



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Neural Network Hyperparameter  
Optimization with Sparse Grids**

Maximilian Michallik





DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# **Neural Network Hyperparameter Optimization with Sparse Grids**

## **Parameteroptimierung von neuronalen Netzen mit dünnen Gittern**

Author: Maximilian Michallik  
Supervisor: Dr. Felix Dietrich  
Advisor: Dr. Michael Obersteiner  
Submission Date:



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich,

Maximilian Michallik

## Acknowledgments

# **Abstract**

In recent years, machine learning has gained much importance due to the increasing amount of available data. The models that are performing very different tasks have a thing in common. They have parameters that are fixed before being trained on the data. The right choice of those hyperparameters can have a huge impact on the performance which is why they have to be optimized. Different techniques like grid search, random search, and bayesian optimization tackle this problem.

In this thesis, a new approach called adaptive sparse grid search for hyperparameter optimization is introduced. This new technique allows to adapt to the hyperparameter space and the model which leads to less training and evaluation runs compared to normal grid search while still finding the optimal model configuration for the best model results.

We compare the new approach to the other three techniques mentioned regarding execution time and resulting model performance using different machine learning tasks. The results show that adaptive sparse grid search is very efficient with a model performance similar to bayesian optimization and grid search.

# **Zusammenfassung**

# Contents

<b>Acknowledgments</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 State of the Art</b>	<b>3</b>
2.1 Introduction to Neural Networks . . . . .	3
2.2 Hyperparameter Optimization . . . . .	6
2.2.1 Grid Search . . . . .	7
2.2.2 Random Search . . . . .	7
2.2.3 Bayesian Optimization . . . . .	8
2.2.4 Other Techniques . . . . .	12
2.3 Sparse Grids . . . . .	12
2.3.1 Numerical Approximation of Functions . . . . .	12
2.3.2 Adaptive Sparse Grids . . . . .	16
2.3.3 Basis Functions for Sparse Grids . . . . .	19
2.3.4 Optimization with Sparse Grids . . . . .	20
2.4 Adaptive Random Search . . . . .	24
<b>3 Hyperparameter Optimization with Sparse Grids</b>	<b>26</b>
3.1 Methodology . . . . .	26
3.1.1 Adaptive Grid Search with Sparse Grids and Iterative Adaptive Random Search . . . . .	26
3.1.2 Implementation . . . . .	26
3.2 Sparse Grid Optimization of Functions . . . . .	28
3.2.1 Sparse Grid Generation with different Adaptivities . . . . .	30
3.2.2 Local and Global Optimization . . . . .	34
3.3 Hyperparameter Optimization with Sparse Grids . . . . .	39
3.3.1 Optimum Approximation with Sparse Grids . . . . .	39
3.3.2 Optimization on Sparse Grids . . . . .	42

3.4	Comparison with Grid-, Random Search and Bayesian Optimization . . . . .	48
3.4.1	Two-dimensional Experiment of Regression with Small Neural Network . . . . .	48
3.4.2	Three- and Five-dimensional Experiments of Regression with Small Neural Network . . . . .	53
3.4.3	Nine-dimensional Experiment with MNIST Dataset . . . . .	56
3.4.4	Comparison with Implementation of other Authors . . . . .	58
3.5	Iterative Adaptive Random Search . . . . .	60
3.5.1	Implementation . . . . .	60
3.5.2	Analysis of Parameters with Functions . . . . .	66
3.5.3	Hyperparameter Optimization . . . . .	76
3.5.4	Comparison with other Techniques . . . . .	76
<b>4</b>	<b>Conclusion and Outlook</b>	<b>77</b>
<b>List of Figures</b>		<b>78</b>
<b>List of Tables</b>		<b>82</b>
<b>Bibliography</b>		<b>84</b>

# 1 Introduction

Machine learning has gained enormous importance in many fields of application. In many areas of daily life, artificial intelligence can increase productivity by automatically making decisions. There are many types of algorithms and architectures to reach this goal. They all have one thing in common, which is that they have to be defined and trained on data. In this process, the developer has to find a suitable configuration for the machine learning algorithm. Some of them depend on each other and for most of them, expert knowledge can help find the right values. Automating this task of hyperparameter optimization or tuning can be done without expert knowledge or an approach called babysitting which is just trying out different configurations until a reasonably good performance is achieved. This is also called trial and error and is often applied as can be seen in [1, 2]. This technique can take some time to find a good configuration which is why there are already many algorithms to automate this.

All existing techniques like grid-, random search, and bayesian optimization have advantages and shortcomings depending on the concrete scenario. In the scope of this thesis, a new approach is introduced which uses sparse grids. They have the general advantage that there is no *curse of the dimensionality* in higher dimensions compared to a general homogeneous grid because the number of grid points does not scale exponentially. We present the new technique and compare it using different types of machine learning models and various datasets and tasks like regression and classification.

We additionally introduce an iterative adaptive random search approach which combines the advantages of random search and iterative methods like the bayesian optimization or the adaptive sparse grid search.

Therefore, in Chapter 2, we give a brief introduction to machine learning with neural networks, present other well-analyzed hyperparameter optimization techniques like grid-, random search, and bayesian optimization, and explain the concept of adaptive sparse grid which is the base for the new approach. In the following in Chapter 3, we explain the Methodology and how the sparse grid optimization method works. First, normal functions are optimized because we know the optimal points. Later on, we transfer the findings to the machine learning model evaluation and further analyze the

behavior in different settings. After finding suitable configurations for the sparse grid, we compare the implemented algorithms with different datasets and tasks followed by an introduction of the iterative adaptive random search. This method is also analyzed and compared to the existing approaches. In the end, we give a conclusion and outlook in Chapter 4.

## 2 State of the Art

Machine Learning [3, 4] is a rapidly evolving field of artificial intelligence. There are different types of algorithms that are used for specific tasks involving supervised learning where the algorithm maps inputs to the given labels, unsupervised learning where the labels to the input are not available, and semi-supervised learning which combines labelled and unlabeled data. Additionally, there is reinforcement learning where the model learns by observing the environment [5]. Specific tasks are e.g. classification where the input has to be assigned to specific classes, regression where the input has to be assigned to a continuous value (both supervised) and clustering (unsupervised) where the goal is to group the input.

There are many different algorithms that accomplish these goals, for example support vector machines [6], the tsetlin machine [7], and decision trees [8]. One very important class of algorithms is *artificial neural networks* 2.1. After the introduction to neural networks, hyperparameter optimization is presented with different techniques to improve machine learning models. In the following, sparse grids are presented which will be needed as a foundation for hyperparameter optimization of neural networks with sparse grids.

### 2.1 Introduction to Neural Networks

Neural networks [9, 10] are very powerful for solving various tasks. They are very versatile and they exist in very different variations, ranging from a very small size up to very large networks for more complex tasks.

The smallest part of a neural network is the *perceptron*. A network consisting only of one perceptron can be seen in Figure 2.1.

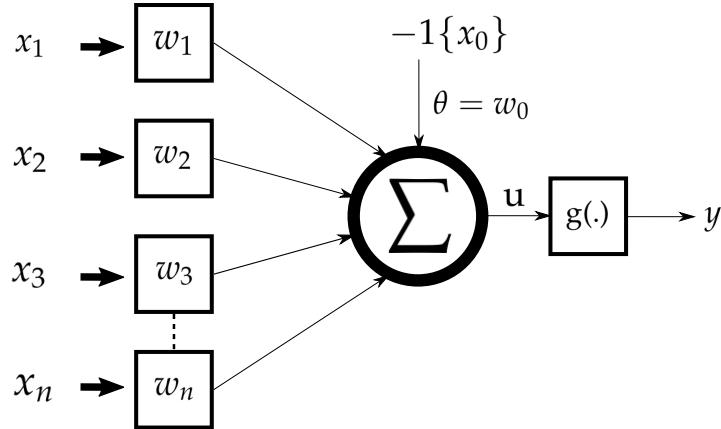


Figure 2.1: Neural network consisting of only one perceptron. The output is computed according to Equation 2.1.

The output  $y$  is computed with

$$u = \sum_{i=1}^n w_i \cdot x_i - \theta, y = g(u). \quad (2.1)$$

The network has  $n$  inputs  $x_i$  and weights  $w_i$ .  $\theta$  is the activation threshold (also called bias),  $g$  is the activation function, and  $u$  is the activation potential [10].

This basic building block can then be used to build a more complex architecture with multiple layers. All neural networks have an input layer consisting of  $n \in \mathbb{N}$  input neurons and an output layer with  $m \in \mathbb{N}$  output values. Between them, there can be multiple hidden neural layers. In deep neural networks, this number of layers is very high as the name suggests. Each neuron of each layer again has weights of the corresponding input and bias. The concrete values for them are important for the behavior of the model and determine the performance. These values are learned during the training phase of the model. There are two stages (forward and backward stage) during the training phase as can be seen in Figure 2.2.



Figure 2.2: Neural network consisting of two hidden layers. The connections between the perceptrons are bidirectional. In the forward phase, the intermediate results are given to the neurons to the right and in the backward phase from right to left.

The figure shows a schematic neural network with two hidden layers and  $n_1$  neurons in the first layer and  $n_2$  ones in the second layer. In the forward stage, an input  $x \in \mathbb{R}^n$  is put into the network and the output is computed by computing the corresponding result of each neuron and feeding it into the next layer to the right according to the connections. This output is then taken to update the weights of all neurons in the backward stage. In the simple case depicted in Figure 2.1 with only one perceptron, the weights are updated with

$$w_{current} = w_{previous} + \eta \cdot (d^{(k)} - y) \cdot x^{(k)} \quad (2.2)$$

where  $w = [\theta \ w_1 \ ... \ w_n]^T$  is the vector with all weights and the bias,  $x = [-1 \ x_1^{(k)} \ ... \ x_n^{(k)}]^T$  is the  $k^{\text{th}}$  training sample,  $d^k$  the desired label,  $y$  the output of the perceptron and  $\eta$  the learning rate. The choice of  $\eta$  is fixed before training and usually  $0 < \eta < 1$ . For the update of the weights of networks with multiple layers, refer to [10].

The perceptron and its training is the basic building block for most neural networks. Based on this, the concrete architecture can still be adjusted. One first thing is to increase the number of layers or neurons per layer. But also the choice of connections between layers can improve the performance of the network. Another thing that can be done is to introduce connections from higher layers to lower layers which makes it

a recurrent neural network. They are especially suited for sequential or time-varying patterns [11]. There is also a specific architecture for grid-structured data like images. For them, convolutional neural networks are used to extract features [12–14]. For further readings on different architecture choices, refer to [15].

In all cases, the weights of the network are updated automatically and it is impossible to understand the concrete decision-making of the model. The weights are called parameters of the network. Besides them, there are hyperparameters that have to be fixed before training. They are design decisions on how the network should behave. For some of them, experience can show which choices lead to better performances of the model but in all cases, they can be optimized which will be discussed in the following section 2.2. Some of the hyperparameters are

- Epochs: Number of times the training data is fed into the network and the weights are updated
- Learning rate:  $\eta$  of Equation 2.2 defining how fast the model should learn
- Optimizer: Optimizer used to update the weights of the network
- Loss function: Concrete loss metric how the label and the output are compared in Equation 2.2
- Batch size: Number of data samples processed in a batch
- Number of layers of the network
- Number of neurons in each layer

All these parameters can drastically influence the model performance. In the following section, different techniques for the optimization are presented.

## 2.2 Hyperparameter Optimization

Most machine learning models have parameters that have to be defined before the learning phase. They are called hyperparameters and strongly influence the behavior of the model. One example is the number of epochs of the learning phase of a neural network. There are different techniques for the optimization of hyperparameters and they all define the machine learning model as a black box function  $f$  with the hyperparameters as input and the resulting performance as output. The overall goal is to find a configuration  $\lambda_{min}$  from  $\Lambda = \Lambda_1 \times \Lambda_2 \times \dots \times \Lambda_N$  that minimizes the function  $f$  with  $N$  hyperparameters with

$$\lambda_{min} = \arg \min_{\lambda \in \Lambda} f(\lambda). \quad (2.3)$$

In our case, the function  $f$  is a machine learning algorithm that is trained on a training set and evaluated on a validation set. With this, the minimization of e.g. the loss of the model optimizes the decisions it is making which leads to better prediction results. Note that one function evaluation of  $f$  is usually very expensive as the training of a machine learning model with many parameters and weights takes much time. The data set consists of  $\{(x_i, y_i) | x_i \in X, y_i \in Y, 0 \leq i \leq m\}$  with  $m$  being the number of data samples. The  $x_i$  is the input data to the model and the goal is that

$$\forall i : M(x_i) = y_i. \quad (2.4)$$

where  $M$  is the model. In the context of supervised learning, the whole data set is split into a training set which is used to optimize the model and a testing set to evaluate the performance on new, unseen data [16].

In summary, the goal is get evaluation scores on the testing data set which can be achieved with Equation 2.3. [17–19]

In the following, different techniques for the optimization are presented and discussed with their advantages and disadvantages.

### 2.2.1 Grid Search

The idea of the first approach for optimization is to discretize the domains of each hyperparameter and evaluate each combination. This suffers from the curse of the dimensionality as it scales exponentially with the number of hyperparameters. For  $d$  parameters and  $n$  values per hyperparameter,  $n^d$  different configurations are possible which all have to be evaluated.

One advantage of this method is that it is easy to implement and very simple. Also, the whole search space is explored evenly.

On the other hand, the curse of the dimensionality makes it very slow if the function evaluations are very expensive which is the case for most machine learning algorithms. Another drawback is that each hyperparameter only takes  $n$  different values. The comparison to random search can be seen in Figure 2.3.

### 2.2.2 Random Search

The next technique [20] is similar to the grid search because the idea is also to evaluate different hyperparameter configurations. In contrast to the previous one, random search generates for each run and for each parameter exactly one random value from an interval which has to be specified. For this approach, a budget  $b$  has to be given. This parameter determines the number of different combinations that are evaluated. A direct comparison of grid search and random search can be seen in figure 2.3.

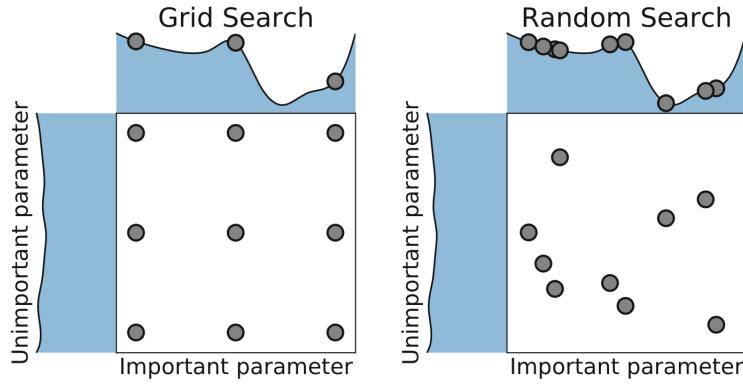


Figure 2.3: Comparison of grid search (left) and random search (right) in the two dimensional case. For both techniques, 9 different combinations are evaluated. In the left case, only 3 distinct values for each hyperparameter are set whereas there are 9 different values for each parameter in the random search. Taken from [17].

In this figure, a two-dimensional setting is depicted. For both techniques, 9 different combinations are evaluated. In the case of grid search, only 3 distinct values are taken for each hyperparameter while there are 9 different ones in the random search. In this example, the better result is found in random search as more distinct values are taken for the important parameter. Note that it is not always the case that random search leads to better results.

Compared to the normal grid search, this is one advantage. For each hyperparameter,  $b$  (budget) different values are taken into consideration which is much more compared to the grid search with the same overall number of combinations. Additionally, this technique is also easy to implement and relatively simple.

One disadvantage is that it is also very expensive if the budget is high because of the long training times of machine learning models.

### 2.2.3 Bayesian Optimization

Another possible technique for finding the best hyperparameters of machine learning models is called bayesian optimization (BO) [21]. This is an iterative approach for optimizing the expensive black box function by modelling it based on observations. A so-called *surrogate model*  $\hat{f}$  is made with the help of the *archive*  $A$  which contains observed function evaluations. This surrogate model is created by regression and the technique which is most often used is the Gaussian process [18] which is only

suitable if the number of hyperparameters is not too high [22]. The problem of this technique arises when some hyperparameters are categorical or integer-valued which is the reason why extra approximations can lead to worse results and special treatment is needed [23]. Another possible technique for the surrogate model is using random forests [24]. All in all, this function estimates the machine learning model depending on the hyperparameter configuration and also the prediction uncertainty  $\sigma(\lambda)$ . A second function called *acquisition function*  $u(\lambda)$  is built based on the prediction distribution. This  $u$  is responsible for the trade-off between exploitation and exploration. This means that configurations that lead to better model performances are exploited and values where not much information is gathered are explored. There are many numerous different possibilities for this function [25] but the most used one is the *expected improvement* (EI) which is calculated with

$$E[I(\lambda)] = E[\max(f_{min} - y, 0)]. \quad (2.5)$$

If the model prediction  $y$  with configuration  $\lambda$  follows a normal distribution [17], it leads to

$$E[\max(f_{min} - y, 0)] = (f_{min} - \mu(\lambda))\Phi\left(\frac{f_{min} - \mu(\lambda)}{\sigma}\right) + \sigma\phi\left(\frac{f_{min} - \mu(\lambda)}{\sigma}\right) \quad (2.6)$$

with  $\phi$  and  $\Phi$  being the standard normal density and standard normal distribution and  $f_{min}$  the best result so far.

In each iteration, a new candidate configuration  $\lambda^+$  is generated by optimizing the acquisition function  $u$ . This  $u$  is much cheaper to evaluate than the  $f$  which includes learning of an expensive neural network which makes the optimization much easier.

The exact steps are presented in Algorithm 1 and Figure 2.4 shows schematic iteration steps.

**Algorithm 1** Bayesian Optimization for a black box function  $f$ . In each iteration, the surrogate model is fitted on the current archive and an acquisition function is built. The optimum of this acquisition function is evaluated and added to the archive.

