded.

After realising that both more complex algorithms seem to show odd behaviour on our problem we used Random Search with 80 trials and 150 epochs per model. From each tuning the best two models were chosen and analysed for a final comparison to determine the best DNN. The hyperparameter configurations of the models can be seen in Table 5.14 To compare the models to each other we

Parameter	Range of values
number of layers	{1, 2, 3, 4}
units in hidden layer i	{16, 32, 48, ..., 256}
include dropout layer	{False, True}
dropout rate	0.01-0.1 (sampling="log")
learning rate	0.0001-0.01 (sampling="log")

Table 5.10: The parameters and their corresponding spectrum for the third DNN hyperparameter optimization using Bayesian optimization

device	number of layers	include dropout	dropout rate	learning rate	units in layer 0	units in layer 1	units in layer 2	units in layer 3
0	4	True	0.01	0.0001	256	256	256	16
1	4	True	0.01	0.0001	256	256	16	256
2	4	True	0.1	0.0001	256	16	256	256
3	4	True	0.01	0.0001	256	16	256	256
4	4	False	-	0.00019	256	256	16	256
5	4	True	0.01	0.0001	256	256	256	160
6	4	True	0.01	0.0001	256	112	256	16
7	4	False	-	0.01	256	16	256	16
8	4	True	0.1	0.0001	256	16	256	256

Table 5.11: The results of the third hyperparameter optimization for the DNN model using Bayesian optimization. For each device the optimal values of the different parameters are shown.

Parameter	Range of values
number of layers	{1, 2, ..., 10}
units in hidden layer i	{16, 32, 48, ..., 1056}
include dropout layer	{False, True}
dropout rate	0.01-0.1
learning rate	0.0001

Table 5.12: The parameters and their corresponding spectrum for the Bayesian optimization experiment. The spectrum are exceptionally wide to test if Bayesian optimization will again converge to the upper bound.

model	MSE difference	test MSE	train-time	score-time
Bayesian tuned DNN	0.020	0.084	839.043	1.385
standard DNN	0.005	0.085	110.042	0.499

Table 5.13: The comparison between the Bayesian tuned and standard DNN model according to 4 different metrics

model name	number of layers	include dropout	dropout rate	learning rate	units in layer 0	units in layer 1	units in layer 2	units in layer 3
2_1	4	True	0.072	0.00049	208	64	112	80
2_7	3	True	0.156	0.00047	80	240	144	-
3_5	4	True	0.017	0.00025	128	32	80	144
3_7	3	True	0.050	0.00064	128	128	48	-
4_2	4	True	0.100	0.00010	256	16	256	256
4_8	4	True	0.100	0.00010	256	16	256	256
5_2	4	True	0.066	0.00012	240	80	48	112
5_7	2	True	0.012	0.00083	112	96	-	-

Table 5.14: The best models of each of the 4 DNN hyperparameter optimization and their corresponding hyperparameter configurations

used cross validation with 5 folds with a maximum of 500 epochs for each fold and early stopping on the models with a patience of 50 epochs. After cross validation a new model is trained without early stopping for 500 epochs and the training history is saved to determine the epochs the model took to reach the lowest validation loss.

It is immediately noticeable that all of the models are tuned and have very similar behaviour. The absolute difference in MSE, R^2-score, MSE-difference between test and train, robustness and prediction time is very small. Taking the MSE as an example the 2_7 model has the highest value with 0,0867 and the 4_2 has the lowest value with 0,0800. Therefore the difference is only 0,0067 which is insignificantly small and shouldn't be noticeable when predicting values with the models. The only noticeable difference is in the training epochs needed for the best performing model but the epochs and the correlating training time is one of the less important metrics. Due to all these mentioned points the selection of a best model is difficult but as our DNNs represent our most accurate models with the highest quality we chose the 4_2 model as the final tuned DNN. Overall it is pretty interesting to see that even though the models differ in their architecture their performance seems to be very similar to each other. Comparing the default with the tuned DNN further proves this assumption. The only noticeable improvement from this optimization is the MSE on the test set. But even this improvement is minor as the MSE only improved by $0,0058$ which is insignificant. This proves that we have reached a local minimum and that the hyperparameters of the DNNs aren't as important as expected. The main advantage is the usage of a DNNs overall.

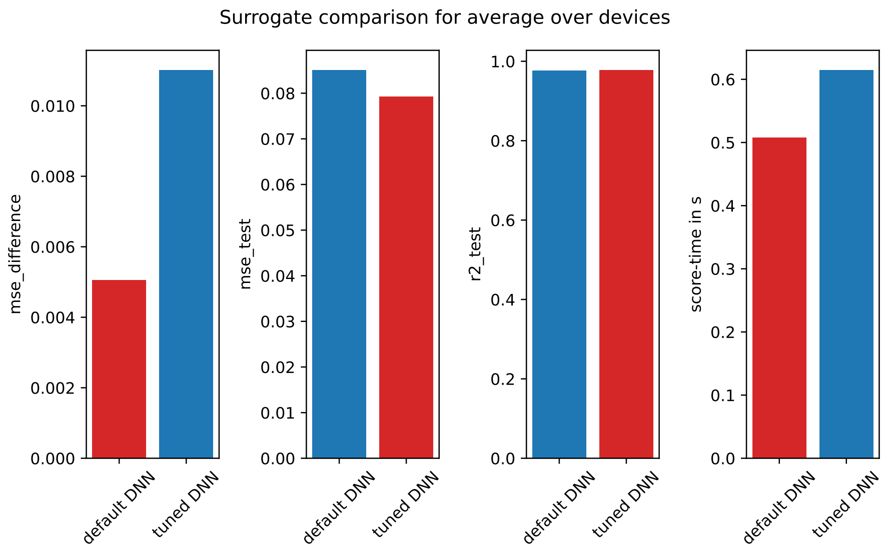


Figure 5.12: Comparison of the default and the tuned DNN according to 4 different metrics

Interesting future research could try other ranges of hyperparameters exceeding our upper limits or even other types of neural networks like recurrent neural networks on the problem as well as analyze why these types or hyperparameters perform especially well on our problem. The only disadvantage of exceeding those limits is the rising complexity of the networks which has a negative impact on the training and prediction speed of the models and therefore on the speed of the later RL. The resulting tuned DNN has the hyperparameters
`number of layers = 4, dropout = True, learning rate = 0.0001, units in layer 0 = 256, units in layer 1 = 16, dropout rate = 0.1, units in layer 2 = 256, units in layer 3 = 256`
and will be referred as “tuned DNN” in the following.