---

```

Generate initial  $\lambda^{(1)}, \dots, \lambda^{(k)}$ 
Initialize archive  $A^{[0]} \leftarrow \left( \left( \lambda^{(1)}, f(\lambda^{(1)}) \right), \dots, \left( \lambda^{(k)}, f(\lambda^{(k)}) \right) \right)$ 
 $t \leftarrow 1$ 
while Stopping criterion not met do
    Fit surrogate model  $(f(\lambda), \sigma(\lambda))$  on  $A^{[t-1]}$ 
    Build acquisition function  $u(\lambda)$  from  $(\hat{f}(\lambda), \sigma(\lambda))$ 
    Obtain proposal  $\lambda^+$  by optimizing  $u : \lambda^+ \in \arg \max_{\lambda \in \Lambda} u(\lambda)$ 
    Evaluate  $f(\lambda^+)$ 
    Obtain  $A^{[t]}$  by augmenting  $A^{[t-1]}$  with  $\left( \lambda^{(+)}, f(\lambda^{(+)}) \right)$ 
     $t \leftarrow t + 1$ 
end while
return  $\lambda_{best}$ : Best-performing  $\lambda$  from archive or according to surrogates prediction

```

---

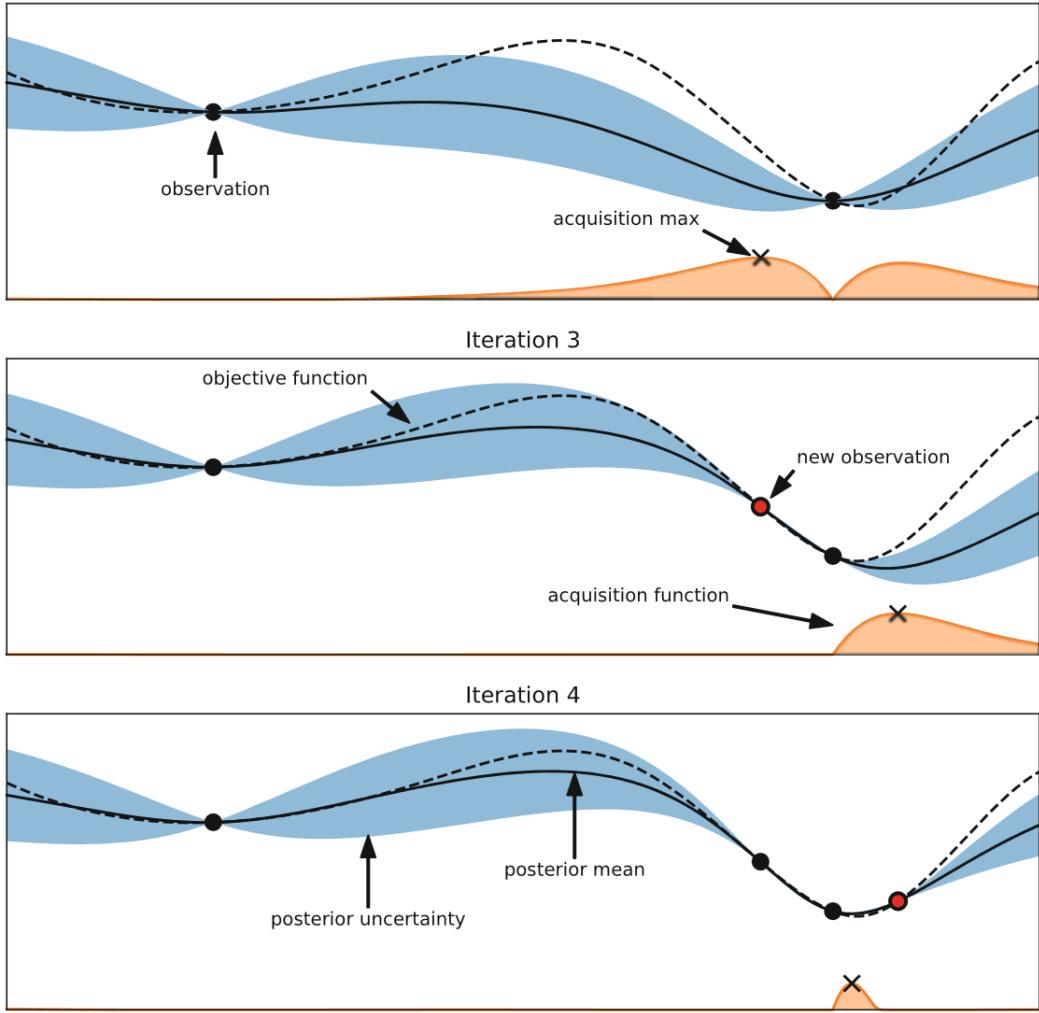


Figure 2.4: Schematic iteration steps of the bayesian optimization. The maximum of the acquisition function determines the next function evaluation (red dot in the middle). The goal is to find the minimum of the dashed line. The blue band is the uncertainty of the function. Taken from [17].

First,  $k$  initial hyperparameter configurations are sampled and evaluated. This set is the starting archive  $A^{[0]}$ . After that, the loop is executed as long as the stopping criterion is not met. This can be for example a budget, meaning a maximum number of function evaluations. The first step of the loop is to fit the surrogate model on the current archive. Then the acquisition function is made and optimized to get the next configuration  $\lambda^+$ . This point is evaluated and added to the archive. The overall result

of the algorithm is the  $\lambda$  which is the hyperparameter configuration for the machine learning model with the overall best result.

#### 2.2.4 Other Techniques

There are also other techniques for finding the best hyperparameters. Multi-fidelity optimization [17] aims to probe the learning of a model on a task with reduced complexity such as a subset of the data or fewer epochs for training the model for discovering the best configurations. For example, the learning curve can be predicted so that early stopping can be done if the prediction is not as good as the best model so far. There are also bandit-based selection methods that do not predict the learning curve but compare the different combinations on a small number of epochs and only perform the best ones. This can be done iteratively like it is done in *successive halving* for hyperparameter optimization [26]. The algorithm is very simple. It starts to evaluate all different combinations with a very small budget. The best half of the candidates are then evaluated in the next iteration with a double budget and so on until only one combination is left. In [27], a similar algorithm is presented. The authors use a model of the objective function (neural network depending on configurations) to find candidate hyperparameters. Those are then trained on a smaller number of epochs and the best ones are then evaluated with a higher budget. Also, neural networks can be used for the optimization which was done by the authors in [28]. Also, the covariance matrix adaptation evolution strategy was implemented as an alternative to bayesian optimization in [29].

### 2.3 Sparse Grids

Sparse grids are a useful tool to mitigate the *curse of the dimensionality* by reducing the number of grid points. In the following, this technique is presented after the general numerical approximation of functions.

#### 2.3.1 Numerical Approximation of Functions

Let  $f : \Omega \rightarrow R$  be a function defined on the unit interval  $\Omega = [0, 1]^d$  in  $d$  dimensions. For simplicity, we first set  $d = 1$ . Now this function can be represented on a grid of level  $l \in \mathbb{N}_0$  with  $2^l + 1$  grid points which are

$$x_{l,i} = i * h_l, \quad i = 0, \dots, 2^l, \quad (2.7)$$

with  $i$  being the index and  $h_l = 2^{-l}$  being the distance between the grid points. Each

of them gets a basis function defined by

$$\varphi_{l,i} : [0, 1] \rightarrow \mathbb{R}. \quad (2.8)$$

There are different possibilities for the basis functions which will be presented later. For the simplicity, we present a simple example being the hat function defined by

$$\varphi_{l,i}(x) = \max \left( 1 - \left| \frac{x}{h_l} - i \right|, 0 \right). \quad (2.9)$$

All in all, the space of functions that can be presented exactly by a linear combination is called the *nodal space*  $V_l$  with the assumption that the basis functions form a basis:

$$V_l = \text{span} \left\{ \varphi_{l,i} | i = 0, \dots, 2^l \right\}. \quad (2.10)$$

Every function  $f : [0, 1] \rightarrow \mathbb{R}$  can be interpolated by a the interpolant  $u$  defined by

$$f_l = \sum_{i=0}^{2^l} \alpha_{l,i} \varphi_{l,i}, \forall i = 0, \dots, 2^l : f_l(x_{l,i}) = f(x_{l,i}) \quad (2.11)$$

for constants  $\alpha_{l,i} \in \mathbb{R}$ . An example can be seen in Figure 2.5.

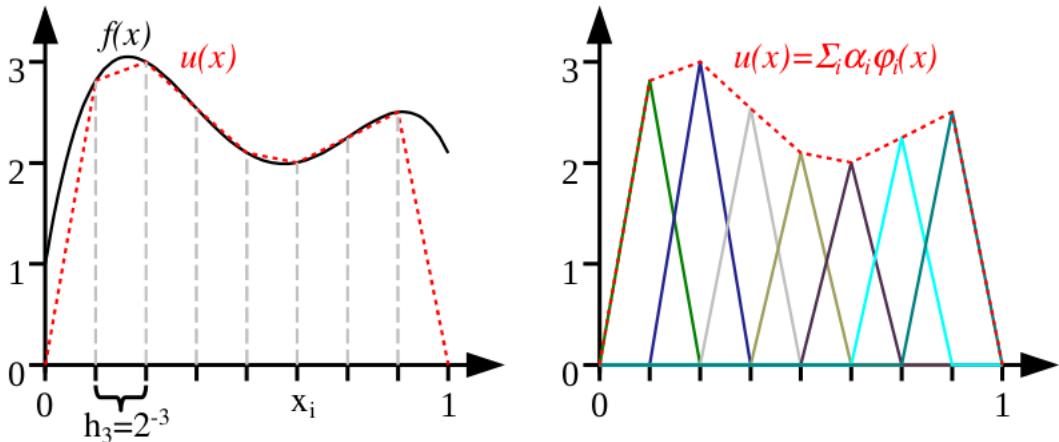


Figure 2.5: Interpolation of the function  $f$  (black line) by its interpolant  $u$  (red, dashed) in the nodal basis. Level of the grid is 3 and hat functions are used. Taken from [30].

On the left side, the function  $f$  (black line) can be seen with a grid of level 3. On the right side, the interpolant  $u$  as a linear combination of the basis functions (hat functions

centered on the grid points) can be seen. This approach is the nodal basis. The second possibility is called hierarchical basis and the index set is  $I_l^h = \{i \in \mathbb{N} | 1 \leq i \leq l, i \text{ odd}\}$ . The hierarchical subspaces are then

$$W_l = \text{span} \left\{ \varphi_{l,i}(x) | i \in I_l^h \right\}. \quad (2.12)$$

The same nodal space  $V_l$  can be obtained with the hierarchical subspaces with

$$V_l = \bigoplus_{i \leq l} W_i. \quad (2.13)$$

An example can be seen in Figure 2.6.

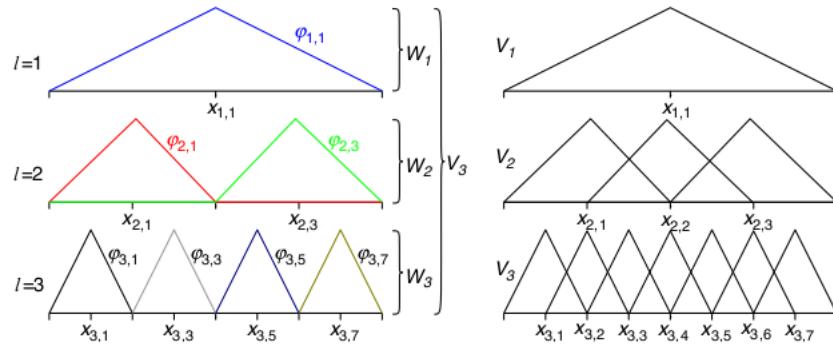


Figure 2.6: Hierarchical subspaces up to level 3 on the left. On the right, nodal spaces up to level 3. The combination of  $W_1$  up to  $W_3$  is the same space as  $V_3$ . Taken from [30].

On the left, the hierarchical subspaces up to level 3 can be seen. All in all, combined they span the same space as  $V_3$ . In the hierarchical case, a function  $f$  can also be interpolated by its interpolant  $u$  by

$$u = \sum_{i \in I_l^h} \alpha_{l,i} \varphi_{l,i}, \forall i = 0, \dots, 2^l : u(x_{l,i}) = f(x_{l,i}). \quad (2.14)$$

An example can be seen in Figure 2.7.

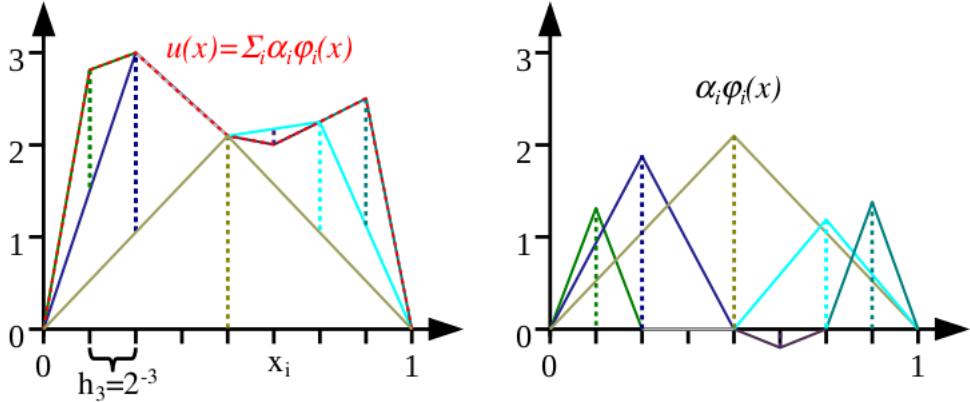


Figure 2.7: Interpolation of the function  $f$  (black line) by its interpolant  $u$  (red, dashed) in the hierarchical basis. Level of the grid is 3 and hat functions are used. Taken from [30].

To get into higher dimensions  $d > 1$ , we use the tensor product. The domain is now  $\Omega = [0, 1]^d$  and the level is defined by the level per dimension meaning  $\vec{l} = (l_1, \dots, l_d) \in \mathbb{N}_0^d$ . The index set is then

$$I_{\vec{l}} = \left\{ \vec{i} \mid 1 \leq i_j \leq 2^{l_j} - 1, i_j \text{ odd}, 1 \leq j \leq d \right\} \quad (2.15)$$

and the subspaces

$$W_{\vec{l}} = \text{span} \left\{ \varphi_{\vec{l}, \vec{i}}(\vec{x}) \mid \vec{i} \in I_{\vec{l}} \right\} \quad (2.16)$$

with the basis functions  $\varphi_{\vec{l}, \vec{i}} = \prod_{j=1}^d \varphi_{l_j, i_j}(x_j)$  which are constructed with the tensor product. The function space  $V_n$  is constructed by

$$V_n = \bigoplus_{\|\vec{l}\|_\infty \leq n} W_{\vec{l}} \quad (2.17)$$

with  $|\vec{l}| = \max_{1 \leq i \leq d} |d_i|$ . Again, a function can be interpolated by its interpolant  $u$  with

$$u = \sum_{\|\vec{l}\|_\infty \leq n, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l}, \vec{i}} \varphi_{\vec{l}, \vec{i}}, \forall \vec{i} \in I_{\vec{l}} : u(x_{\vec{l}, \vec{i}}) = f(x_{\vec{l}, \vec{i}}). \quad (2.18)$$

The resulting regular grid has then  $(2^n - 1)^d$  basis points. An example of a basis function in two dimensions can be seen in Figure 2.8. It is constructed by the tensor product of two 1d hat functions.

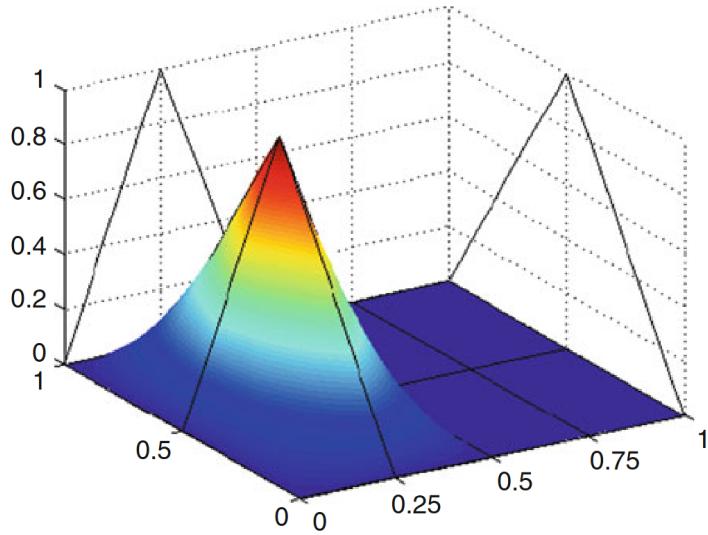


Figure 2.8: Example of a basis function in two dimensions. It is constructed with the tensor product of two 1d hat functions. Taken from [31].

In the higher dimensional case, the grid can also be constructed hierarchically. The proof that the hierarchical splitting given by

$$V_{\vec{l}} = \bigoplus_{\vec{m}=0}^{\vec{l}} W_{\vec{m}} \quad (2.19)$$

with  $W_{\vec{l}} = \text{span} \left\{ \varphi_{\vec{l}, \vec{i}} \mid \vec{i} \in I_{\vec{l}} \right\}$ ,  $I_{\vec{l}} = I_{l_1} \times \dots \times I_{l_d}$  holds for the basis with hat functions can be found in [32].

### 2.3.2 Adaptive Sparse Grids

The problem of regular grids is the *curse of the dimensionality* because of the high number of grid points in higher dimensions. This is tackled by sparse grids [33, 34] by reducing this number. The first technique to achieve this is by just leaving out subspaces. The resulting sparse function space is given by

$$V_n^1 = \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}} \subset V_n. \quad (2.20)$$

An example with  $n = 3$  can be seen in Figure 2.9.

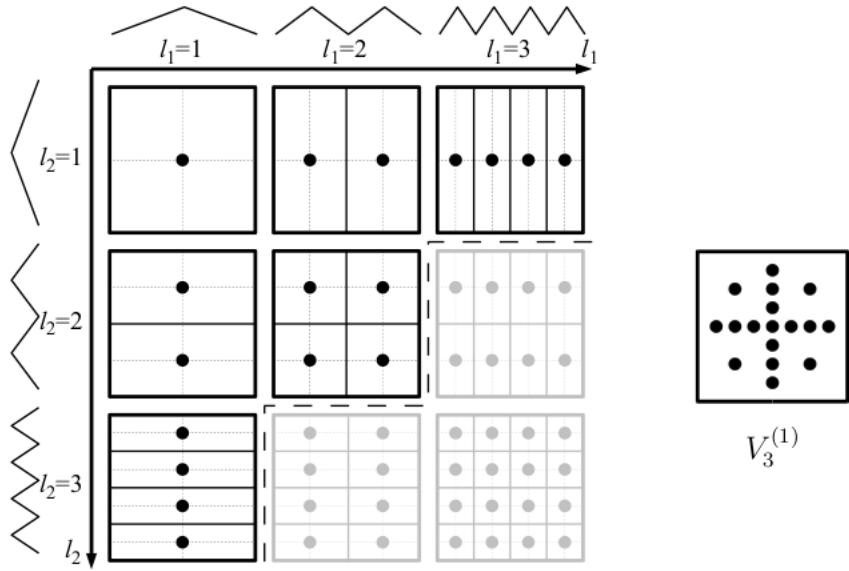


Figure 2.9: Two dimensional example of a sparse grid with  $n = 3$ . Left, the subspaces  $W_{\vec{l}}$  can be seen and on the right is the resulting sparse grid. Taken from [31].

An interpolant  $u_n$  of a function  $f$  is then constructed by

$$u_l = \sum_{|\vec{l}|_1 \leq l+d-1} \sum_{\vec{i} \in I_{\vec{l}}} \varphi_{\vec{l}, \vec{i}} \alpha_{\vec{l}, \vec{i}} \quad (2.21)$$

where the  $\alpha_{\vec{l}, \vec{i}}$  are the coefficients of the basis functions [35].

A second approach for sparse grids exists. The so-called *combination technique* [36] combines anisotropic full grids to get the same subspace as the conventional sparse grid approach. This has the advantage that we can use normal full grid operations on each subspace which will then be combined. This implies the possibility of parallelization. The combined solution can be computed with

$$u_l^c = \sum_{\vec{l} \in I} u_{\vec{l}} c_{\vec{l}} \quad (2.22)$$

where  $\vec{l}$  is the level vector of the full grid solution  $u_{\vec{l}}$ ,  $c_{\vec{l}}$  is a scalar factor, and  $I$  is the set of included level vectors. For a standard sparse grid, this evaluates to

$$u_l^c = \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{\vec{l} \in I_{l,q}} u_{\vec{l}} \quad (2.23)$$

with  $I_{l,q} = \left\{ \vec{l} \in \mathbb{N}_0^d : \left| \left| \vec{l} \right| \right|_1 = l + d - 1 - q \right\}$  [37]. An example of the 2-dimensional combination technique can be seen on the left side of Figure 2.10.

With the normal combination technique, this grid is still symmetric and focuses on a low global error. Especially in optimization or data-driven problems where the points are not distributed equally in the domain, special regions are of interest. In the case of optimization which is our focus, the errors around the extrema have to be interpolated more exactly than other regions. This is the reason why we use *refinement*. In the case of dimension-adaptive refinement [38], more grid points are added in the dimensions of higher relevance.