5.3 Comparison of models with tuning

After running hyperparameter optimizations on the models we will make another comparison between the tuned models. The included models in the comparison are the tuned versions of KNN, DT, RF, ET, GBT. Additionally, the PSV model and the tuned DNN model are included. All models were again trained and tested like described in Chapter 4 on page 23 and the results are shown in Figure 5.13 and 5.14. In this section, we present the findings from the analysis and evaluate them by comparing the different models and their behaviour.

The results obtained from the analysis show that the overall performance of all models improved significantly. Regarding the R^2-score, which is a good representation of the overall performance of an estimator, all models seem to be in a similar range of values. This shows that are models are capable of approximating the behaviour of the black-box fairly well. The MSE shows a severe difference between the models but this is due to the relative and not absolute difference in values and the MSE already being rather small. This is further proven by comparing the prediction graphs of the KNN model and the ET model as seen in Figure 5.15. Between both graphs there is a noticeable

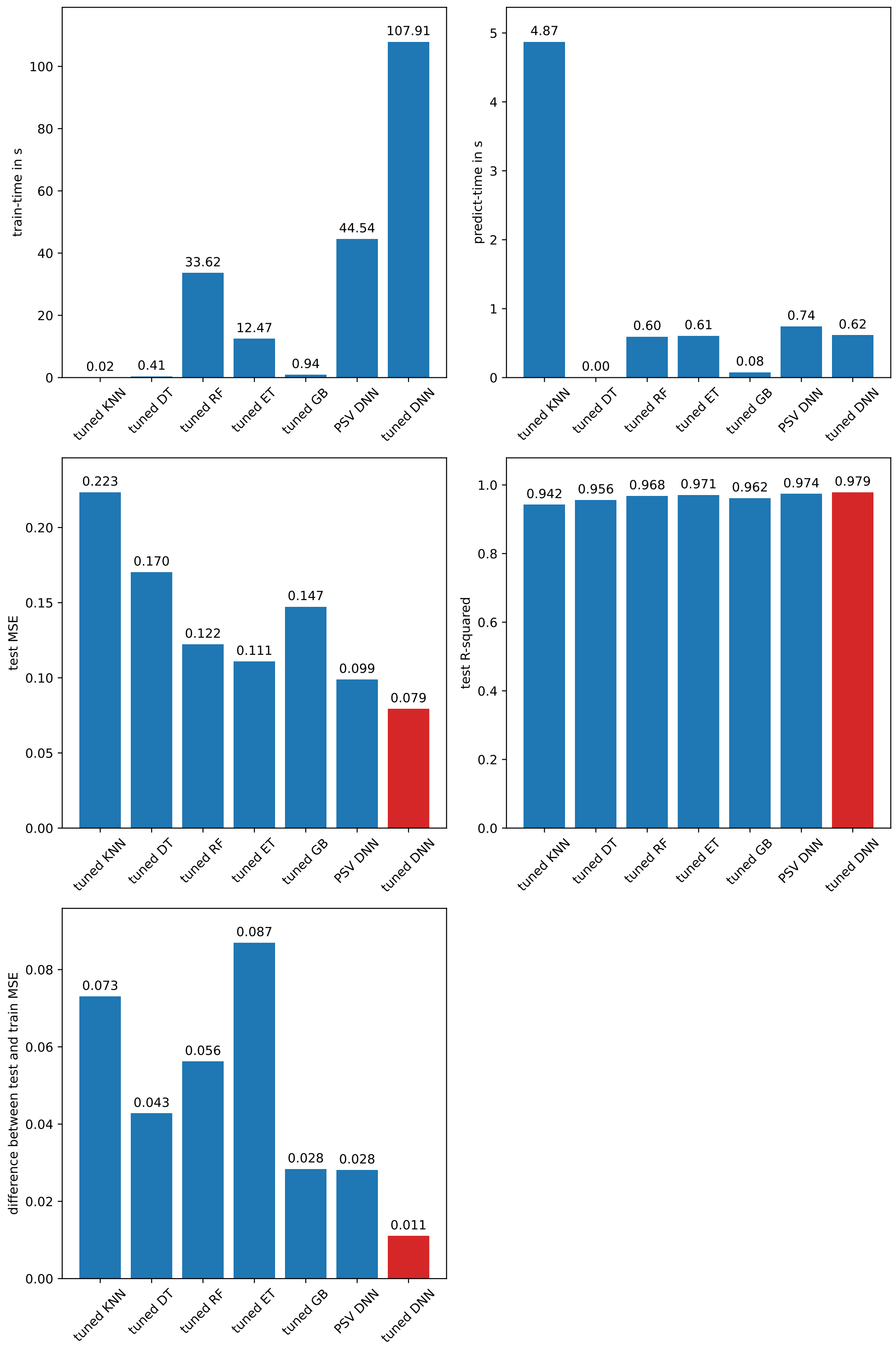


Figure 5.13: The results of the analysis of the tuned models trained as explained in Section 4.3.3. All 5 metrics are shown in bar plots with the different surrogate models as x-axis and the metrics as y-axis. The best surrogate model in the corresponding metric is marked in red. The different bars are annotated with their rounded values.