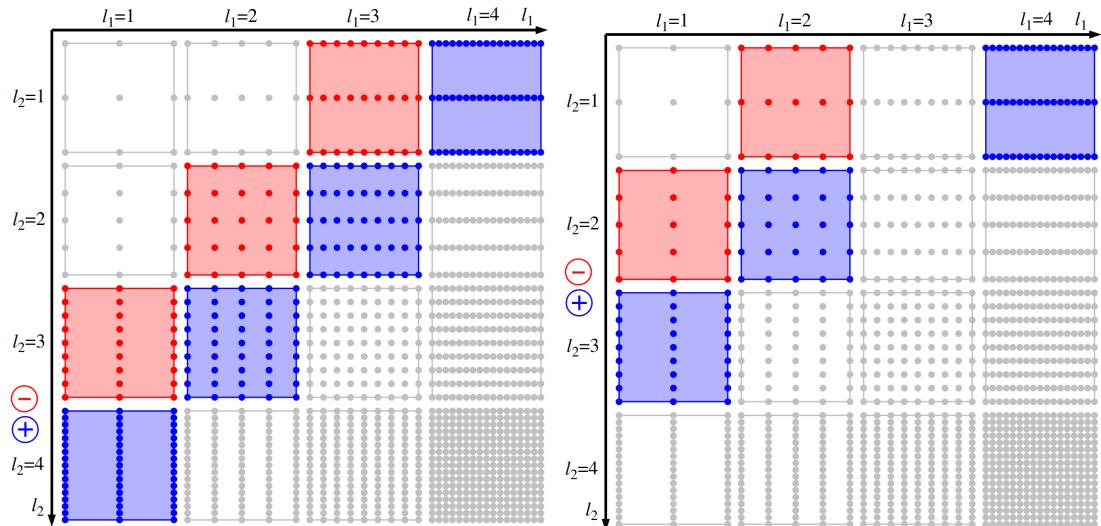


Figure 2.10: Example of the 2-dimensional combination technique. Here the blue regular grids are added and the red ones are subtracted. On the left, the normal combination technique can be seen and on the right is an dimension-adaptive version. Taken from [30].

In contrast to the previously mentioned refinement concentrating on whole dimensions, the *spatially adaptive refinement* directly adds grid points where the discretization error is still high. An example of the spatially adaptive combination technique presented by [37] can be seen in figure 2.11. In this example, the basis points of the component grids are no longer equidistant because refinement was already made.

In summary, Table 2.1 shows the comparison of full grids, sparse grids, and the combination technique in terms of number of the points and interpolation accuracy [30].

Table 2.1: Comparison of sparse grids, full grids, and the combination technique in terms of number of grid points and the accuracy.

Grid	Number grid points	Accuracy
Full grid	$\mathcal{O}(h_n^2)$	$\mathcal{O}(2^{nd})$
Sparse grid	$\mathcal{O}(h_n^{-1} (\log h_n^{-1})^{d-1})$	$\mathcal{O}(h_n^2 (\log h_n^{-1})^{d-1})$
Combination technique	$\mathcal{O}(d (\log h_n^{-1})^{d-1}) \times \mathcal{O}(h_n^{-1})$	$\mathcal{O}(h_n^2 (\log h_n^{-1})^{d-1})$

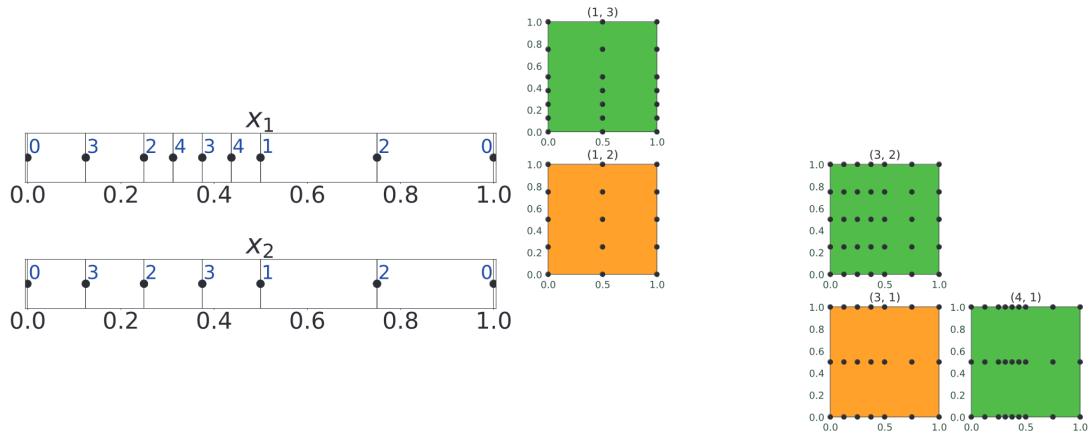


Figure 2.11: Example of the spatially adaptive combination technique in two dimensions. Taken from [37].

### 2.3.3 Basis Functions for Sparse Grids

So far, we only considered the simple case of the hat function on the support points. Besides them, there are other possibilities, for example piecewise d-polynomial, wavelet, and B-spline basis functions. For the first two cases, refer to [30, 34, 39] for further readings. In this thesis, we will concentrate on the B-spline basis for the sparse grids as the hat function is not continuously differentiable [32]. This is the reason why we can not compute globally continuous gradients which is a problem for the optimization. The general cardinal B-spline with degree  $p \in \mathbb{N}_0$  is defined by

$$b^p(x) = \begin{cases} \int_0^1 b^{p-1}(x-y) dy & p \geq 1 \\ \chi_{[0,1]}(x) & p = 0 \end{cases} \quad (2.24)$$

with  $\chi_{[0,1]}$  being the characteristic function of the half-open unit interval [40]. The  $b^p$  as defined above has the following 8 properties:

1. compactly supported on  $[0, p + 1]$
2. symmetric and  $0 \leq b^p \leq 1$
3. weighted combination of  $b^{p-1}$  and  $-b^{p-1}$
4. piecewise polynomial of degree  $p$
5.  $\frac{d}{dx} b^p$  is the difference of  $b^{p-1}$  and  $-b^{p-1}$
6. has unit integral
7. is the convolution of  $b^{p-1}$  and  $b^0$
8. hat function and gaussian function are special cases

This is the case for uniform B-splines. For adaptive grids, the distances between basis points are not always uniform. This is the reason why we need also non-uniform B-splines. Let  $m, p \in \mathbb{N}_0$  and  $\xi = (\xi_0, \dots, \xi_{m+p})$  be an increasing sequence of real numbers called *knot sequence*. For  $k = 0, \dots, m - 1$ , the non-uniform B-spline is defined by

$$b_{k,\xi}^p(x) = \begin{cases} \frac{x - \xi_k}{\xi_{k+p} - \xi_k} b_{k,\xi}^{p-1}(x) + \frac{\xi_{k+p+1} - x}{\xi_{k+p+1} - \xi_{k+1}} b_{k+1,\xi}^{p-1}(x) & p \geq 1 \\ \chi_{[\xi_k, \xi_{k+1}]}(x) & p = 0 \end{cases} \quad (2.25)$$

This definition and the proof that the hierarchical splitting also holds for using the B-splines for restricted functions can be found in [32].

### 2.3.4 Optimization with Sparse Grids

In general, an optimization problem can be constrained or unconstrained. In the first case, additionally to finding an optimum, there is a constraint that has to be fulfilled. In the case of sparse grids within the standard hypercube, the input values are restricted to the interval  $[0, 1]^d$ . The optimization problem which is called *box-constrained* can be solved by defining the function outside the box as infinity with  $f(x) = +\infty$  for all  $x \notin \Omega = [0, 1]^d$ .

Depending on whether the optimization algorithm uses the gradient or not, it is called a gradient-based method or a gradient-free method, respectively. In the following, algorithms of both types are presented [32].

## Gradient-Free Methods

**Nelder Mead Method** This iterative algorithm stores  $d + 1$  vertices of a  $d$ -dimensional simplex in ascending order of function values. In each round, either reflection, expansion, outer contraction, inner contraction or shrinking is performed on the vertices. In this way, the simplex contracts around the optimum.

**Differential Evolution** This algorithm maintains a list of points which are iteratively updated by the weighted sum of the previous generation. The mutated vector is *crossed over* with the original vector entry by entry and the resulting points are only accepted if they have better function values.

**CMA-ES** CMA-ES (covariance matrix adaption, evolution strategy) keeps track of the covariance matrix of the Gaussian search distribution. After sampling  $m$  points from the current distribution, the  $k$  best samples are used to calculate the distribution of the next iteration as the weighted mean of them. Then the covariance matrix is updated.

**Gradient-Based Methods** Important values for the following methods are the *gradient*  $\nabla_x f(x_k)$  and the *Hessian*  $\nabla_x^2 f(x_k)$ . Most methods of this type update the current position in each iteration with

$$x_{k+1} = x_k + \delta_k d_k \quad (2.26)$$

where  $\delta_k$  is the step size and  $d_k$  is the search direction.

**Gradient Descent** This method uses the gradient and sets the search direction to the standardized negative gradient at this point with  $d_k \propto -\nabla_x f(x_k)$ . If the Hessian is ill-conditioned, then the convergence is slow.

**NLCG** NLCG (non-linear conjugate gradients) is equivalent to the conjugate gradient method when optimizing function of the form  $f(x) = \frac{1}{2}x^T Ax - b^T x$ . It finds the optimum after  $d$  steps for strictly convex quadratic functions. According to the Taylor theorem, it converges also for non-convex functions that are three times continuously differentiable with positive definite Hessian because those functions are similar to strictly convex quadratic function in the region of the optimum.

**Newton** This method replaces the objective function with the second-order Taylor approximation  $f(x_k + d_k) \approx f(x_k) + (\nabla_x f(x_k))^T d_k + \frac{1}{2} (d_k)^T (\nabla_x^2 f(x_k)) d_k$  and sets the search direction to  $d_k \propto -(\nabla_x^2 f(x_k))^{-1} \nabla_x f(x_k)$ . This way  $x_k + d_k$  is the minimum of the approximation.

**BFGS** BFGS (Broyden, Fletcher, Goldfarb, Shanno) is a quasi-newton method. The previous technique has the disadvantage that the Hessian has to be evaluated which is expensive. BFGS approximates this matrix by a solution of  $\nabla_x^2 f(x_k) (x_k - x_{k-1}) \approx \nabla_x f(x_k) - \nabla_x f(x_{k-1})$ .

**Rprop** Rprop (resilient propagation) is not dependent of the exact direction of the gradient of the function but often works robustly in machine learning scenarios. The gradient entries are considered separately for each dimension and the entries of the current point  $x_k$  are updated depending on the sign of the gradient entry. Also the step size is adapted dimension-wise.

For constrained optimization methods, refer to [32]. There, the optimal point has to be found while constraining another function  $g$  with  $x_{opt} = \text{argmin}_f(x)$ , such that  $g(x) \geq 0$ .

One application of the optimization with sparse grids is presented in [41]. The goal was to solve forward-dynamics simulations of three-dimensional, continuum-mechanical musculoskeletal system models. The authors use B-splines on sparse grids for surrogates of the muscle model and use it in simulations that are subject to constraint optimization.

An alternative approach for global optimization of a function is presented by [42]. One application is dealing with induction motor parameter estimation [43]. There, the search space is discretized using the hyperbolic cross points (HPC). In one dimension, the points of level  $k$  are defined with

$$x = \pm \sum_{j=1}^k a_j 2^{-j}, a_j \in \{0, 1\}. \quad (2.27)$$

The representation of those points (but not 0) is unique, for example  $0.375 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$ . The level of a three-dimensional point  $[0.25, 0.375, 0]$  is 5. Figure 2.12 shows the HPC for  $k = 5$ .

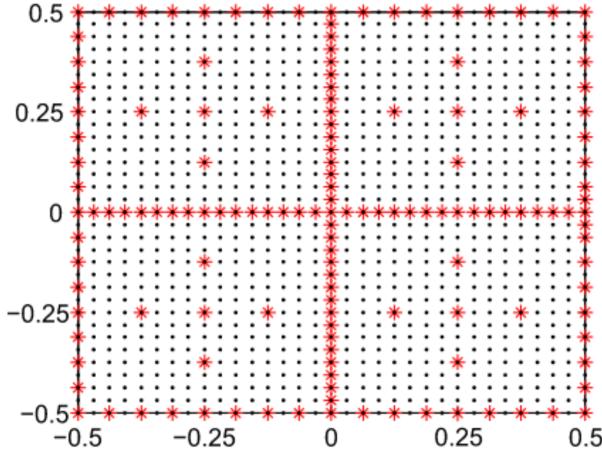


Figure 2.12: HPC of level  $k = 5$  (red stars) and full grid (black dots). A full grid would have  $33^2 = 1089$  and this grid has 147 points. Taken from [43].

A full grid of this level has 1089 points, whereas the number of HPCs is 147 which is much less, although the accuracy is nearly as good as using the full grid with  $\mathcal{O}(N^{-2} \cdot (\log N)^{d-1})$  with  $N$  being the number of grid points in one dimension. Now with the reduced number of search points, the optimization is made faster while still getting accurate results.

The optimization problem generally applies in various scientific fields. The authors of [44] present an application for path planning of car-like robots. The same global minimization approach based on sparse grids is used as in the previous application using the HPCs [42, 43].

Another method for optimization is presented in [45]. The authors introduce an optimization scheme based on sparse grids and smoothing that is derivative-free. It is an iterative algorithm for finding a local optimum.

Similar to this method, the authors of [46] also present a derivative-free optimization based on sparse grid numerical integration. Their technique applies to smooth nonlinear objective functions where the gradient is not available and point evaluations are very expensive.

Also, the authors of [47] present an optimization algorithm with the help of sparse grids. They use the collocation scheme and it can be used in a wide range of applications. Examples that are presented include stochastic inverse heat conduction problems and contamination source identification problems.

## 2.4 Adaptive Random Search

As well as the sparse grid optimization, an adapted random search approach can also be used for finding the minimum of arbitrary functions. The so-called adaptive random search was also already used in different settings and multiple variations.

One example is presented in [48]. The authors introduce Adaptive Random Search Technique (ARSET) and compare its performance with other optimization approaches. The algorithm dynamically adapts the search space depending on the current and new function value, as well as the one from two epochs before. The authors describe that with their algorithm, not only local optima but also the global optimum can be found. Benchmark tests are presented that show that other techniques like David–Fletcher Method [49] are outperformed. Note that the authors used up to 50000 epochs to find the best solution. This number is way too high for hyperparameter optimization because of the high cost of model training and evaluation.

The authors of [50] also introduce an alternative approach to making the random search adaptive. The search space in each iteration is adapted by changing the standard deviation. For the exact algorithm, refer to [50]. The authors present tests that illustrate that the adaptive approach accelerates convergence compared to the standard random search optimization.

In the context of hyperparameter optimization, where neural networks are initialized with random weights, evaluating the same configuration can yield stochastic results. To address this issue, various techniques can be employed, such as setting the seeds of random functions. However, in their work [51], the authors introduce three different adaptive random search approaches for tackling stochastic optimization problems. The first approach is called Adaptive Search with Resampling (ASR), which aims to reduce the noise in the objective function by resampling already evaluated points. The other two methods, Deterministic Shrinking Ball (DSB) and Stochastic Shrinking Ball (SSB) mitigate this noise by averaging the function values within a specific ball. The authors demonstrate that ASR is a highly promising algorithm, particularly in situations where normal random search struggles to identify optimal solutions.

Also, the authors of [52] introduce an algorithm for adaptive random search. Their approach is to make the step size adaptive with the so-called adaptive step size random search (ASSRS) as an approximation of the optimum step size random search (OSSRS). Also, the fixed step size random search (FSSRS) is described. A comparison with the fixed step size gradient method is provided which shows that the FSSRS performs

better than the gradient-based method for problems of dimension 5 or higher and a sufficiently small ratio of step size to distance to the minimum.

In our case for the hyperparameter optimization, we try to keep the number of function evaluations very small because of the time-consuming training and evaluation of the models. In [53], the adaptive partitioned random search (APRS) is presented which is at least 100 times more efficient compared to the normal random search in terms of the number of function evaluations. The authors present the simplicity and robustness of the algorithm. It partitions the search space into regions with function evaluations and estimates how "promising" each region is. Depending on this metric, more points are evaluated in more promising ones.

One other important point of hyperparameter optimization that we have to keep in mind is that we sometimes have variables that are not continuous e.g. the choice of the type of optimizer of a neural net or even integer parameter like the number of epochs to train. This also applies to other optimization problems. The authors of [54] mention the example of a heat exchanger with parameters like discrete tube diameters and lengths. This is why they apply the adaptive random search first introduced in [55] to four different problems with discrete and mixed parameters. They show that their approach is more efficient than previously applied algorithms and even new optima could be found.

# 3 Hyperparameter Optimization with Sparse Grids

## 3.1 Methodology

### 3.1.1 Adaptive Grid Search with Sparse Grids and Iterative Adaptive Random Search

In the scope of this thesis, we implemented and analyzed a new approach for the problem of the hyperparameter optimization of machine learning models. The basic idea is to use sparse grids and adaptively refine the grid points that seem to be promising. Similar to the grid search, first a few sample points are tried out and similar to the bayesian search, more promising configurations are exploited. With the adaptivity parameter of the sparse grid, a trade-off between exploration and exploitation is balanced by generating either a homogeneous or adaptive sparse grid. We analyzed the algorithm for evaluating the neural networks on its behavior to find a general configuration performing well for most datasets.

The reason for this work is that all well-analyzed hyperparameter optimization algorithms have advantages and shortcomings. We saw the promising possibility of using sparse grids for these kinds of optimization problems and compared the performance with grid-, random search, and bayesian optimization.

For this comparison, we used several kinds of artificial neural networks such as fully connected or convolutional ones. Additionally, multiple datasets of different sizes and with categorical or numerical features are being used. We tried really general tasks like regression or object detection and classification to validate the findings in a general way. In 3.1.2, a rough description of the implementation of the algorithms as well as the experiments, is presented.

### 3.1.2 Implementation

The implementation was done in Python (version 3.10.6) and two main classes are introduced. The first is the *Dataset* encapsulating the input and target data of the

dataset used for the machine learning model. One can either instantiate an object with a given X and Y vector for the data and target or by giving a task id. This identification number is used to instantiate both arrays with the dataset that can be fetched with the functionality given by OpenML [56]. Each task has a unique id, we first concentrate on Regression tasks with small neural networks.

The second class introduced is the Optimization class. The abstract super class has 5 concrete implementations, one for each of grid-, random search, bayesian optimization, the sparse grid search and iterative adaptive random search. In all cases, a dataset object has to be given. Additionally, the machine learning model as a function depending on the hyperparameters is required. For the sparse grid search, the functionality of the SG++ [57] is used. The function has to inherit the class *ScalarFunction* and the concrete model evaluation is done in the member function *eval(x)*. Also, the hyperparameter space has to be defined. This is a dictionary with information about the names and the types of parameters. Possible types are list for categorical ones, interval (int) for continuous (or integer) parameters, and log interval for ones that should be sampled logarithmically. Additionally, the lower and upper bound for each of them has to be provided. The next parameters are the budget defining the upper bound for function evaluations and the verbosity for outputs.

For the sparse grid optimization, the hyperparameters have to be scaled to the standard interval  $[0, 1]$  and back to the original one for interpretation. Therefore, the functions *to\_standard*, *from\_standard*, *to\_standard\_log* and *from\_standard\_log* are introduced for the linear and logarithmic scaling.

```

1 def to_standard(lower, upper, value):
2     return (value - lower) / (upper - lower)
3
4 def from_standard(lower, upper, value):
5     return value * (upper - lower) + lower
6
7 def to_standard_log(lower, upper, value):
8     a = math.log10(upper / lower)
9     return math.log10(value / lower) / a
10
11 def from_standard_log(lower, upper, value):
12     a = math.log10(upper / lower)
13     return lower * 10 ** (a * value)

```

The parameters lower and upper are the bounds of the original hyperparameter interval. The value argument is then scaled to or from the standard domain  $[0, 1]$ ,

respectively.

Due to many stochastic influences in the scope of neural networks, reproducibility is not always given by default. Examples are the network's weight initialization, random dataset splitting, and shuffling during the training phase. To overcome this problem, the seeds of random functions can be set to be deterministic and reproducible.

### 3.2 Sparse Grid Optimization of Functions

Before optimizing the configurations of machine learning models, simple functions are used. This has the advantage that the optimal point is already known in advance and a function call is much faster than evaluating a neural network. Therefore, three different test functions are given with the following properties [57]:

Table 3.1: Three test functions and their properties.