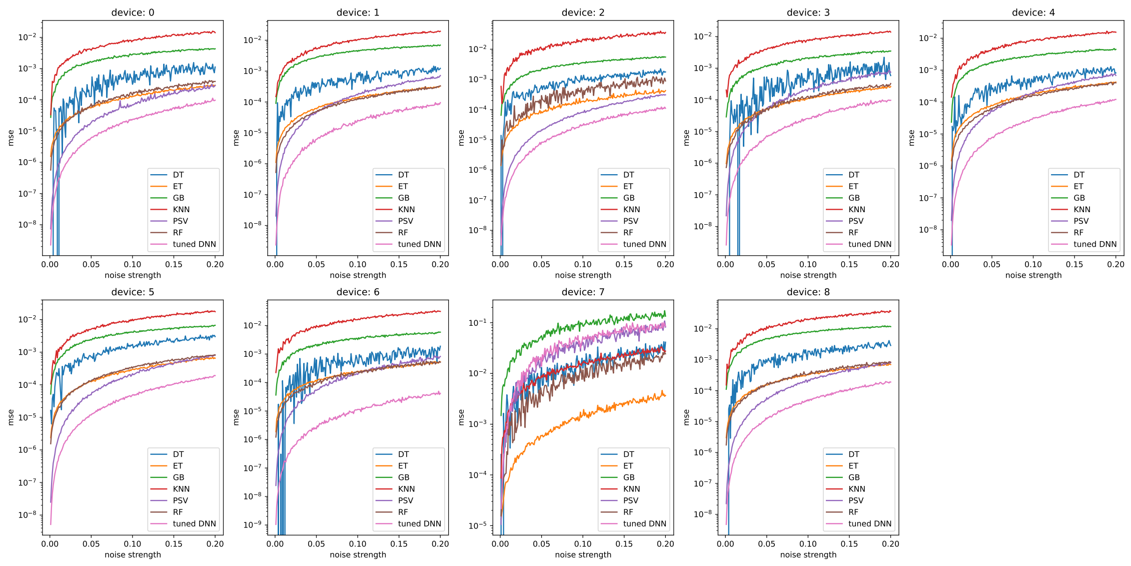


Figure 5.14: Comparison of the robustness of the tuned models for each device. The x-axis shows the noise strength and the y-axis shows the MSE calculated as explained in Section 4.3.3

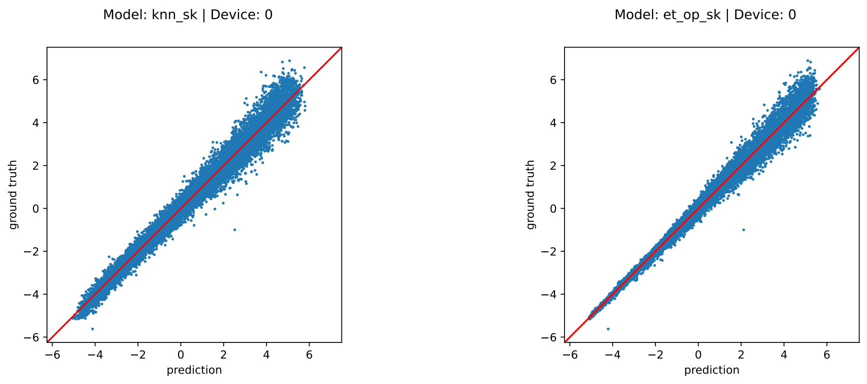


Figure 5.15: The testset prediction of the KNN and ET model on device 0 plotted against the ground truth values. Each point represents one of the 20000 sample predictions. The first bisector is the perfect prediction and marked in red

difference as the prediction points of the ET model are a little bit more narrow and closer to the optimal solution, the first bisector. Particularly the low values show the largest difference as the ET model predictions are really close to the true values while the KNN model deviate more from the ground truth. Still, this difference is not as severe as the MSE values might suggest, since in terms of relative difference the KNN model seems to be only half as good. This further proves the point that all models are capable of predicting the behavior.

Regarding the robustness of the models, the tuned DNN, the KNN and the GBT models stand out. The KNN is the least robust of all the models, while also being the least accurate. Accuracy affects our results from the robustness analysis, so this result was expected. However, the GBT is the second least robust model overall. This is surprising since the model did not show extreme overfitting or particularly poor accuracy. The model seems to be a less robust model overall, regardless of the other metrics. The tuned DNN is again the most robust model. The combination of robustness with accurate predictions and low overfitting generally shows that the model is able to find the pattern in our data. Therefore, the tuned DNN is a very reliable model.

Overall the KNN model seems to be the worst out of the models. It has the lowest MSE and R^2-score and is one of the most overfitted and less robust models. The training time seems to be the only advantage as the model only saves the training values but this is masked by the very bad performance at predicting data queries. In pairwise comparison the GBT model seems to outperform the DT model as it has a better quality of measurement with a better MSE and R^2-score, overfits less and is about as fast at training and scoring. This was expected as the goal of the GBT model is to build upon the DT and further improve its performance. Only in terms of robustness the DT model outperforms the GBT model by being way less sensitive to noise which is really interesting behaviour. As with the previous pairwise comparison the tuned DNN seems to be better than the PSV model. It has a better quality at predicting unknown data, overfits less, is consistently more robust and faster at predicting queries, the only disadvantage being the time to train the model. But this disadvantage is not of great significance as the model only needs to be trained once on the data and a difference of approximately one minute shouldn't be as important.

Generally the tuned DNN performs the best on our problem while having small disadvantages in training and scoring time but the selection depends on the desired attributes. For the most accurate prediction we choose the tuned DNN whilst for a also good and really fast estimator we would chose the GBT or DT model. Another interesting observation is the ET model being very robust towards noise independent of the device it approximates. It is not always the most robust model but stays robust even on broken devices like device 7 while all other models get really susceptible to noise. As ET is also the most overfitted model this might correlate because the model is more fixed on the training values and therefore won't deviate from its prediction due to noise as fast as other models.

5.4 Surrogate-based Monte Carlo Sampling

Before we research on the influence of the different surrogate models on the learn-to-optimize process, we will compare the global maxima of the models. Afterwards, we will choose the models for the later learn-to-optimize experiment.

To determine the global maximum the surrogate models were trained on device 0 using 90% of the data for training and 10% as test data to monitor the quality and overfitting of the model. For the DNN models 12,5% of the training data is used as validation data. We than ran a Monte Carlo sampling with 10000 samples and 1000 trials. The optimal parameters, optimal score and all sample scores of the trial with the optimal score are saved. The results of the Monte Carlo sampling are shown in Table 5.15 and the surrogate metrics are shown in Table 5.16. Additionally the maximum values of the training and test set and the default DT model are included in the table for comparison. On first glance, the models all seem to find a fairly similar maxima. Only the GBT model stands out with a uncommonly high score close to the actual maximum in the training set. All of the maxima are somewhat distant from the actual maximum of the training set, even though the models were all able to achieve a good MSE and R2. This seems like a failure of the models at first, but a closer look at the variation in the training set gives insight into this behavior. In Figure 5.16 the distribution of the training data is shown in the format of a strip plot. Each point in the plot represents one

model name	max score	c1	c2	c3	c4	t1	t2	t3	t4	t5	t6	t7
tuned KNN	5.737	0.737	-0.998	-0.272	-0.492	9	12	14	0	2	2	1
tuned DT	5.694	0.878	0.978	0.642	0.195	9	14	14	3	3	2	2
tuned RF	5.656	-0.964	-0.680	0.525	0.329	11	14	13	3	3	2	1
tuned ET	6.018	-0.083	0.791	-0.781	0.979	13	12	14	1	1	2	2
tuned GBT	6.813	0.244	-0.594	0.173	-0.744	3	11	13	3	2	2	2
tuned PSV	6.036	0.773	-0.989	-0.636	0.871	4	13	13	3	3	2	2
tuned DNN	5.658	0.847	-0.891	0.914	-0.451	4	11	13	3	1	2	2
default DT	6.989	-0.941	0.243	0.792	0.440	3	12	11	1	1	2	1
train max	6.989	-0.714	-0.090	0.579	-0.323	3	12	11	1	3	2	2
test max	6.337	0.406	0.744	0.140	0.471	4	9	13	1	0	2	2

Table 5.15: The input parameters resulting in the best possible score found by Monte Carlo sampling for each model.

model name	MSE train	MSE test	R^2-score train	R^2-score test
tuned KNN	0.089	0.099	0.985	0.983
tuned DT	0.077	0.100	0.987	0.983
tuned RF	0.040	0.076	0.993	0.987
tuned ET	0.015	0.070	0.997	0.988
tuned GBT	0.068	0.084	0.989	0.986
tuned PSV	0.051	0.061	0.991	0.990
tuned DNN	0.050	0.056	0.992	0.991
default DT	0.000	0.153	1.000	0.974

Table 5.16: The performance metrics of each surrogate model in the Monte Carlo experiment

of the 100000 samples in the dataset and the corresponding y / FOM value. Only a few samples are located above 6 and in the range of the maximum. Therefore, most models fail to accurately approximate the space around the maximum because it is sparsely sampled.
Furthermore, the maximum of the tuned KNN, DT, RF, and ET models all use averaging in their algorithm, which further explains the distance between the models' maxima and the training set maximum. ET especially is a little bit higher than the other three models using averaging. This might be due to the fact that it overfits more than the other models and memorized the outliers of the training set better. The model's maximum is less reliable because the overfitting seems quite significant.

Especially the tuned DT model shows that limiting the models complexity and overfitting lead to more averaging and to a cut-off maximum. Comparing the model with the default DT, which clearly overfits, with a perfect training score and a significantly worse test score, a higher maximum does not necessarily mean that the model is better. The default DT is built to perfectly memorize the training values and does not smoothen the function as much as the tuned DT.

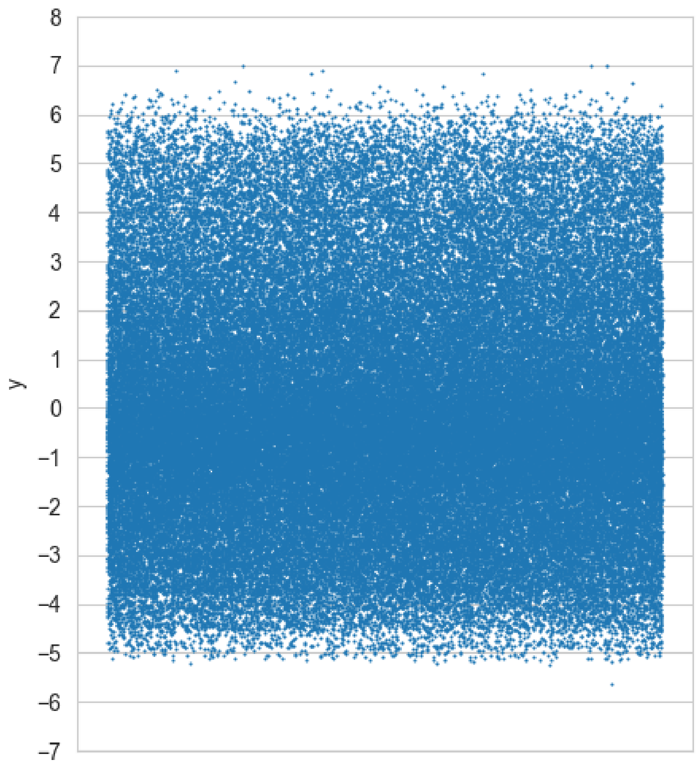
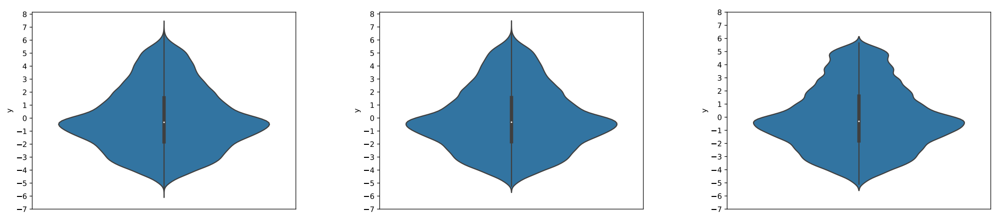