Function	Domain	$x_{opt}$	$f(x_{opt})$
Eggholder	$[-512, 512]^2$	(512, 404.2319)	-959.6407
Rosenbrock	$[-5, 10]^2$	(1, 1)	0
Rastrigin	$[-2, 8]^d$	$\vec{0}$	0

The Eggholder function is defined with [58]

$$f(x_0, x_1) = -x_0 * \sin(\sqrt{|x_0 - (x_1 + 47)|}) - (x_1 + 47)\sin(\sqrt{|x_1 + 47 + \frac{x_0}{2}|}). \quad (3.1)$$

The second function (Rosenbrock) [59] is calculated with

$$f(x_0, x_1) = (1 - x_0)^2 + 100(x_1 - x_0^2)^2. \quad (3.2)$$

and the third one (Rastrigin) [59] is defined with

$$f(\vec{x}) = 10d + \sum_{i=1}^d (x_i^2 - 10\cos(2\pi x_i)) \quad (3.3)$$

where  $d$  is the dimensionality of the input vector  $\vec{x}$ . Figure 3.1 shows the functions in two dimensions.

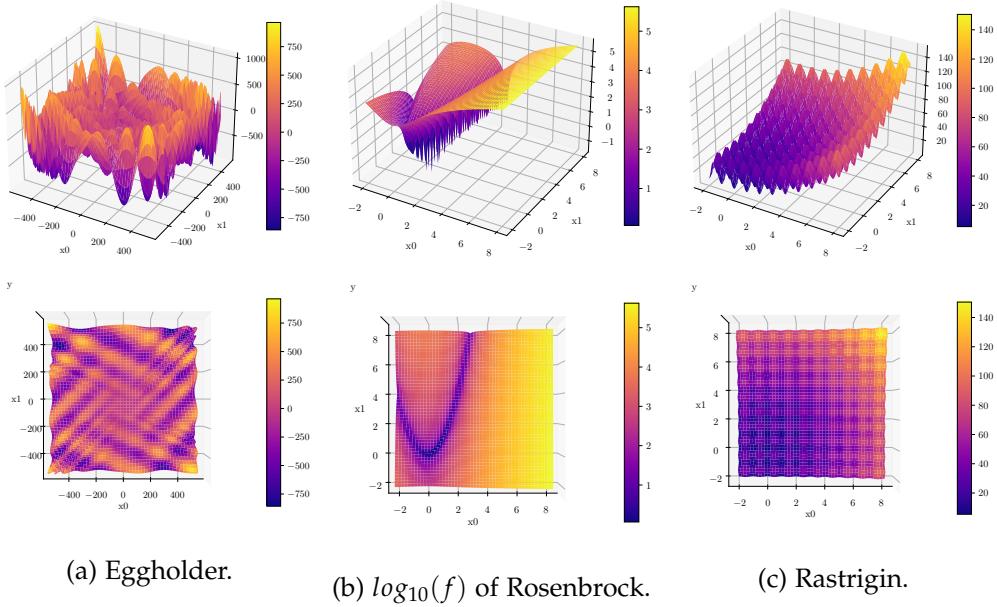


Figure 3.1: Test functions used for evaluating the sparse grid optimization. Each one is plotted with 200 samples in each dimension. Note that the function values of the Rosenbrock function are transformed with  $\log_{10}(f(x))$  for better visualization.

The Eggholder function (see Figure 3.1a) is oscillatory and the optimal point  $x_{opt}$  lays at the border of the domain. Additionally, it is multi-modal. The second one (see Figure 3.1b) is plotted with the function values additionally transformed with  $\log_{10}(x)$  for better visualization. The last one is also multi-modal (see Figure 3.1c).

As a first step, the sparse grid generation which is done with the Ritter Novak refinement criterion [32] is evaluated. In each iteration, the grid point  $x_{l,i}$  that minimizes the product

$$(r_{l,i} + 1)^{1-\gamma} \cdot (||l||_1 + d_{l,i} + 1)^\gamma \quad (3.4)$$

is refined. In this case,  $r_{l,i} = |\{(l', i') \in K | f(x_{l', i'}) \leq f(x_{l,i})\}| \in \{1, \dots, |K|\}$  is the rank of the grid point with  $K$  being the current set of level-index pairs of the grid. This rank denotes the place of the function value in the ascending order of all values of the current grid. On the other hand, the degree of the point  $d_{l,i} \in \mathbb{N}_0$  is the number of previous refinements at this point. One important choice has to be made for the adaptivity parameter  $\gamma$ . This value has to be between 0 and 1 and the smaller this value is, the more adaptive is the sparse grid. With this value, a trade-off between exploration and exploitation can be balanced. The optimal value generally depends on the function

that has to be optimized.

### 3.2.1 Sparse Grid Generation with different Adaptivities

In the following, the behavior of the sparse grid generation is analyzed with the help of the three test functions. In each case, 3 different values for the adaptivity parameter  $\gamma \in \{0.0, 0.5, 1.0\}$  are used and the resulting sparse grid with the triangulated function values is plotted. The first test case is the Eggholder function and the result is depicted in Figure 3.2.

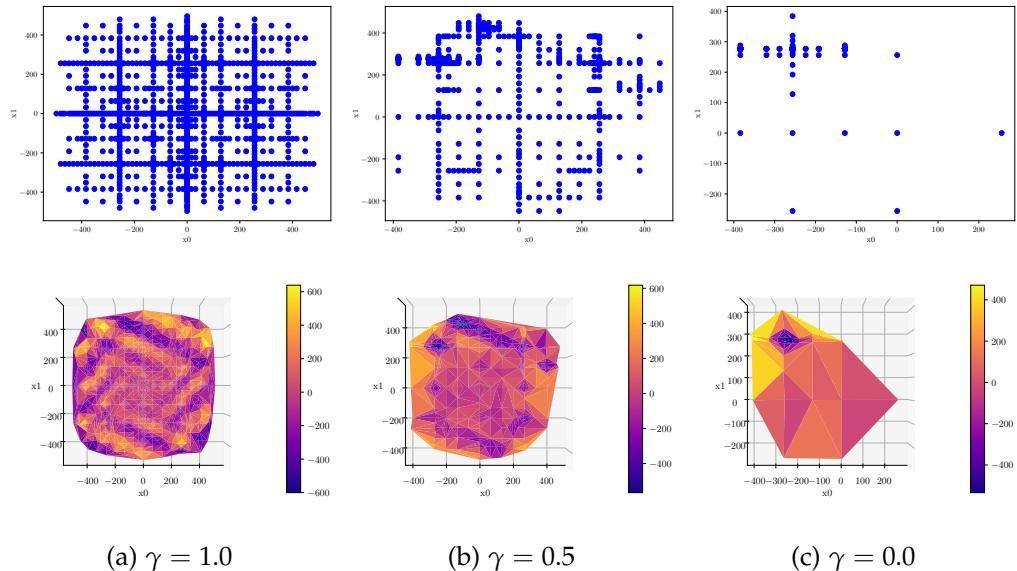


Figure 3.2: Sparse grid generation depending on the adaptivity parameter  $\gamma$  for the Eggholder function. In all cases, the same number of grid points is used. Here, in each of the 249 iterations, 4 new grid points are added resulting in a overall number of 997 function evaluations.

In the first case with  $\gamma = 1.0$ , the grid is homogeneous and not adaptive at all. The grid points are distributed over the whole domain and the interpolated function looks very similar to the real plot from Figure 3.1a.

The other extreme case is depicted in Figure 3.2c. There, the grid is maximally adaptive and really concentrated at the top left corner around  $(-260, 280)$ . As it is known from Table 3.1, this is not where the optimal point is located. This behavior can be explained by the high exploitation throughout the iterations. With such a low

adaptivity parameter, the grid points with low function values in the first iterations are mostly refined in the other iteration steps.

The middle case with  $\gamma = 0.5$  depicts a case where exploitation and exploration are more balanced. The grid points are more distributed than in the case of  $\gamma = 0.0$  but there are also some regions where they are more refined, e.g. in the top left corner of the domain.

Note that in all three cases, the exact same number of grid points are evaluated. In this case, it is very hard for the sparse grid to find the real optimum because it is located at the border of the domain and it does not use grid points at the border. Also, the oscillatory nature of the function makes it hard to not concentrate on a local optimum which can happen in case of too high adaptivity.

The next function used is the Rosenbrock function (see Figure 3.1b). Again, three different values for the adaptivity parameter are used. The results are depicted in Figure 3.3.

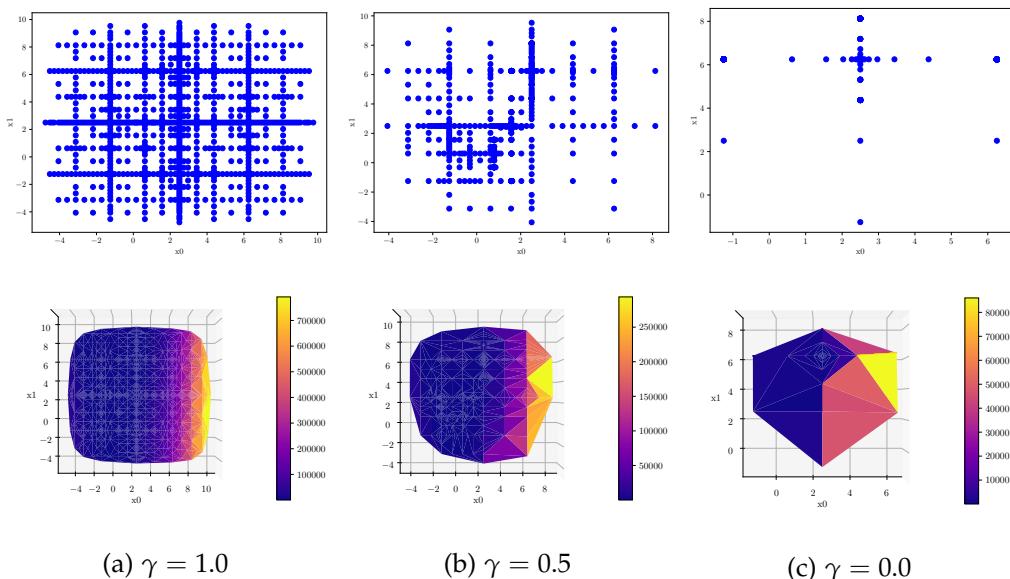


Figure 3.3: Sparse grid generation depending on the adaptivity parameter  $\gamma$  for the Rosenbrock function. In all cases, the same number of grid points is used. Here, in each of the 249 iterations, 4 new grid points are added resulting in a overall number of 997 function evaluations.

The first fact that can be observed is that the sparse grid in the first case (Figure 3.3a) is exactly the same as the one for the Eggholder function (Figure 3.2a). This is due to

the fact that this value of  $\gamma$  leads to a homogeneous sparse grid which is not dependent on the function used but rather the number of iterations for the grid generation. In this case, the interpolated function looks similar to the real function (Figure 3.1b).

Now with decreasing value of  $\gamma$ , the grid gets more and more inhomogeneous, while concentrating to refine smaller function values. The extreme case can be seen in Figure 3.3c, where the most grid points are next to the point  $(2.5, 6.25)$ . After refining the first point in the middle, the one on top of it is getting refined all the time for the case  $\gamma = 0.0$ . Again, the sparse grid with  $\gamma = 0.5$  is more balanced with more grid points on the left where the real optimum is located.

The last function used is the Rastrigin function (see Figure 3.1c for the plot of the function and 3.4 for the resulting sparse grids).

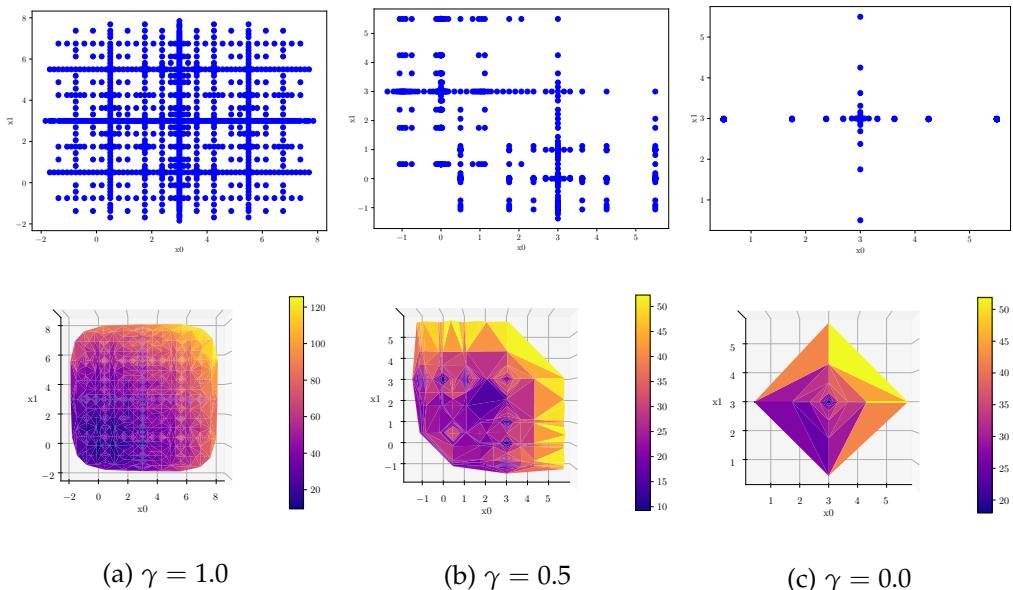


Figure 3.4: Sparse grid generation depending on the adaptivity parameter  $\gamma$  for the Rastrigin function. In all cases, the same number of grid points is used. Here, in each of the 249 iterations, 4 new grid points are added resulting in a overall number of 997 function evaluations.

As in the previous two cases, the sparse grid is exactly the same for  $\gamma = 1.0$ . We can also see the same behavior for a decreasing adaptivity parameter. For  $\gamma = 0.0$ , only the grid point in the center is refined in all steps. In the middle case, the grid looks more balanced with a tendency to the bottom left corner.

In conclusion, these experiments show that the value for the adaptivity parameter strongly influences the grid generation and the resulting optimal value found by the algorithm. In general, this trade-off between exploitation with trying to find a solution fast and exploration with reconstructing the function in the whole domain can be balanced with this adaptivity parameter.

In the following, the goal is to further analyze this adaptivity parameter. For each of the three test functions, experiments with  $\gamma \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$  are made. The error of the optimum found by the grid is calculated depending on the number of grid points used. It is evaluated with

$$Error = f(x_{opt}^*) - f(x_{opt}) \quad (3.5)$$

where  $x_{opt}^*$  is the optimal point found by the sparse grid and  $x_{opt}$  is the real optimum. Figure 3.1 shows the resulting Errors depending on the number of grid points used. The top left plot is for the Eggholder function, the one on the top right belongs to the Rosenbrock function and the bottom one corresponds to the Rastrigin function.

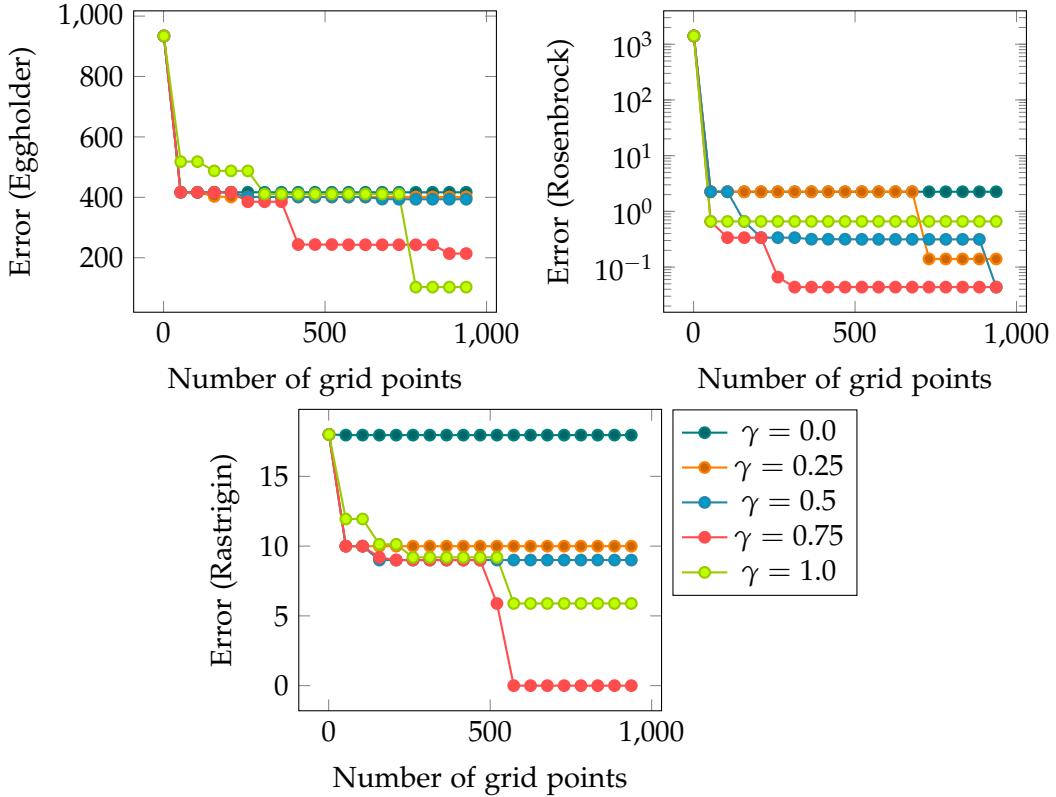


Figure 3.5: Influence of the adaptivity parameter on the error (difference to the actual optimal value) of the optimum found by the sparse grid.

For all three test functions, a value  $0.75 \leq \gamma \leq 1.0$  leads to the smallest error with higher number of grid points. This means that a fast exploitation is not good for finding the global optimum. For the Rastrigin function, the most adaptive sparse grid does not even lead to a smaller error with up to 1000 grid points. This is the same example as in Figure 3.4c, where only the center point is refined. Note that by now, no optimization algorithm is applied on top of the sparse grid. For further experiments, the adaptivity parameter will be set to  $\gamma = 0.85$ .

### 3.2.2 Local and Global Optimization

The next improvement is to add optimizers after the grid generation phase. There are two different types of algorithms. The first one is local optimization, for example, based on the gradient like the gradient descent. The second one is global optimization. An

example therefore is to use a multi-start approach by trying out multiple different starting points for the algorithm. These various initial values can be uniformly distributed in the domain. The following experiments show how the error (difference between optimum found by the algorithm and actual optimal function value) behaves with a higher number of grid points of the underlying sparse grid. As a local optimization algorithm, gradient descent is used. The starting point for this algorithm is set to the point of the sparse grid where the smallest function value was found. For the global optimization, a multi-start approach with 20 equally distributed starting points is used. The concrete algorithm is the nelder mead optimization as described in 2.3.4. The number of function evaluations used in this method, as well as the number of steps for gradient descent is set to 1000. One important parameter for the optimization is the degree of the B-splines used on the sparse grid. The resulting errors depending on the number of grid points for the degrees 2, 3, and 5 can be seen in Figure 3.6. The Rosenbrock function is used in all cases.

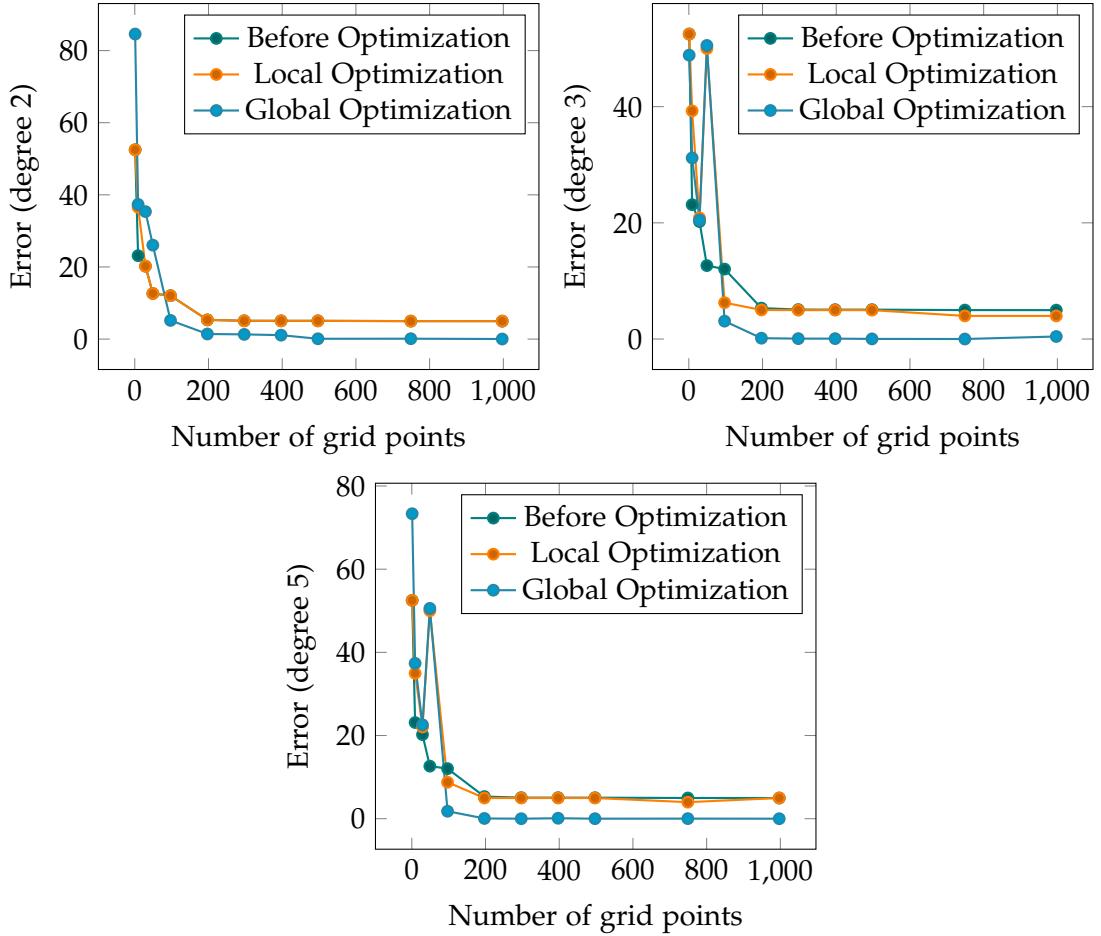


Figure 3.6: Optimization error for different algorithms depending on the number of grid points in two dimensions. The plots show the results with degree 2 (top left), degree 3 (top right) and degree 5 (bottom). The Rastrigin function was used.

In all three cases, the errors of the local and global optimizers first increase with increasing number of grid points until about 100 to 200 function evaluations are done. After that, both local and global optimization algorithms are at least as good as the result found by the sparse grid. With increasing number of grid points, the global optimization finds the best solution in all cases with degrees 2, 3, and 5.

The visualization in Figure 3.7 indicates why the global optimizer achieves the best results for a high number of sparse grid points.

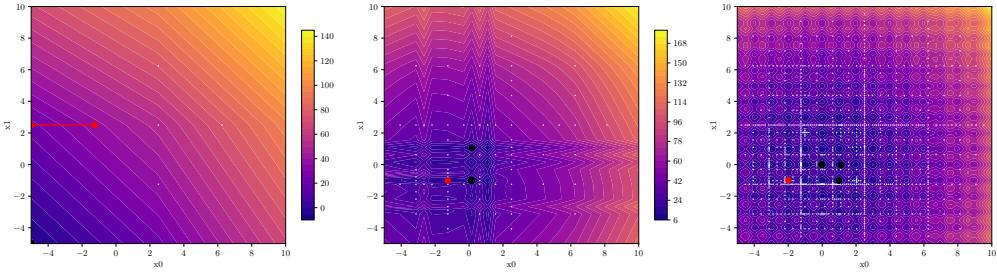


Figure 3.7: Resulting optimal points from local (red points) and global (black points) optimization algorithm. In the background, the contour of the interpolated function is depicted with the corresponding sparse grid points in white. The left one has 5 grid points, the one in the center 77, and the right one 997. The degree of the B-splines on the sparse grid is 2.

In the left example of Figure 3.7 with 5 sparse grid points, the result of the local optimizer is at the left border of the domain as indicated by the red line which connects the different steps of the algorithm. All resulting optimal points found by the multi-start approach are located in the lower left corner. Note that the interpolated function is not very similar to the original one (see Figure 3.1c) because of the low number of grid points.

With an increasing number of these basis points, both algorithms find a better solution to the problem. In the center plot of Figure 3.7, 77 grid points were used and the interpolated contour looks more oscillatory and more similar to the original function. The local and global algorithms find different optimal points in this case.

In the right example of Figure 3.7, 997 grid points were used and the contour already looks very similar to the function plot. There are many candidates of the global optimizer and so some of them are near to the actual optimal point which is located at  $(0, 0)$ .

The exact solutions of each algorithm can be seen in Table 3.2.

The big advantage of the global optimizer is that multiple starting points are used so that the chance of reaching the global minimum is very high. Especially for a high number of grid points used, this is the case. While the local optimizer only finds a local minimum (see Figure 3.7, red dot at  $(-1.99, -0.994)$ ), the global optimizer finds multiple candidates at more than one local minimum. One of them is very near to the global optimal point in this case (see Figure 3.7, black dot at  $(0.002, 0.013)$ ).

One approach is to combine all three resulting points with their corresponding function value. The overall optimal point is then just set to the one with the smallest function value.

Table 3.2: Overview over the exact solutions found by the local and global optimizer for the same problem as in Figure 3.7. The actual optimum is at (0, 0), with function value 0 (see Table 3.1).

Number grid points	Optimizer	Coordinates	Interpolated value	Actual value
5	Local	(-5,2.5)	23.125	51.25
5	Global	(-4.998,-4.971)	-6.191	49.862
77	Local	(-1.25,-1.016)	12.642	12.642
77	Global	(0.157,-1.006)	5.756	5.514
997	Local	(-1.990,-0.994)	4.975	4.975
997	Global	<b>(-0.014,-0.002)</b>	0.151	<b>0.042</b>

The inputs of the previous experiments were all just two-dimensional. The following ones analyze the impact of the dimension on the performance of the algorithm. Therefore, the Rastrigin function is optimized, again with increasing number of grid points. The degree of the B-splines used on the sparse grid is set to 5 and for the adaptivity parameter, we set  $\gamma = 0.85$ . The results are depicted in Figure 3.8.

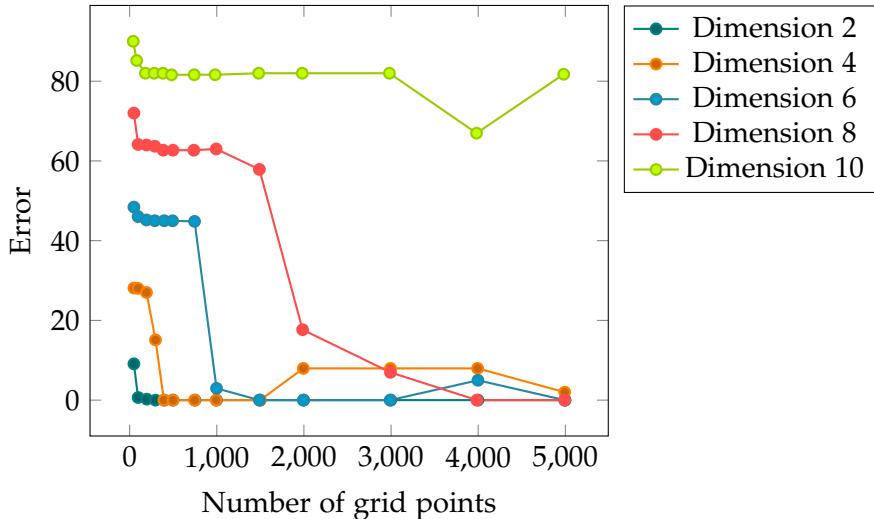


Figure 3.8: Error of the optimization depending on the number of grid points used for different dimensions of the Rastrigini function.

The results show that with increasing dimensionality of the problem, more grid points are needed in order to reduce the error. The metric was again defined as the

optimum found by the algorithm subtracted by the actual optimal value.

In conclusion, the adaptivity parameter, the degree of the B-splines on the sparse grid, and the dimensionality of the problem that is optimized, strongly influence the behavior of the sparse grid optimization. For the adaptivity parameter, a value with about  $\gamma = 0.85$  is in general a good choice for a function that is not known in advance. With a higher degree, we can find the optimum a bit faster and for higher dimensionality, more refinement iterations of the sparse grid are needed.

### 3.3 Hyperparameter Optimization with Sparse Grids

Now we replace the function, where we know the analytical optimum with the evaluation of a machine learning model. The biggest change is the much higher duration of one function call and the complexity of finding the analytical solution which is impossible due to the high number of network weights in neural networks.

#### 3.3.1 Optimum Approximation with Sparse Grids

The following plots show an example of a small neural network being evaluated on the diamonds dataset which is available on OpenML [56]. We used a model consisting of two fully connected layers, each made up of 30 neurons. In one batch, 100 data samples are processed. The two-dimensional plot (see Figure 3.9) depicts the network evaluation depending on the number of epochs used and the value for the learning rate of the Adam optimizer of the model. The loss of the network is calculated with the mean squared error. As pre-processing of the data, the numeric features of the input data are scaled with a standard scaler and the categorical features are one hot encoded. The target values are also scaled separately. For the evaluation metric, we chose the average result of 2-fold cross validation with the mean absolute percentage error (MAPE). This metric is defined as

$$\text{MAPE} = \frac{1}{n} \sum_{i=1}^n \left| \frac{y^i - y_{\text{pred}}^i}{y^i} \right| \quad (3.6)$$

where  $y^i$  is the target value and  $y_{\text{pred}}^i$  is the predicted value of data point  $x^i$ .

The interval of the epochs is  $[1, 40]$  and the learning rate is sampled equidistant and logarithmic between  $[10^{-10}, 10^{-1}]$ . The resulting plot with the result depending on the epochs and the  $\log_{10}(\text{learning rate})$  can be seen in Figure 3.9.

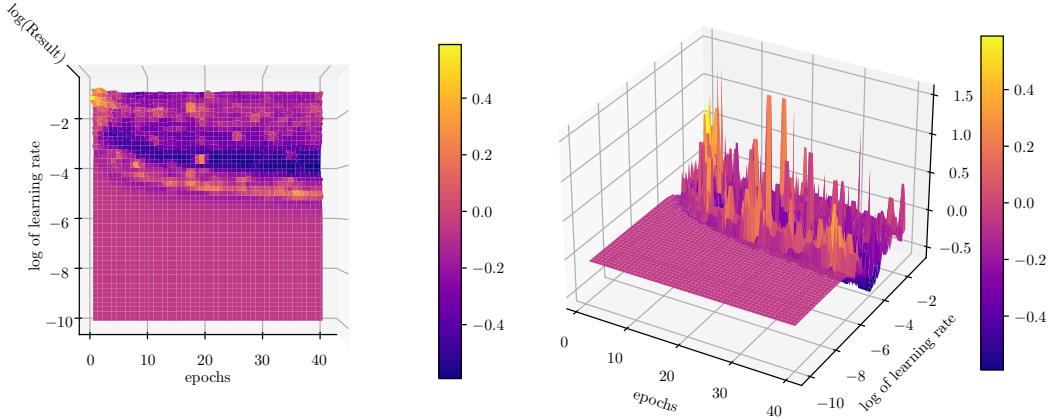
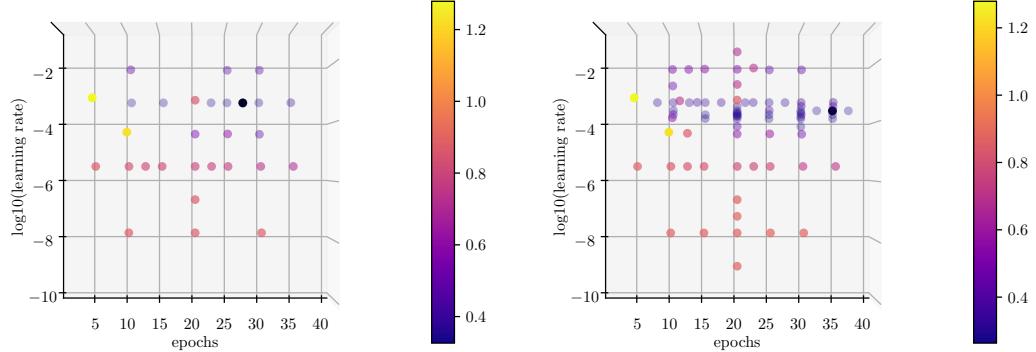


Figure 3.9: 2-layer (with 30 neurons each) fully connected network evaluation on the diamonds dataset depending on the number of epochs and learning rate of the Adam optimizer. The result is plotted with  $\log_{10}(\text{Result})$  for better visualization.

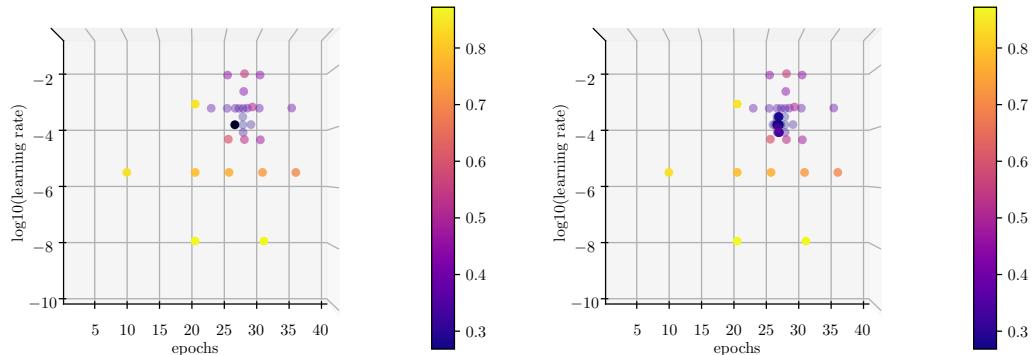
In the plot in Figure 3.9, for each epoch and learning rate, 100 different, equidistant values are sampled from the corresponding interval (linear sampling for epochs and log sampling for learning rate). All different combinations are evaluated leading to  $100^2 = 10000$  function evaluations. This took about 45 hours on a normal machine. Regarding the goal of hyperparameter optimization, this technique of trying all different combinations of distinct values is comparable to the grid search approach which is not very efficient. Although, the rough behavior of the machine learning model can be analyzed. The function is nearly constant for the learning rate between  $10^{-6}$  and  $10^{-10}$ . This is caused by the Adam optimizer adjusting the weights of the network too slowly in order to minimize the error. In the other half of the domain (learning rate between  $10^{-1}$  and  $10^{-5}$ ), a blue region where the smallest errors are achieved can be observed. This is where the combination between epochs and the learning rate is very good for achieving good results. With this plot, a general observation can be made. However, it takes far too much time to analyze each problem with a plot like this. Additionally to the high effort, only the region of the minimum can be determined, not the optimal point itself.

With this knowledge about the underlying function that has to be minimized, the behavior of the sparse grid can be analyzed. Figure 3.10 depicts the generated grid colored corresponding to the function value for different adaptivity values and the number of grid points. Note that the scale of the learning rate is the one interpreted by

the sparse grid. The actual scale is logarithmic between  $10^{-10}$  for 0.0 and  $10^{-1}$  for 1.0.



(a) Sparse grid generated with adaptivity parameter  $\gamma = 0.85$  and number of grid points of 29 (left) and 77 (right). Most points are in the upper half for higher values of the learning rate.



(b) Sparse grid generated with adaptivity parameter  $\gamma = 0.5$  and number of grid points of 29 (left) and 77 (right). Most points are in the upper right half for higher values of the learning rate and epochs between 25 and 30.

Figure 3.10: Analysis of sparse grid with machine learning evaluation for different adaptivity parameters (0.85 3.10a and 0.5 3.10b), each with 29 and 77 grid points.

In the first case with  $\gamma = 0.85$ , more grid points are generated in the upper half of the domain. A similar behavior can be observed for a higher number of grid points (3.10a, right). Especially in regions where the learning rate is about 0.0002 to 0.0003

(scale marked at 0.7), many grid points are refined. From the analysis in Figure 3.9, there might be the optimal value.

For a smaller adaptivity value of  $\gamma = 0.5$ , the sparse grid is much more adaptive as depicted in Figure 3.10b. For both the number of grid points 50 and 100, the most refined part of the sparse grid is at the same region. One important thing to observe here is that with higher adaptivity, a lower number of grid points is necessary to find one candidate of the optimum value. On the other hand, with lower adaptivity, more grid points are needed to find a good minimal value.

However, as shown in Table 3.3, the best value is found with the configuration  $\gamma = 0.85$  and number of grid points of 100. This means that lower adaptivity leads to better results.

Table 3.3: Best hyperparameter configuration found by the sparse grid with different adaptivity parameters and number of grid points. The best result is found with  $\gamma = 0.85$  and 77 grid points.

$\gamma$	N	Epochs	Learning rate	Result
0.85	29	27	0.00056	0.32599
0.85	77	35	0.00029	<b>0.26339</b>
0.5	29	26	0.00015	0.26837
0.5	77	26	0.00015	0.26837

The results shown in this table prove that too high adaptivity is not good for finding the smallest function value. However, when using higher adaptivity, the number of grid points has no big impact on finding the best solution in this case. This is because of the function shown in Figure 3.9. In general, a higher adaptivity is good for finding a configuration faster. This can be useful in cases of hyperparameter optimization where not much time is available and a relatively good solution is enough.

### 3.3.2 Optimization on Sparse Grids

By adding optimization methods like gradient descent or evolutionary algorithms, an even better approximation of the optimum might be found if enough grid points are available. If this number is too small then the interpolant is not precise enough and the solution of the optimizer is not the real optimum. One example is depicted in Figure 3.11. The steps of the normal gradient descent optimization algorithm are depicted and connected with arrows. In the background, the interpolated function is shown. The problem is the same as in Figure 3.9. The degree of the B-splines on the grid is 2 and the sparse grid points are shown in white.