Figure 5.16: A stripplot showing the distribution of all y / ground truth values in our dataset on device 0

To visualize this smoothing, the violin plot of the y / ground truth values of the dataset and of the predictions from the default and tuned DT model of the dataset can be seen in Figure 5.17. Violin plots visualize the distribution of a numeric variable similar to a box plot. However, the violin plot uses density curves, where the width of each curve corresponds to the approximate frequency of the data points in each region. For additional information, a box plot is overlaid on top of the violin plot. Comparing the dataset and the default DT model the graphs are almost identical as the major part of the dataset is used as training data. However, regarding the space around the maximum a significant difference between the default and the tuned DT model is noticeable. The tuned model has a general lower maximum and the values in the upper part gather around different points, creating a wave like shape. It is important to emphasize the fact that the violin plots are made on the training set data and therefore do not represent the overall performance of the model. The MSE and R^2-score exist for this purpose.

Looking only at the maxima of the models, the default DT seems to be the best model. However, to evaluate the model other metrics like overfitting need to be considered. For our task at hand the model needs to be able to predict arbitrary queries as accurate as possible, regardless of whether they are present in the training set. On the unseen test data the model is the worst performing



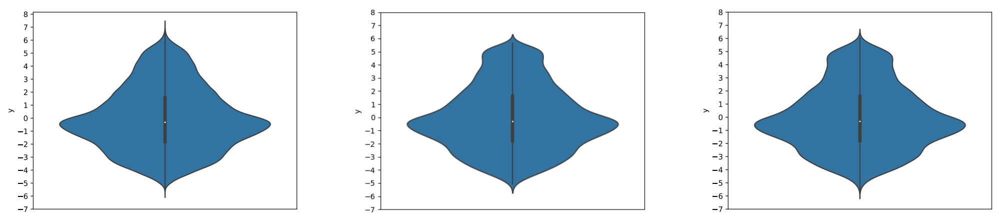
(**a**) dataset (**b**) default DT (**c**) tuned DT

Figure 5.17: Comparison of the violin plots of the ground truth values of the dataset and the prediction of default DT and tuned DT on the dataset

model name	max score	c1	c2	c3	c4	t1	t2	t3	t4	t5	t6	t7
default DT	6.989	-0.941	0.243	0.792	0.440	3	12	11	1	1	2	1
dataset sample 1	4.891	-0.343	-0.483	-0.043	-0.888	3	12	11	1	1	2	1
tuned DNN	5.658	0.847	-0.891	0.914	-0.451	4	11	13	3	1	2	2
dataset sample 2	5.023	-0.374	0.537	0.164	0.026	4	11	13	3	1	2	2
dataset sample 3	5.233	0.410	-0.722	-0.579	-0.436	4	11	13	3	1	2	2

Table 5.17: The best possible input parameters found by the default DT and tuned DNN compared to 3 different samples from the dataset

one. Therefore, we can't solely evaluate the models on the maximum scores. Furthermore, the configuration for the maximum from the decision tree shows this problem as it differs from the configuration of the training set. In our experiment setting we are not able to test the configurations of the models on a semiconductor circuit and our only data are the samples of our dataset. We will compare the found configuration of the default DT model and our tuned DNN, which predicted our function the most accurate according to previous experiments, each with corresponding samples from the dataset. The corresponding data samples match the configuration predicted by the model in tuning parameters. While the score predicted by the default DT is fairly distant to the score of the near dataset sample one, the score predicted by the tuned DNN is much closer to the two data samples with the same tuning parameters. It is important to notice that this observation is highly theoretical as we cannot confirm it through testing the configurations on a real device. The conditions also have a influence on the score and differ between the dataset samples and the corresponding model. Therefore, it cannot be excluded that the predicted score is correct. However, keeping the MSE and R^2-score of the DT model in mind, it seems unlikely that the predicted score is accurate. Those problems arise the question of reliability and if the found configuration by the DT is even a maximum. It is not beneficial for our purpose to have a high but not reliable maximum configuration. In a best-case scenario, the configuration found by the model has the same score on a real device as the one predicted by the model.
According to this observation the MSE and R^2-score can be seen as a reliability score. A model like the tuned DNN is good at approximating even unseen data. Therefore, the predicted values by the model generally are more accurate than predictions by a model with a worse MSE and R^2-score, like the default DT. This assumption can also be applied on the maximum of the models.