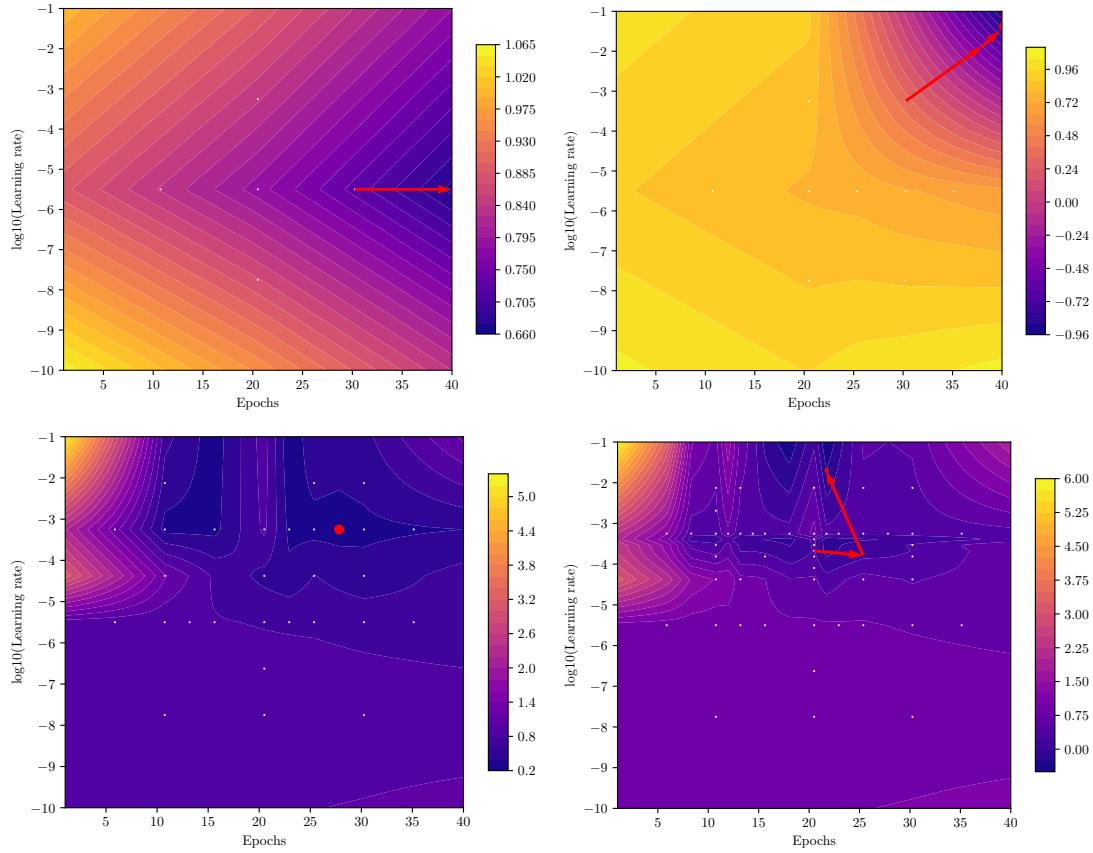


Figure 3.11: Optimization steps of gradient descent algorithm for 5 (top left), 9 (top right), 29 (bottom left) and 49 (bottom right) grid points. In the background of each plot, the contour of the interpolated function is shown. The function evaluated is the same as depicted in Figure 3.9.

In Figure 3.11, the solution of the gradient descent optimization algorithm is analyzed with 5, 9, 29 and 49 grid points on the top left, top right, bottom left and bottom right, respectively. The minimal interpolated function values and the actual function evaluations can be seen in Table 3.4.

Table 3.4: Exact values for the optima found by the sparse grid points and the gradient descent algorithm. The values for  $x_{min}^{grid}$  are the configurations evaluated by the sparse grid during the generation and the coordinates in column  $x_{min}^{opt}$  are found by the optimizer. In bold, the best function values of the optimum found by the sparse grid and gradient descent algorithm, respectively.

N	$x_{min}^{grid}$	$f(x_{min}^{grid})$	$x_{min}^{opt}$	$f(x_{min}^{opt})$	$f_{interpolated}(x_{min}^{opt})$
5	(30.25, $3.162 * 10^{-6}$ )	0.729	(40, $3.162 * 10^{-6}$ )	0.668	0.670
9	(30.25, $5.623 * 10^{-4}$ )	0.353	(40, $1 * 10^{-1}$ )	1.257	-0.950
29	(27.81, $5.623 * 10^{-4}$ )	0.326	(27.81, $5.623 * 10^{-4}$ )	<b>0.326</b>	0.326
49	(20.5, $2.129 * 10^{-4}$ )	<b>0.267</b>	(21.72, $2.094 * 10^{-2}$ )	0.524	-0.247

In the first case, the optimizer starts at the right grid points and makes one step towards the right boundary of the domain. In the top right case with 9 grid points, the solution of the optimizer is at the right top corner of the domain. With 29 basis points in the left lower case, the solution of the optimizer is the same as the grid point with the smallest evaluated function value and in the right lower plot, the optimizer first takes a step to the right and then a second step up to a higher learning rate.

As illustrated in Table 3.4, the optimizer does not always find an actual better solution. While the interpolated function value actually decreases, the evaluation of the network with the given configuration results in a higher value which is worse. This behavior is the same as in Figure 3.6 where both optimizers lead to a higher error than the smallest value found by the sparse grid. This is due to the small number of function evaluations made so far and the interpolant of the function being too inaccurate. Also the visual comparison of the interpolant plots in Figure 3.11 and Figure 3.9 shows that there are still big differences. Nevertheless in the scope of this thesis, further increasing the number of grid points is infeasible because of the high costs of one single function evaluation which is the training and evaluation of a machine learning model.

Note that the additional optimization is very cheap compared to generating the adaptive sparse grid because the algorithm uses the interpolated function and does not have to train and evaluate whole machine learning models. In Figure 3.12, the different algorithms for optimization based on the sparse grid are compared. The same neural network with the same dataset is used for the comparison.

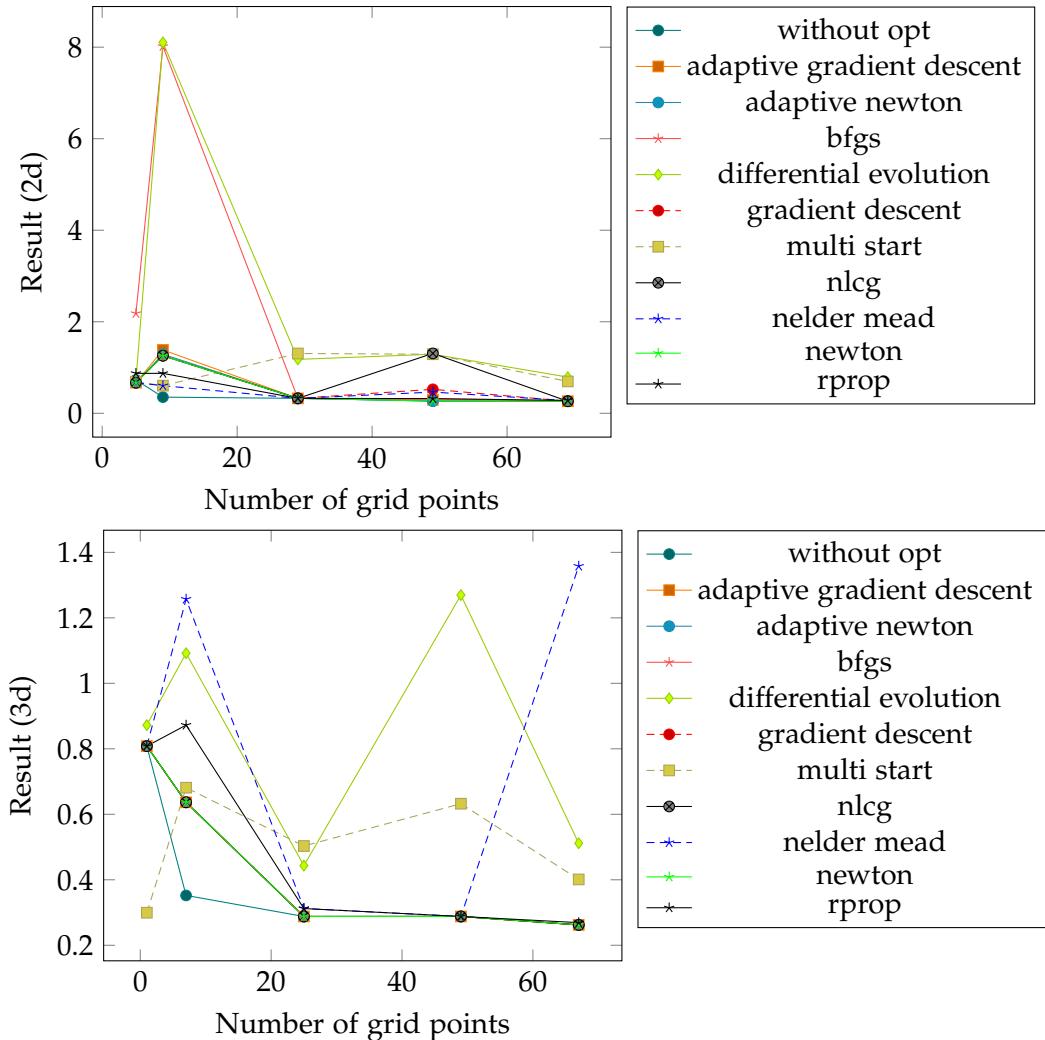


Figure 3.12: Comparison of optimization algorithms on the sparse grid with increasing number of grid points in two and three dimensions. Epochs, learning rate, and the batch size of a two-layer neural network on the diamonds dataset for regression are optimized. Result is the mean absolute percentage error.

For both cases, the solution found by the sparse grid without the optimization step decreases with increasing number of basis points. For the optimizers, this is not always the case. The resulting machine learning performance with the configuration found by the respective algorithm is even much worse than the one found by the sparse grid in some cases. In comparison to Figure 3.6, we are only using a small number of grid

points and get the same result in this situation. But further increasing the number of grid points is not feasible because of the high evaluation cost of the neural network.

One thing that has to be taken into consideration is that e.g. the epochs hyperparameter is not continuous. The values for the number of epochs are always integers converted with `int()`. This means that all values in the interval  $[x, x + 1]$  for any  $x \in \mathbb{N}$  are interpreted as the smaller integer  $x$ . This means that there are regions in the interval that have the exact same function value because they are evaluated with the same configuration. This fact might impact the behavior of a gradient-based optimizer. If the function values in an interval are exactly the same then the gradient is zero in this interval. This would make it impossible to find the real optimum if the optimizer gets stuck. In the experiment shown in Figure 3.13, an extreme example is tried out. We use two hyperparameters with three distinct values each. The first one is the batch size with values from 100, 400, 2000 and the second one is epochs taking values from 1, 5, 13. In two dimensions, nine different configurations are then possible. We used B-splines of degree 0 and a regular sparse grid with the smallest adaptivity possible. As the machine learning model being evaluated, we took a 2-layer network with 30 neurons each and a learning rate of  $10^{-5}$ . As dataset, the diamonds regression set is used.

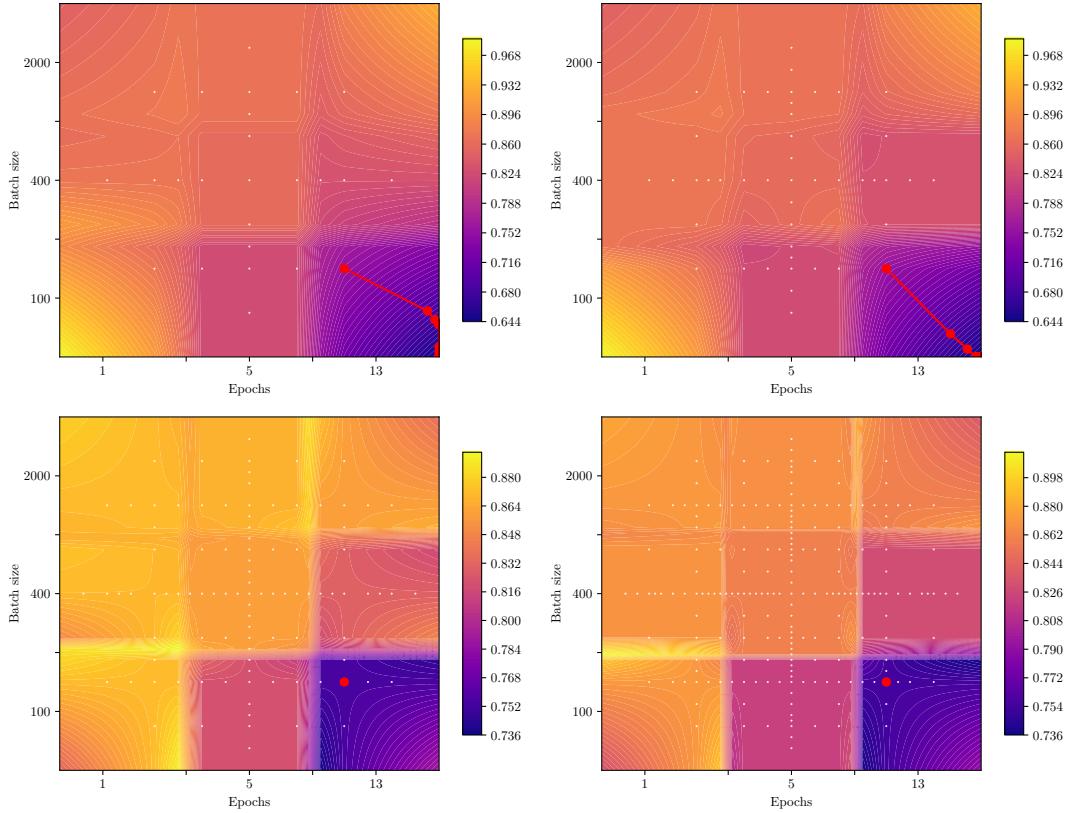


Figure 3.13: Sparse grid optimization with the interpolated function in the background, the white grid points, and the optimizer steps in red. Four different budgets are used (30: top left, 50: top right, 100: bottom left, 200: bottom right). Only three different configurations with values for batch size from 100, 400, 2000 and values for epochs from 1, 5, 13.

In Figure 3.13, four different budgets (30, 50, 100, 200) which are shown top left, top right, bottom left and bottom right, respectively, are depicted. In each case, the interpolated function is shown in the background with the grid points from the sparse grid. In red, the steps of the gradient descent optimizer are shown.

We can observe one tendency. The more grid points are used, the more clear it gets that there are 9 different configurations and corresponding function values. For the first two cases in the top, the gradient descent still makes steps towards the bottom right corner of the domain. The interpolated function also still looks smooth. In the other two cases with a budget of 100 and 200, the optimizer does not take any steps

because the grid points located next to the one with the smallest value also have the same smallest value.

Now in this case with only 9 distinct values, the sparse grid with its optimizer is finding the global optimum. But in other cases where there are more distinct configurations, this might lead to the optimizer not finding the global optimum.

We can conclude that the use of any optimizer is not very promising for finding an improved solution in our case. However, we will still use a local and global optimizer because it is very cheap compared to the sparse grid generation and in some cases, the solution is improved. The resulting configuration returned by the algorithm will just be the best of the three alternatives.

## 3.4 Comparison with Grid-, Random Search and Bayesian Optimization

Now with the sparse grid search being analyzed with defined functions and a small neural network, this algorithm will be compared to the already existing optimization methods presented in Chapter 2. We will always compare the algorithms depending on some given budget. This is the upper bound for the number of function evaluations the algorithm is allowed to make. For the grid search, this budget is always  $n^d$  where  $d$  is the number of hyperparameters or dimension of the problem. Therefore,  $n$  different values are taken for each hyperparameter. On the other hand, for the sparse grid search, the highest number of grid points is  $\text{budget} - 2$  because we have to evaluate the point of the local and global optimization for comparison.

### 3.4.1 Two-dimensional Experiment of Regression with Small Neural Network

The first experiment is using a two-layer neural network with 40 neurons in each layer. A batch size of 100 is used and as datasets, 4 different tasks from OpenML are taken. Table 3.5 gives an overview of the datasets.

Table 3.5: Overview over the datasets used for the first comparison of the optimization algorithms. They are all available on OpenML [60].

Dataset	Number of features		Number of instances
	numeric	categorical	
diamonds	7	3	53940
house_16H	17	0	22784
sensory	11	1	576
house_sales	16	2	21613
Brazilian_houses	9	3	10692

All the categorical features are transformed with one hot encoding and the numeric as well as the target values are scaled using the *StandardScaler* from the *scikit-learn* library [61]. Both transformers are trained on the training set and used for both training and test set. We used 2-fold cross-validation with the mean absolute percentage error as the metric. Before each run, the seeds for getting the random values are reset. This is necessary because the evaluation of the neural network with a specific configuration should always return the same value. For the B-splines on the sparse grid, we use a degree of 2 and the grid is generated with an adaptivity value of 0.85. Two hyperparameters, the number of epochs and the learning rate for the neural network's optimizer, are optimized. The first one is linear from 1 to 40 and the second hyperparameter is logarithmic from  $10^{-9}$  and  $10^{-1}$ . The resulting performance can be seen in Figure 3.14.

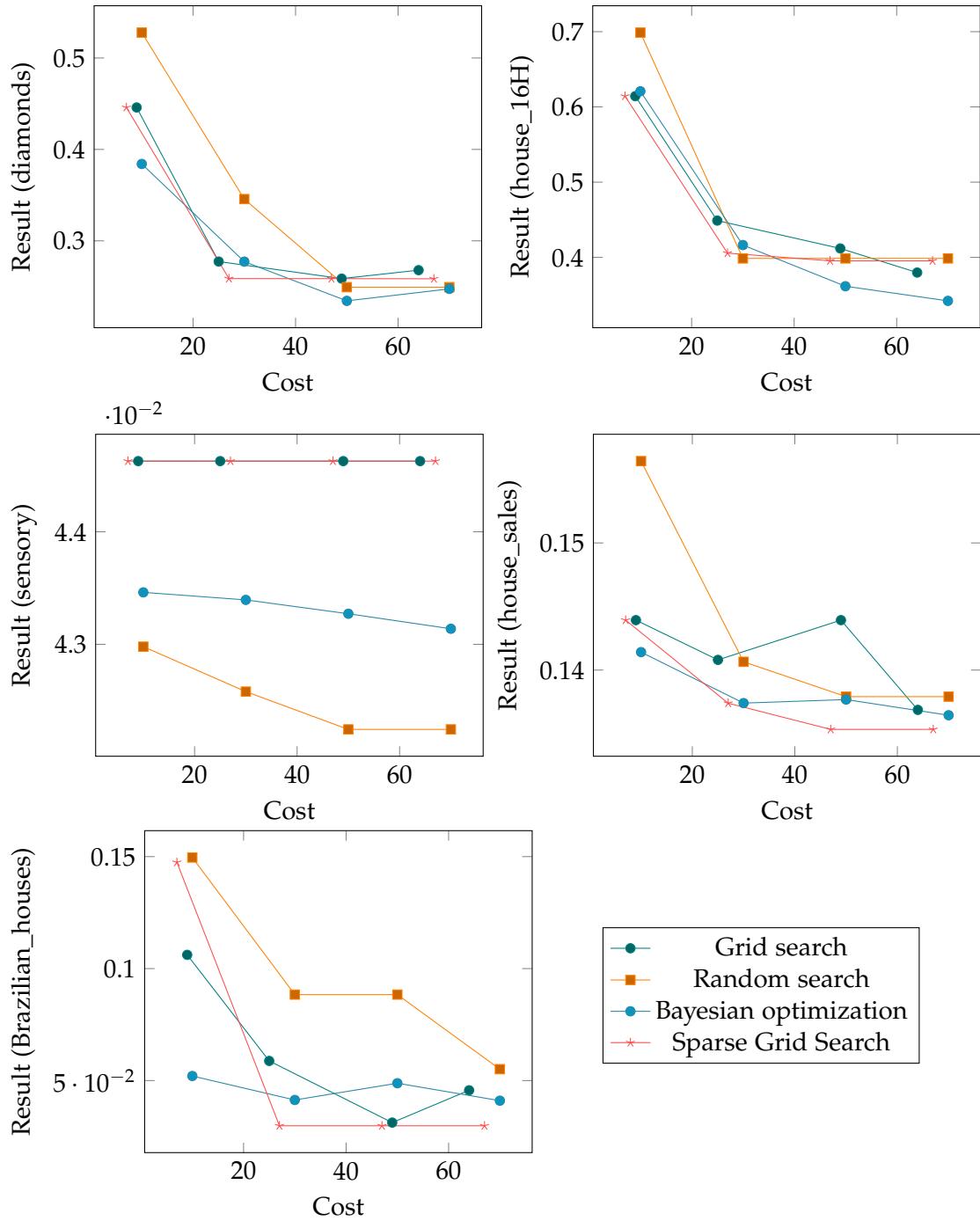
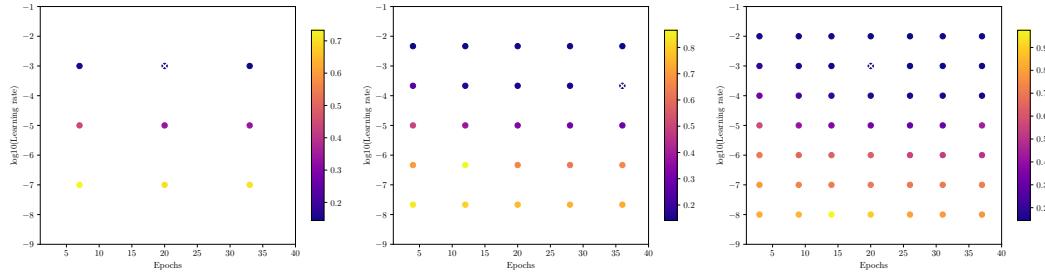


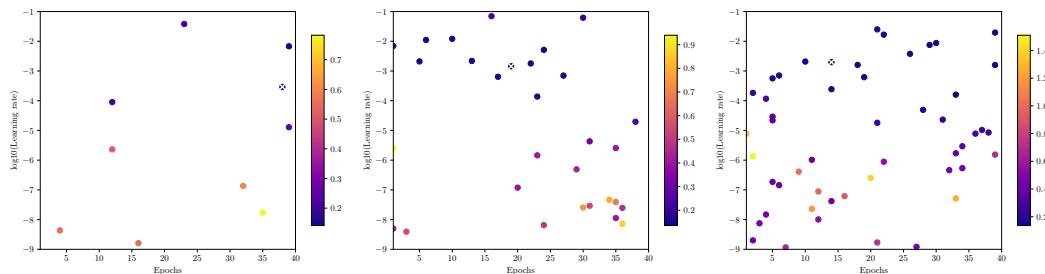
Figure 3.14: Comparison of grid-, random search, bayesian optimization and sparse grid search for the datasets shown in Table 3.5. Two hyperparameters, epochs and learning rate of the neural network's optimizer, were optimized.