(a) dataset **(b)** tuned DNN on testset **(c)** tuned GBT on testset

Figure 5.18: Comparison of the violin plots of the ground truth values of the whole dataset and the tuned DNN and tuned GBT on the testset

This experiment highlights one of the flaws of the MSE and R^2-score. The scores don't provide any information about the distribution of errors. A model may have a low MSE but still poorly approximate the maximum of the function.

Another interesting comparison to make is between the GBT model and the tuned DNN model. The GBT model has the second highest maximum. It still has a good MSE and R^2-score on the test data and does not overfit as much as the default DT model. The tuned DNN model is the overall best model from previous experiments. However, the GBT seems to be better fit for our problem as it seems to be a compromise of reliability and a high maximum. This might be due to the way the algorithm works. The algorithm iteratively tries to improve its performance in spaces where it performs poorly. Therefore, it actively improves itself in areas like the space around the maximum, resulting in a model approximating the maximum better. In contrary, the tuned DNN model averages and smoothes the maximum due to the sparse sampling in the maximum space. Comparing the violin plots of the two models further proves this assumption (see. Figure 5.18). Both models differ from the ground truth values of the whole dataset mainly in the space around the maximum. While the dataset is a bit sharper in the maximum space, both models accumulate values exceptionally around $y = 5$. In the remaining areas, both models approximate the function very well. This assertion is supported by the low MSE of both models on the test set. However, if you look at the maximum and minimum, the tuned DNN appears to be flatter, while the GBT model has a higher and sharper maximum and minimum, which is closer to the shape of the actual maximum and minimum. The GBT was already interesting due to its performance. The model looks even more promising for its ability to approximate the optimum of the function more accurately than other models.

In conclusion, many of our examined models struggle with approximating the sparsely sampled space around the maximum. This is sometimes due to averaging or the way our models have been trained or work. Our models were trained to have a low MSE in general, aiming on approximating the whole function well. Because most of the samples are not located around the extremas of the function, the models were still able to achieve a good MSE and R^2-score. GBT was still able to approximate the maximum fairly well, due to how the algorithm works. As we can't change the data we are working on, a future approach could be to design a loss function prioritising the higher scores in the dataset. Furthermore, we could use this score to hyperparameter optimize DNN models, resulting in models built to perform especially well at approximating the maximum of our black-box function.

For our next experiment we will further examine the GBT and tuned DNN. The GBT model showed to have a high maximum while maintaining a good quality and performance. The DNN model is slower but has the highest quality and a high robustness as conducted in previous experiments. Furthermore its smoother function could have a beneficial impact on the convergence of the RL process. This should enable us to investigate the impact of performance, quality and robustness on the RL process.

5.5 Learn-to-Optimize

In the following experiment we will study the influence of the surrogate models on the learn-to-optimize process. The surrogate models are trained as usual to the data. Then the AutoTune learn-to-optimize process trains the NN using RL. To evaluate the process, the c1 and c2 parameters are set to a worst-case scenario, and the NN and a Monte Carlo sampler find the optimal input parameters and FOM for each possible combination of the c3 and c4 parameters, uniformly spaced between -1.0 and 1.0 with a step size of 0.1. The experiment was run on the following hardware:

- **CPU:** 4x Xeon E7-8880v3 (72 Core, 144 Threads)
- **RAM:** 504 GB

Comparing the training times of the GBT and tuned DNN, the training with the GBT model took 52704s while the training with the DNN model only took 38952s. This result is surprising as, according to previous experiments, the GBT is the faster model at predicting samples. There are multiple possible reasons for this behaviour. TF has many built-in optimizations for their models which accelerate the tuned DNN. Especially on better hardware, where parallelization gets more important, those optimizations become noticeable. Furthermore, the RL process is implemented with TF. This leads to a better interaction between the TF surrogate models and the RL process compared to the Sk-learn models. TF is able to optimize the whole execution better when all components are from the TF module.

The resulting machine learning model / tuning law trained with the tuned DNN model is also faster at predicting the optimal parameters. The tuning law trained with the GBT model took on average 0.204s to predict the optimal parameters, while the tuned DNN model took 0.131s.

In Figure 5.19 the violin plot shows the distribution of the predicted optimal FOM values of the different method and surrogate models. As expected the GBT model has generally predicted higher maxima than the tuned DNN model using the Monte Carlo method. This reflects the results of the previous Monte Carlo experiment. However, the tuned DNN predicts noticeably higher FOM values using the learn-to-optimize method. The key difference between the both methods is that the Monte Carlo method is random based while the learn-to-optimize method is based on RL "searching" for the optimal solution. The Monte Carlo model therefore is not impacted by the robustness nor the shape of the function. In contrary, the RL process can get stuck in local optima if the function is rough. This could be the reason the smoother model performs better with the RL in the learn-to-optimize scenario than the GBT model. The impact of smoothness also can be seen in the shape of the violin plot. The distribution of the tuned DNN is way more condensed while the distribution of the GBT model is more outspread.

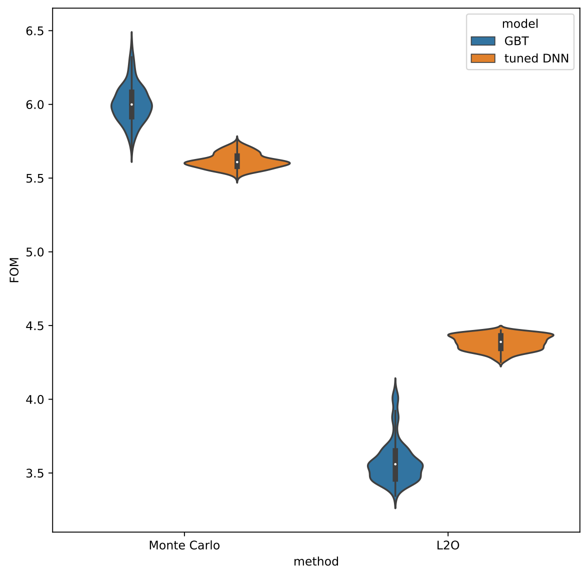


Figure 5.19: Violin plots of the distributions of the FOM values with the different methods and surrogate models

model name	training time	average prediction time
tuned GBT	52704s	0.205s
PSV	33656s	0.109s
tuned DNN	38952s	0.131s
tuned DT	45188s	0.113s
tuned ET	203881s	0.775s
tuned RF	268819s	1.047s

Table 5.18: The performance results for the different surrogate models

The FOM distributions of all models using the learn-to-optimize method are shown in Figure 5.19. This plot further proofs that the tuned DNN seems to be the most suitable model for the learn-to-optimize task, as the predicted FOM values are generally higher than the other models. In terms of performance the training time of the tuned DNN is one of the fastest, while the average prediction time is fairly moderate (see Table 5.18). Robustness and accuracy therefore appear to be the most important metrics for a surrogate model in the learn-to-optimize scenario. For a fast performing model in terms of training time and prediction time, TF models seem to be the most suitable for the AutoTune implementation, since the modules' learn-to-optimize is implemented with TF. The TF optimizations outweigh the faster prediction time of the Sk-learn models measured in our previous experiments. Future research could therefore focus more on the TF models and compare them in a

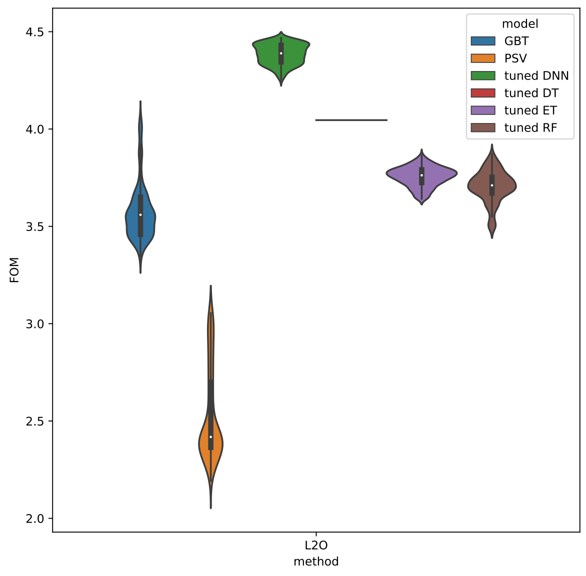


Figure 5.20: Violin plots of the distributions of the FOM values with the different surrogate models

learn-to-optimize scenario. At the same time, the Sk-learn models could be implemented in a TF manner using optimizations from the library. Finally, the AutoTune module could be optimized to use Sk-learn models in the learn-to-optimize scenario.

6 Future Work

In previous sections, we have already mentioned further research on this topic/scenario and possible ideas worth investigating. In this section, we will discuss some ideas we had during our work on this thesis. These ideas are based on our investigations and findings during the various experiments.
First, an obvious approach is to vary the parameters of different algorithms. Many of the algorithms and machine learning models we used in this thesis are highly configurable with many different parameters. Even though we performed hyperparameter optimizations, some parameters of various models were not explored, leaving potential space for further hyperparameter optimizations of the models. Especially the NN leave room for further configuration, trying different activation functions or even different types of NNs like convolutional neural networks or recurrent neural networks.
Finally, even the hyperparameter optimization algorithms themselves could be run with different parameters. Taking Bayesian optimization as an example, different acquisition functions or methods for defining the posterior distribution could be used.

Ensembles are another effective method to increase the performance or quality of a surrogate model. In this theses we only explored ensemble methods with DTs such as RF, ET and GBT. While the GBT model showed promising results for our scenario in different experiments. Gradient boosting appears to be suited for our scenario as it approximates the maximum fairly well. It is not limited to DT and could be applied to other machine learning models. Ensemble learning can be used with any number or variety of models, providing unlimited possibilities for future research. A very simple ensemble could consist of two models, one having an overall accurate and robust prediction and one solely trained on approximating the space around the maximum.
Generally our work shows that TF models tend to work best with the AutoTune module. Future work could focus more on the TF models and explore different types or architectures in the learn-to-optimize scenario. At the same time, Sk-learn models could be optimized for our use case with TF, or some of our promising algorithms could be manually implemented just for our purpose.

As mentioned in Section 5.4, our maximum is sparsely sampled, and most of our models failed to approximate it accurately. One idea to improve this behavior is to use different loss or score functions in training. These loss functions could improve the performance of our existing models in the learn-to-optimize process and facilitate the model selection process. Furthermore, the loss functions can be used to hyperparameter optimize different models, resulting in models that are better suited for our purpose. However, the models can only be improved to a certain point because we are limited by our dataset and the few samples around the maximum.
Additionally to tuning the surrogates to approximate the maximum more accurately, the surrogates could also be tailored especially for the RL process. As our work shows, more robust models with a smoother function lead to better results in the learn-to-optimize scenario. One possible idea is to map all insignificant values to a certain threshold, thus creating a plane in spaces that are not interesting for the search. This could possibly reduce the chance of the RL getting stuck in local minima or lead to faster convergence.