It can be seen that in almost all cases, the result is decreasing with increasing cost. For the house\_sales and Brazilian\_houses datasets, the sparse grid optimization is performing best for a cost which is higher than 20. For the sensory dataset, all four optimization algorithms already get very good results with small costs. This is due to the small number of data entries. This is not representative of general behavior. When comparing the normal grid search and the sparse grid optimization, the second one performs better in most cases. Figure 3.15 shows the behavior for the house\_sales dataset and an upper bound of 10, 30, and 50 for the cost.

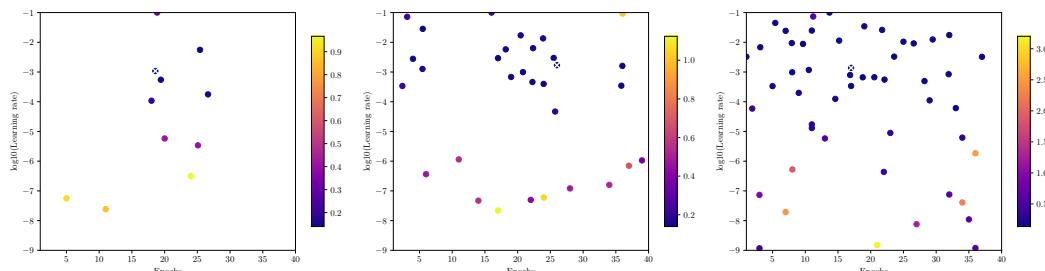
### 3 Hyperparameter Optimization with Sparse Grids



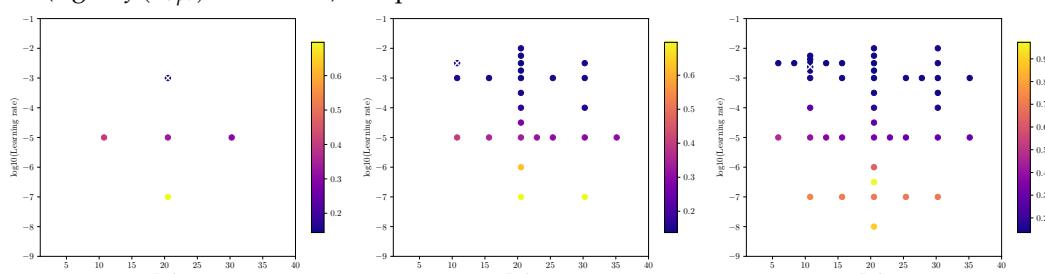
(a) Grid search with 9 (left,  $f(x_{opt}) = 0.14393$ ), 25 (center,  $f(x_{opt}) = 0.14080$ ), and 49 (right,  $f(x_{opt}) = 0.14393$ ) grid points.



(b) Random search with 10 (left,  $f(x_{opt}) = 0.14141$ ), 30 (center,  $f(x_{opt}) = 0.13526$ ), and 50 (right,  $f(x_{opt}) = 0.13619$ ) samples.



(c) Bayesian Optimization with 10 (left,  $f(x_{opt}) = 0.13926$ ), 30 (center,  $f(x_{opt}) = 0.13877$ ), and 50 (right,  $f(x_{opt}) = 0.13542$ ) samples.



(d) Sparse grid optimization with 5 (left,  $f(x_{opt}) = 0.14393$ ), 25 (center,  $f(x_{opt}) = 0.13739$ ), and 45 (right,  $f(x_{opt}) = 0.13533$ ) grid points.

Figure 3.15: Comparison of the grids generated for the house\_sales dataset. The best configuration found  $x_{opt}$  is marked with a white cross.

The big difference between grid search and sparse grid optimization is the adaptivity and the location of grid points. In Figure 3.15, this can be seen very clearly. While the grid generated for the grid search (see Figure 3.15a) is very homogeneous and spread over the whole domain, the grid points are more gathered for the sparse grid optimization (see Figure 3.15d). For these upper bounds for the cost and for this dataset, the sparse grid optimization finds configurations with a bit better performance.

### 3.4.2 Three- and Five-dimensional Experiments of Regression with Small Neural Network

While the two-dimensional case is good for the visualization of the approaches, one is often interested in the optimization of more hyperparameters. In the following, we will focus again on regression problems, but this time we will include the batch size, the number of neural layers, and the number of neurons per layer in the hyperparameter space.

For the three-dimensional case, we first use the same network with fixed architecture again, consisting of 2 layers with 40 neurons each. The hyperparameter space consists of the epochs, the batch size and the learning rate of the neural network. The first two are linear integers from 1 to 40 for the epochs and the batch size has 100 and 2050 as bounds. The learning rate is again logarithmic from  $10^{-9}$  to  $10^{-1}$ . We used the three datasets house\_16H, house\_sales, and Brazilian\_houses (see Table 3.5). The resulting performance can be seen in Figure 3.16.

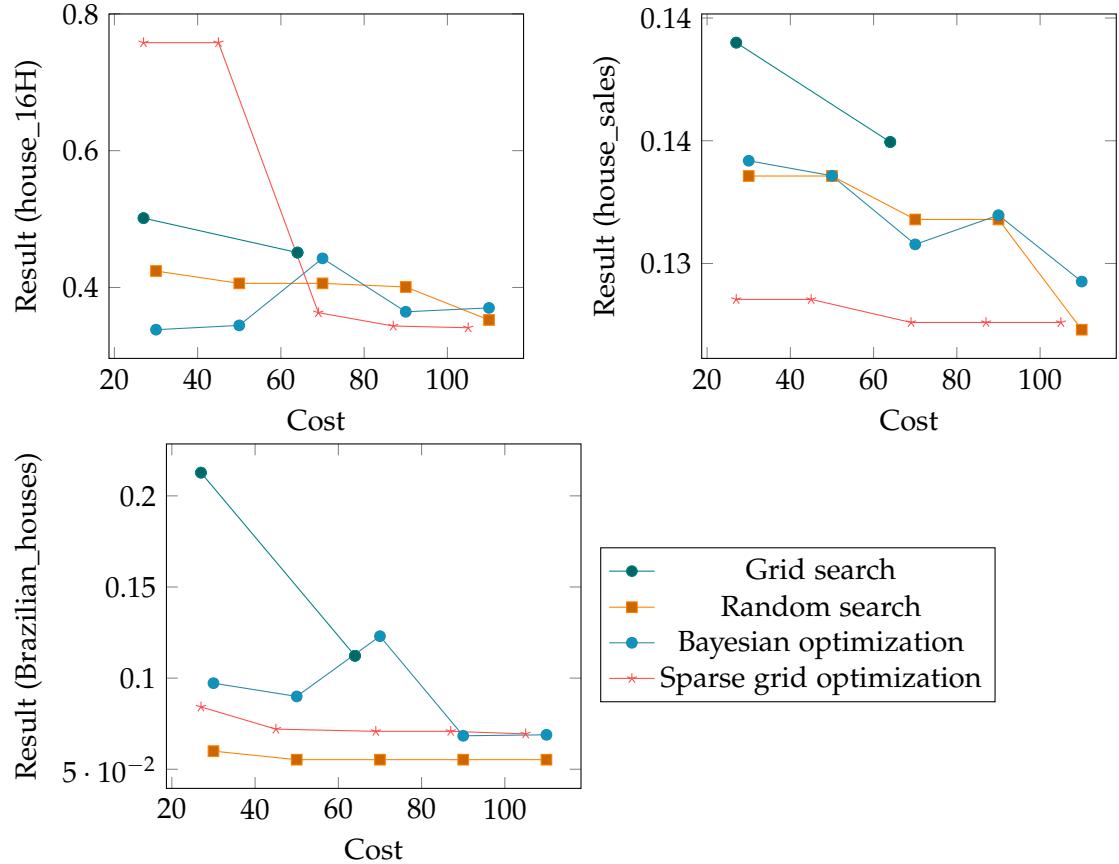


Figure 3.16: Comparison of grid search, random search, bayesian optimization and sparse grid search. The model optimized consists of two layers with 40 neurons each. The number of epochs, batch size, and learning rate are optimized.

The advantage of adaptive sparse grid optimization compared to a homogeneous grid is adaptivity meaning that more evaluations are done in a region of already small values. Especially with the house\_sales dataset, the sparse grid optimization achieves good results already with a small cost. Although in general, this method needs at least some cost to generate a grid that is precise enough to find the optimal point. This can be seen in the case of the house\_16H dataset. At first, the resulting mean average percentage error is very high compared to the other three techniques but after a cost of 69, the sparse grid optimization achieves the best results with the smallest error.

To also analyze the behavior in settings of higher dimensionality, we now also add

the number of layers and the number of neurons per layer as the fourth and fifth variables to the hyperparameter space. The other variables and their intervals stay the same. The number of layers can now range from 2 to 21 and each layer can have 1 to 20 neurons. The resulting errors depending on the cost for the same three datasets as in Figure 3.16 can be seen in Figure 3.17.

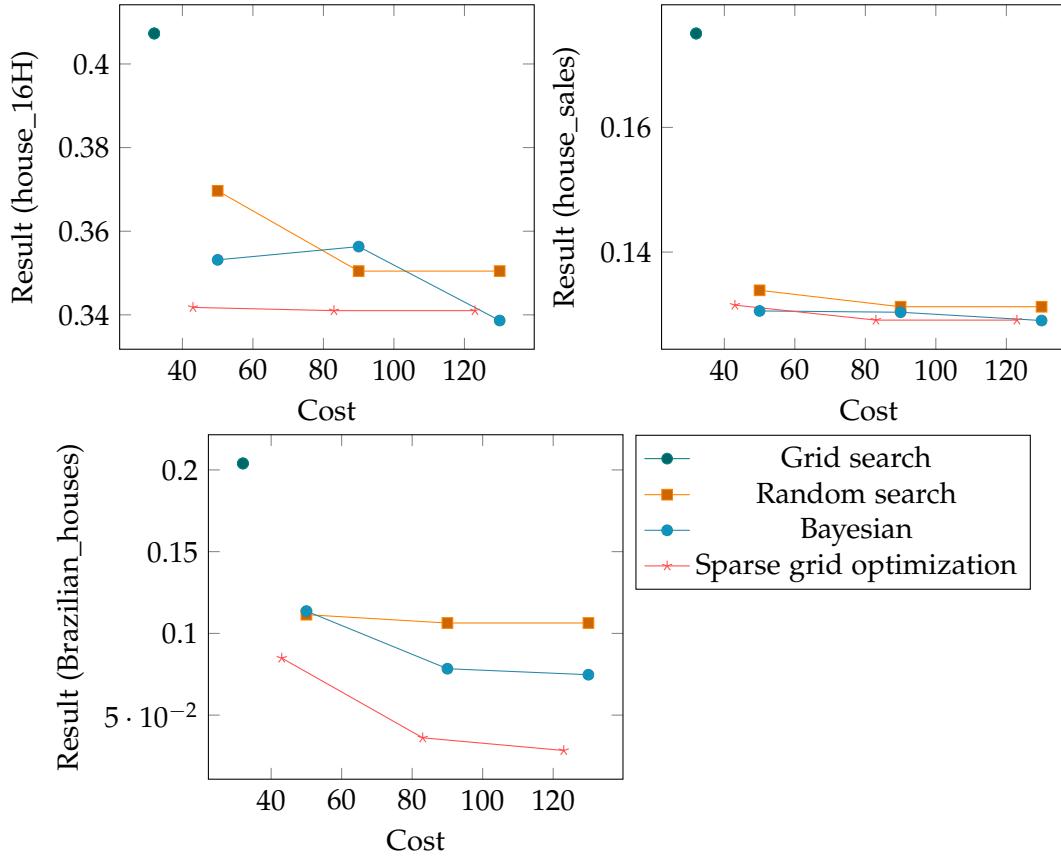


Figure 3.17: Comparison of grid search, random search, bayesian optimization and sparse grid search. The number of epochs, batch size, and learning rate are optimized as well as the number of layers and number of neurons per layer.

The big advantage of the sparse grid is the efficiency in higher dimensions. This can be seen in Figure 3.17. Here with 5 dimensions and a maximum limit for a cost of 130, the normal grid search can only use 2 different values for each hyperparameter leading to 32 function evaluations. That is the reason why we can only see one point in each of the three plots for the datasets. The big advantage of sparse grid optimization is that

fewer grid points are needed and those basis points generated are also adaptive to the problem leading to a fast decrease of error. For almost all three datasets, the sparse grid optimization finds the configuration leading to the smallest mean absolute percentage error.

### 3.4.3 Nine-dimensional Experiment with MNIST Dataset

To also compare the different approaches with a convolutional neural network for object detection, we use a small model with the MNIST dataset. Nine different hyperparameters are optimized. An overview of the type and the ranges can be seen in Table 3.6.

Table 3.6: Hyperparameters with their type and interval for the nine dimensional space. The values for the Int-Interval are discretized with Python's `int()` function. The learning rate is sampled logarithmic and the dropout probability can take continuous values between 0 and 1.

Hyperparameter	Type	Interval
Epochs	Int-Interval	[1, 10]
Batch size	Int-Interval	[200, 1000]
Learning rate	Log-Interval	$[10^{-10}, 10^{-1}]$
Number conv layers	Int-Interval	[1, 3]
Number fully connected layers	Int-Interval	[1, 3]
Kernel size	Int-Interval	[1, 4]
Pool size	Int-Interval	[1, 3]
Neurons in fully connected	Int-Interval	[1, 7]
Dropout probability	Interval	[0, 0.999]

The input shape is always  $(28, 28, 1)$  which defines the first layer of the network. After that, blocks consisting of convolutional and pooling layers are added. The number of blocks is defined by the hyperparameter number of convolutional layers. The kernel and pool size of the 2D convolution and the following pooling layer is defined by the kernel size and pool size and they are optimized. After that, fully connected layers are added. The number of layers as well as the number of neurons per layer are optimized with the two hyperparameters. Between the convolutional part and the fully connected layers, a dropout layer is added. The probability is optimized as well. The other hyperparameters like epochs, batch size, and learning rate are optimized like in the previous experiments. As an optimizer, we used the Adam optimizer and the loss function is the categorical cross-entropy.

The evaluation of the network is done as follows. The dataset is fetched with the predefined training and test set from keras. The model is fit on the training set with a validation split of 10%. It is not shuffled as we want to have reproducibility because the same hyperparameter configuration should produce the same result. This is also the reason why we reset the seeds of Python’s random functions to always have the same weight initialization of the network. After the training step, we evaluate the model using the test set. As an optimization metric, we use the negative accuracy of the prediction as we still minimize. The resulting accuracy depending on the increasing cost is depicted in Figure 3.18.

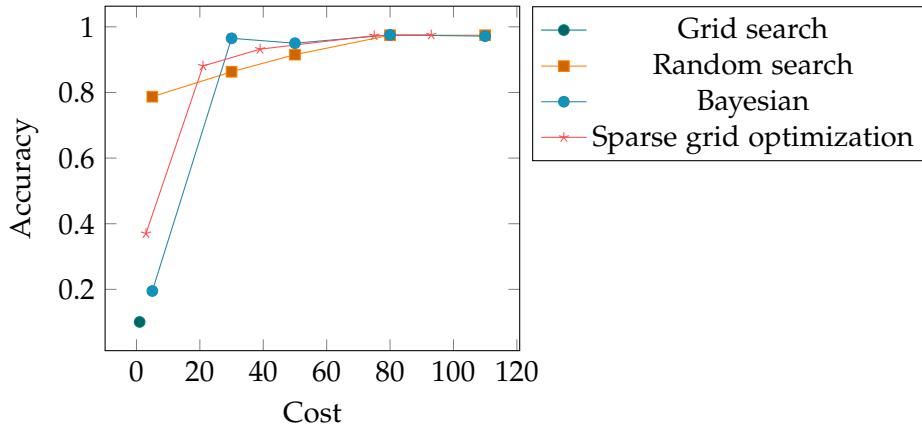


Figure 3.18: Hyperparameter optimization of a convolutional neural network on the MNIST dataset with different algorithms. The hyperparameter intervals are presented in Table 3.6.

Again, with increasing cost, the algorithms find better configurations leading to higher accuracy. In this example, the grid search can only evaluate the model with one single configuration. This is due to the reason that the next step would be  $2^9 = 512$  different configurations where it would try out 2 different values for each hyperparameter.

The first points for sparse grid optimization and bayesian search are also very low, but with a bit higher cost, the performance increases very much. For the random search, even the first point with a budget of 5, the best accuracy is already at about 79%.

For the sparse grid search, the advantage compared to the normal grid search is that there is no *curse of the dimensionality*. The performance is comparable with the other two algorithms, random search and bayesian optimization.

The best configurations found by each algorithm for the problem are depicted in Table 3.7.

Table 3.7: Overview over the best configurations found by the different algorithms grid search (GS), random search (RS), bayesian optimization (BO) and sparse grid optimization (SG). The configuration is given as tuple (epochs, batch size, learning rate, number conv layers, number fully connected layers, kernel size, pool size, neurons per fc layer, dropout probability).

Algorithm	Configuration	Accuracy	Cost
GS	(5, 600, $10^{-6}$ , 2, 2, 2, 4, 0.5)	10.1%	1
RS	(9, 975, 0.0173, 2, 1, 3, 1, 6, 0.619)	97.4%	80
BO	(6, 584, $10^{-2.17}$ , 2, 1, 3, 1, 5, 0.281)	97.5%	80
SG	(7, 400, $10^{-2}$ , 2, 2, 2, 2, 5, 0.5)	97.5%	93

We can observe that the algorithms find different best configurations for the convolutional model. However, the resulting best accuracy is very similar. This is normal for many machine learning problems. There are many different possibilities for very good model performances. In this case, we already achieve a high accuracy of about 97%. However, it is still possible to further increase this performance by e.g. increasing the number of epochs or using a more complex neural network. But this is not what we want to analyze in this thesis.

### 3.4.4 Comparison with Implementation of other Authors

A similar experiment with the same dataset is presented in [62]. The authors concentrate on bayesian optimization and optimize two hyperparameters. The first one is the batch size with an interval [20, 2000]. The second one is the learning rate which can take values from [0, 1]. The network that is used for predicting the class of the images is a small convolutional neural network. The architecture can be seen in Table 3.8.

Table 3.8: Architecture used in [62] for MNIST image classification.