7 Conclusion

In this thesis, we studied the influence of different surrogate models in the context of learn-to-optimize and PSV. We have an interesting scenario where we have an initial dataset with measurements of different semiconductor circuits. Our surrogate models are based solely on this initial dataset. We introduced different machine learning algorithms and NN used as surrogate models. We optimized these models using various hyperparameter optimization algorithms for our dataset. Some hyperparameter optimization algorithms proved to be less suitable for our task. Surrogate models can have many different properties depending on the underlying algorithm. We implemented different evaluation methods and metrics and compared our optimized surrogate models on 3 core metrics: robustness, performance, and quality. All surrogate models were able to approximate the black box function quite well. However, we found significant differences in the performance, quality, and robustness of the models. However, no model was able to perform well on all metrics, leading to an interesting trade-off between the three metrics. We then used Monte Carlo sampling to find the maximum FOM or score value of each model. We evaluated these values, taking into account the different metrics of the model, such as overfitting. Most of the models failed to accurately approximate the maximum of the function due to sparse sampling around the maximum in the data set. Some models, such as the GBT, are better suited to the problem because they focus on approximating different parts separately. The averaging and smoothing of the underlying function of some models is detrimental to the approximation of the maximum. The RL method gives the best results with our TF models. In particular, the tuned DNN model has the best impact on the RL in terms of performance and finding a higher maximum. We came to the conclusion that the RL prefers models with a higher robustness, in contrast to the Monte Carlo method. Finally, we presented several ideas and approaches for future research and improvement of the interaction between surrogate models and learn-to-optimize.

Bibliography

[BB12] J. Bergstra, Y. Bengio. “Random Search for Hyper-Parameter Optimization”. In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305. URL: http://jmlr.org/papers/v13/bergstra12a.html (cit. on pp. 16, 33, 34, 37).

[CSM14] A. Cozad, N. V. Sahinidis, D. C. Miller. “Learning surrogate models for simulation-based optimization”. en. In: *AIChE Journal* 60.6 (2014). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/aic.14418, pp. 2211–2227. ISSN: 1547-5905. DOI: 10.1002/aic.14418. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/aic.14418 (visited on 01/16/2023) (cit. on p. 11).

[Dom22] P. Domanski. *AutoTune*. AutoTune is a package to automatically run data-driven performance tuning in post-silicon validation. 2022. URL: https://gitlab-sim.informatik.uni-stuttgart.de/domanspr/auto%5C_tune (cit. on pp. 9, 12).

[DPLR21] P. Domanski, D. Pflüger, R. Latty, J. Rivoir. “ORSA: Outlier Robust Stacked Aggregation for Best- and Worst-Case Approximations of Ensemble Systems”. In: *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2021, pp. 1357–1364. DOI: 10.1109/ICMLA52953.2021.00220 (cit. on p. 9).

[DPRL21] P. Domanski, D. Pflüger, J. Rivoir, R. Latty. “Self-Learning Tuning for Post-Silicon Validation”. In: *CoRR* abs/2111.08995 (2021). DOI: 10.48550/ARXIV.2111.08995. arXiv: 2111.08995. URL: https://arxiv.org/abs/2111.08995 (cit. on pp. 9, 12).

[FH19] M. Feurer, F. Hutter. “Hyperparameter optimization”. In: *Automated machine learning*. Springer, Cham, 2019, pp. 3–33 (cit. on p. 15).

[Fri01] J. H. Friedman. “Greedy function approximation: A gradient boosting machine.” In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451. URL: https://doi.org/10.1214/aos/1013203451 (cit. on p. 19).

[JT16] K. Jamieson, A. Talwalkar. “Non-stochastic best arm identification and hyperparameter optimization”. In: *Artificial intelligence and statistics*. PMLR. 2016, pp. 240–248 (cit. on p. 17).

[KB20] S. H. Kim, F. Boukouvala. “Machine learning-based surrogate modeling for data-driven optimization: a comparison of subset selection for regression techniques”. en. In: *Optimization Letters* 14.4 (June 2020), pp. 989–1010. ISSN: 1862-4480. DOI: 10.1007/s11590-019-01428-7. URL: https://doi.org/10.1007/s11590-019-01428-7 (visited on 01/22/2023) (cit. on p. 11).

[LJD+17] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, A. Talwalkar. “Hyperband: Bandit-Based Configuration Evaluation for Hyperparameter Optimization.” In: *ICLR (Poster)*. 2017, p. 53 (cit. on p. 17).

[LJD+18] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, A. Talwalkar. “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization”. In: *Journal of Machine Learning Research* 18.185 (2018), pp. 1–52. URL: http://jmlr.org/papers/v18/16-558.html (cit. on p. 17).

[LM16] K. Li, J. Malik. *Learning to Optimize*. arXiv:1606.01885 [cs, math, stat]. June 2016. DOI: 10.48550/arXiv.1606.01885. URL: http://arxiv.org/abs/1606.01885 (visited on 01/15/2023) (cit. on p. 12).

[MAP+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on p. 24).

[OBL+19] T. O’Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi, et al. *KerasTuner*. https://github.com/keras-team/keras-tuner. 2019 (cit. on pp. 17, 24).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 24).

[RCV+17] F. E. Rangel-Patiño, J. L. Chávez-Hurtado, A. Viveros-Wacher, J. E. Rayas-Sánchez, N. Hakim. “System Margining Surrogate-Based Optimization in Post-Silicon Validation”. In: *IEEE Transactions on Microwave Theory and Techniques* 65.9 (2017), pp. 3109–3115. DOI: 10.1109/TMTT.2017.2701368 (cit. on p. 11).

[tea22] T. pandas development team. *pandas-dev/pandas: Pandas*. Version v1.5.2. Nov. 2022. DOI: 10.5281/zenodo.7344967. URL: https://doi.org/10.5281/zenodo.7344967 (cit. on p. 24).

[VDHL17] K. K. Vu, C. D’Ambrosio, Y. Hamadi, L. Liberti. “Surrogate-based methods for black-box optimization”. In: *International Transactions in Operational Research* 24.3 (2017), pp. 393–424. DOI: https://doi.org/10.1111/itor.12292. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/itor.12292. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/itor.12292 (cit. on p. 11).

All links were last followed on March 17, 2023.

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