Type	Filters	Kernel size	Activation function
Convolutional 2D	32	(5, 5)	ReLU
Max Pooling 2D		(2, 2)	
Convolutional 2D	32	(5, 5)	ReLU
Max Pooling 2D		(2, 2)	
Dense			Softmax

As an optimizer, they used the gradient descent and the loss function is the categorical cross-entropy. The metric to be optimized is accuracy.

In [62], it is not described for how many epochs they train the neural network which makes it hard to directly compare the algorithms. Additionally, we set the interval of the batch size to  $[20, 1020]$  because of hardware limitations. The learning rate is in our case again a logarithmic hyperparameter with a lower bound of  $10^{-16}$  and an upper bound of 1. For the neural network evaluation, we train the model on the training set with a validation split of 10%. The resulting score is then defined as the accuracy of the predictions of the test set. The result of the experiment with our four different algorithms is presented in Figure 3.19.

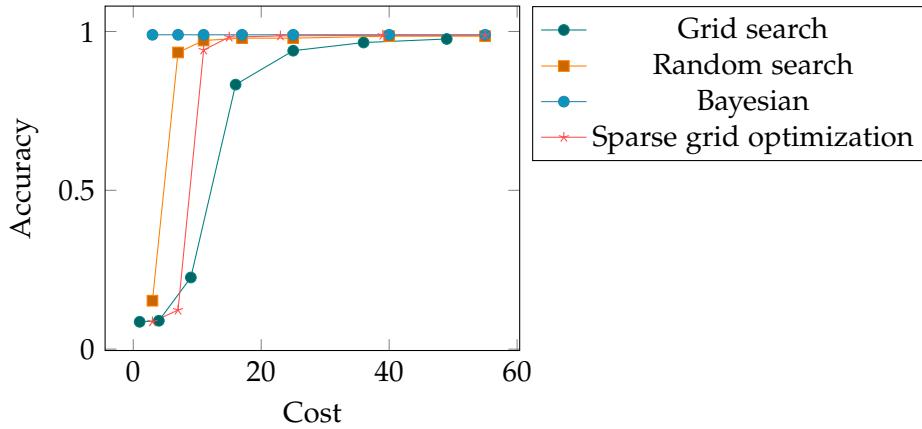


Figure 3.19: Resulting accuracy with increasing budget for the grid search, random search, bayesian optimization and sparse grid optimization. The MNIST dataset and a small convolutional neural network were used.

Again, with increasing cost, the performance of the optimization techniques improves. The absolute highest accuracy is achieved by the bayesian optimization with 98.96% followed by the sparse grid optimization with 98.83%, and random search with 98.52%. The lowest accuracy is achieved by grid search with a value of 97.64%. The authors of [62] reach an accuracy of 99.14% with a budget of 50, however, we do not know for how many epochs they trained the network. For the evaluation of our four algorithms, we always used exactly ten epochs. The best configurations found by the techniques are depicted in Table 3.9.

Table 3.9: Best configurations and corresponding test accuracy found by the four algorithms for the MNIST dataset.

Algorithm	Batch size	Learning rate	Accuracy
Grid search	91	0.01000	0.9764
Random search	42	0.03448	0.9852
Bayesian optimization	57	0.1979	0.9896
Sparse grid optimization	270	0.3162	0.9883

The results shown in Table 3.9 indicate that in this case, a rather small batch size leads to better testing results. This shows that the reduced interval is not restricting the performance of our algorithms. For this experiment, the authors in [62] do not provide the configuration found by their algorithm which makes it hard to compare the performances.

### 3.5 Iterative Adaptive Random Search

In the previous analysis of the sparse grid optimization for finding suitable hyperparameter configurations, we have seen that using sparse grids has advantages compared to the other algorithms in some situations. However, pure random search is very powerful in most cases. One possible adaption of this method is to add iterative refinement like it is done in the grid generation of sparse grids. This is a promising way of combining the advantages of iterative methods and random search which allows to evaluate many different values for each hyperparameter. This is why we introduce a new algorithm for optimizing the hyperparameters of machine learning models in an efficient manner.

#### 3.5.1 Implementation

The algorithm starts with initial points that are distributed randomly in the whole search space. The number of these first points is variable and can be given by the user. A suitable value is dependent on the problem and the number of hyperparameters to be optimized. In each following iteration, one point is selected to be refined. The refinement criterion is similar to the Ritter-Novak criterion used for the sparse grid generation. The point  $i$  that minimizes

$$(rank_i + 1)^{1-\gamma} \cdot (level_i + refinements_i + 1)^\gamma \quad (3.7)$$

is refined. Here, the rank is the position of the point in the list of all points with ascending function values. The adaptivity parameter  $\gamma$  can be used to balance exploration

and exploitation. Each point additionally has an attribute level which is 0 for the initial points. In each iteration, the level of the added points is one higher than the level of the refined point. The refinements attribute is increased each time the point is refined.

For the refinement strategy, meaning where the points are added, random points are sampled following a distribution on a specific domain. There are different possible techniques for refinement. In general, the algorithm takes the steps as depicted in Algorithm 2. The iterations take place as long as the number of points where the function is evaluated is smaller or equal to the budget which has to be given.

---

**Algorithm 2**

**Input**  $m$ : number of initial points,  $n$ : number of refinements per iteration, **budget**: upper bound for evaluations,  $\gamma$ : adaptivity parameter

**Output points**: Array of all evaluated points

```

for  $\min(m, \text{budget})$  do
    sample point uniformly in search space
    add point with level 0 to array
end for
while Number of points +  $n \leq \text{budget}$  do
    sort points with ascending function value
    select point  $p_{\text{ref}}$  according to refinement criterion
    for  $n$  do
        sample point according to refinement strategy
        add point with level  $p_{\text{ref}}.\text{level} + 1$ 
        increase  $p_{\text{ref}}.\text{refinements}$ 
    end for
    increase number of points by  $n$ 
end while
return array of points

```

---

As input, the algorithm 2 expects the number of initial points  $m$ , the number of points to be added in each iteration, the upper bound for the number of function evaluations called budget and the adaptivity parameter  $\gamma$ . The overall output is the array of all points where the function was evaluated.

The first loop is for sampling the initial points. The number of iterations is defined with  $\min(m, \text{budget})$  to assure that the number of function evaluations is smaller or equal to the budget. All initial points have level 0.

In the following, the refinements are done in the loop as long as the budget is not

exceeded. First, the array is sorted with the ascending function value. Then, one point is selected based on the refinement criterion 3.7. The rank of the point is then just the index in the sorted array.

In the next loop which is iterated  $n$  times, a new point is sampled according to the concrete refinement strategy. This point is then added with a level which is one higher than the one selected to be refined. Additionally, the `refinements` parameter of the point selected is increased by one.

The general idea of the refinement strategy is to sample points which are near to the point that is being refined because this point is either promising due to a small function value and thus rank, or it has not been refined that often. Concretely, we provide three different strategies which are presented in the following.

### Interval Based Refinement Strategy

For this first refinement strategy, intervals in each dimension are used. The bounds are defined by neighboring points with and the sampling is done uniformly. An example for such a refinement is presented in Figure 3.20.

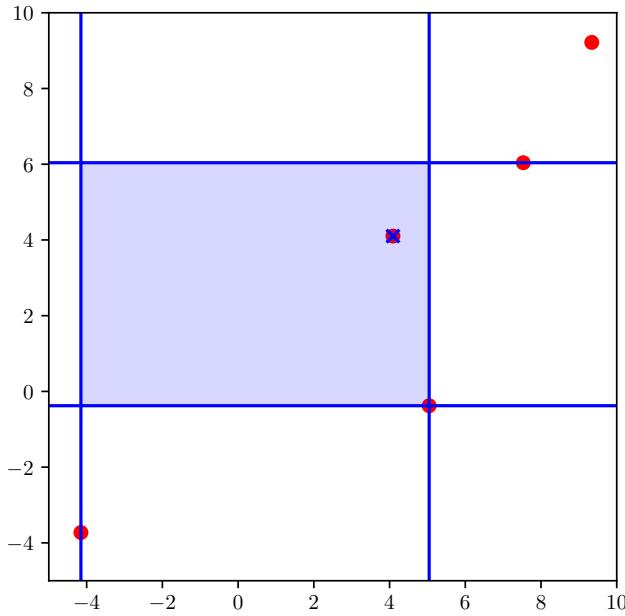


Figure 3.20: Example for the interval based refinement strategy. The point being refined is indicated with the blue cross and the lines show the upper and lower boundary in each direction. The sampling is done uniformly in the blue area.

In the example in Figure 3.20, the number of initial points is 5. The point with the blue cross is selected by the refinement criterion defined with 3.7. In each dimension, an interval is defined with the bounds from the next points in each dimension. In this case, the interval in the horizontal direction is  $[-4.151, 5.045]$  and in the vertical dimension  $[-0.3779, 6.039]$ . Note that only points with the same or smaller level are considered when building the interval.

### Uniform D-Ball Sampling Strategy

This strategy is based on uniform sampling inside a multi-dimensional ball. The point that is refined is the center of this ball and we define a certain radius of the ball. For the uniform sampling, multiple possible algorithms exist [63]. Some of them suffer from the curse of the dimensionality. One example is the rejection method where a value from  $[0, 1]^d$  is sampled and rejected if it is not inside the ball. The probability of a sample being rejected increases with higher dimensionality. Another method is using the normal distribution. For a random variable  $Y \sim N_d(0_d, 1_d)$ ,  $S_n = \frac{Y}{\|Y\|}$  is uniformly

distributed on the d-sphere which is the surface of the ball. The last step is to multiply  $S_n$  by  $U^{\frac{1}{d}}$  where  $U$  has the uniform distribution on the interval between 0 and 1. Now the sampling is done inside the standard d-ball with a radius of 1. The final value just has to be multiplied by the radius to change the sampling to a d-ball with an arbitrary size. One two-dimensional example can be presented in Figure 3.21.

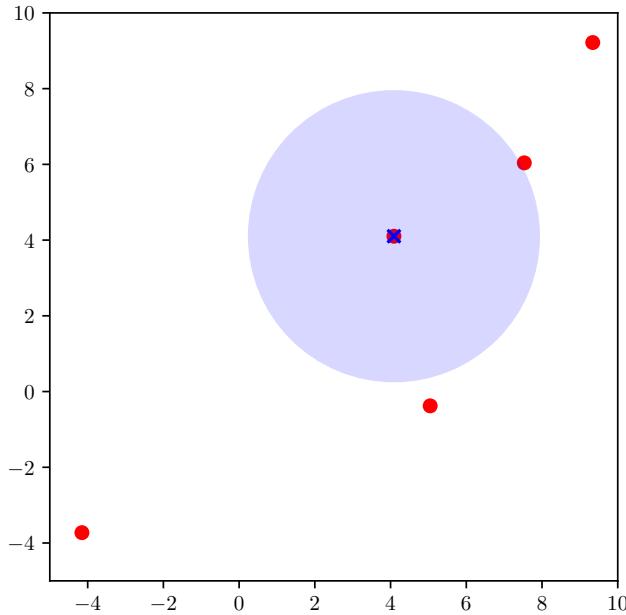


Figure 3.21: Example for the uniform d-ball sampling strategy. The point being refined is indicated with the blue cross and the circle is the area where the uniform sampling is done.

In the example depicted in Figure 3.21, five initial points (colored in red) are sampled. The one marked with the blue cross is chosen to be refined in this iteration. The radius is depending on the level of the point and the smallest and maximum distance to other points. It is computed with

$$r = \frac{dist_{max} + dist_{min}}{(level + 2) \cdot 2} \quad (3.8)$$

where  $dist_{max}$  and  $dist_{min}$  are the highest and lowest distance to other points, respectively. The  $level$  is the level of the current point that is refined. With this, the radius of the ball at points with higher levels is smaller leading to samples near to the point being refined.

### Normal Distribution Sampling Strategy

The last alternative is not sampling uniformly but based on the normal distribution. The idea behind this strategy is that we might be interested in sampling new points near the one that has to be refined in this iteration. This might lead to better results because the point that is refined might already be very promising. Therefore, in each dimension, the coordinate is sampled. A two-dimensional example is presented in Figure 3.22.

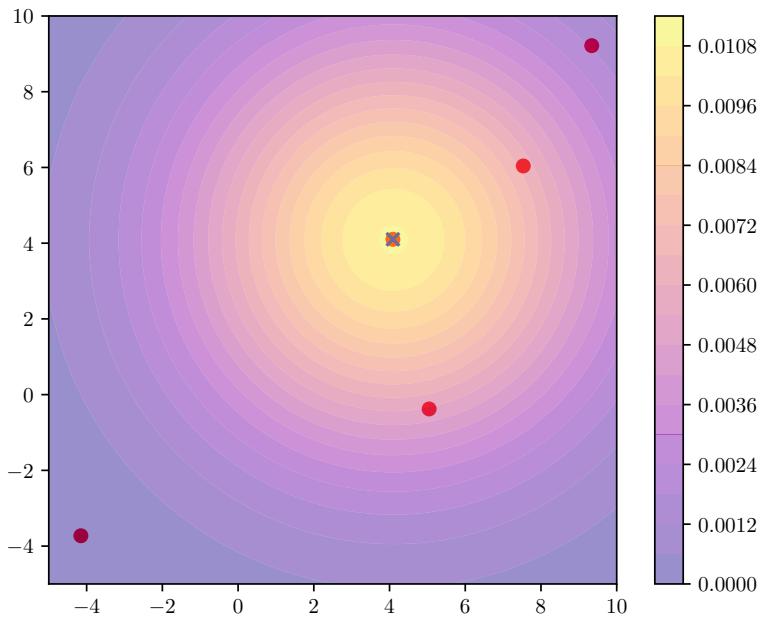


Figure 3.22: Example for the normal distribution sampling strategy. The point being refined is indicated with the blue cross and contour represents the likelihood of samples.

The example in Figure 3.22 shows five initial points in red. The point marked with the blue cross is refined in this iteration step. The contour plot in the background indicates the probability distribution for new points. In each dimension, a normal distribution is constructed with mean and standard deviation with

$$\begin{aligned}
 dist &= \frac{dist_{max} + dist_{min}}{(level + 2) \cdot 2} \\
 lower &= \min(\max(coord - dist, bound_{lower}), bound_{upper}) \\
 upper &= \max(\min(coord + dist, bound_{upper}), bound_{lower}).
 \end{aligned} \tag{3.9}$$

Here,  $coord$  is the coordinate of the point refined in the corresponding dimension,  $bound_{lower}$  and  $bound_{upper}$  represent the bounds of the interval of the hyperparameter in this dimension and the distance  $dist$  is calculated the same way as the radius of the second refinement strategy. Additionally, after sampling a point, it is ensured that the point lays inside the search space with

$$\begin{aligned} coordinate &= \max((coordinate, bound_{lower})) \\ coordinate &= \min((coordinate, bound_{upper})). \end{aligned} \tag{3.10}$$

This is done for each dimension so that for all  $coordinate$  variables drawn, the sample is not higher than the upper bound and not lower than the lower bound in each dimension.

For the implementation, we added a new subclass inheriting `Optimization`. It has an additional parameter for the choice of refinement strategy.

### 3.5.2 Analysis of Parameters with Functions

In this case, we also have the problem that the optimal configuration of hyperparameters for the model evaluation is not known in advance. This is why we first analyze the algorithm with functions where we know the optimum. We will especially focus on the following parameters of the adaptive random search:

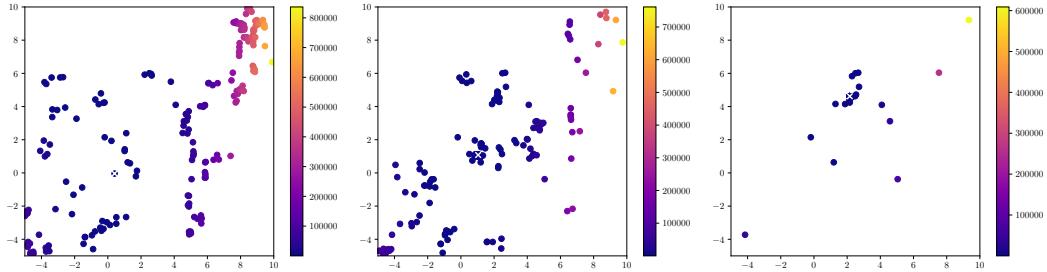
- Adaptivity parameter
- Number of initial points
- Number of refinements per iteration
- Refinement criterion
- Budget

We will optimize the Rosenbrock function (see Table 3.1 and Figure 3.1) in two dimensions for additional visualization. The optimal point is at  $(1,1)$  with a function value of 0.

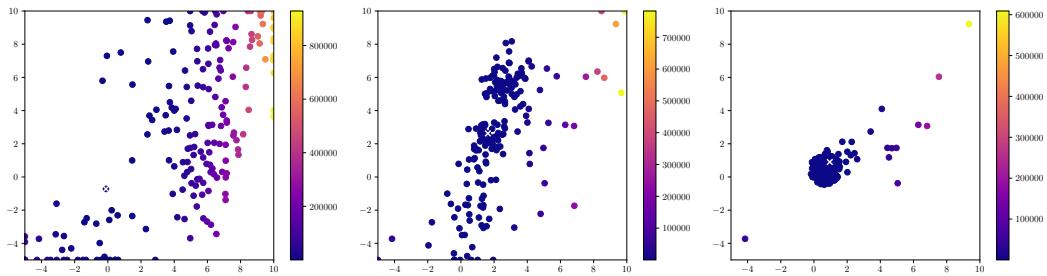
#### Adaptivity Parameter

The first parameter we want to analyze is the adaptivity parameter  $\gamma$  to check if the algorithm behaves as expected. Especially the refinement criterion and strategy are

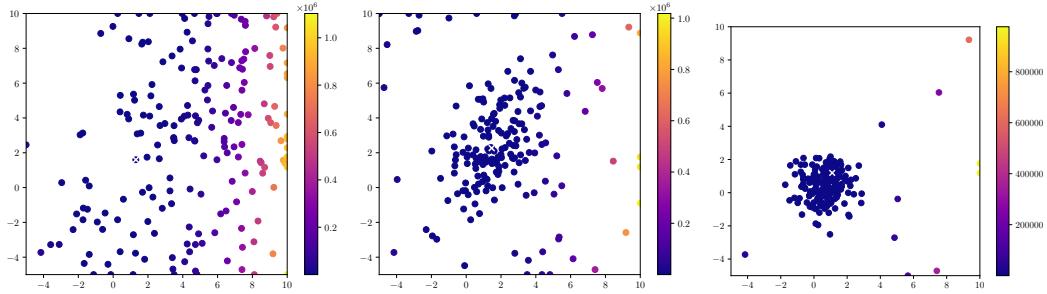
influencing the behavior depending on this parameter. Therefore, we want to visualize how the points are distributed depending on the concrete value. We optimize the Rosenbrock function (see Table 3.1 and Figure 3.1) in two dimensions to visualize the resulting points. The optimal point is at  $(1, 1)$  with a function value of 0. The results with a 200 budget and 5 initial points are depicted in Figure 3.23.



(a) Resulting points for the **interval based refinement strategy**. Adaptivity parameter  $\gamma$ : 1.0 (left), 0.75 (center), 0.0 (right).



(b) Resulting points for the **uniform d-ball sampling strategy**. Adaptivity parameter  $\gamma$ : 1.0 (left), 0.75 (center), 0.0 (right).



(c) Resulting points for the **normal distribution sampling strategy**. Adaptivity parameter  $\gamma$ : 1.0 (left), 0.75 (center), 0.0 (right).

Figure 3.23: Generated points for optimizing the Rosenbrock function with interval based refinement strategy (3.23a), uniform d-ball sampling strategy (3.23b), and normal distribution sampling strategy (3.23c). In each case, the left one has adaptivity parameter  $\gamma = 1.0$  which distributes the points most homogeneous. In the center with  $\gamma = 0.75$ , they are concentrated on few areas and in each right case where  $\gamma = 0.0$ , the grid is most adaptive and almost all points are in the same region.

The resulting points in Figure 3.23 show the behavior of the algorithms depending

on the adaptivity parameter  $\gamma$ . In all three cases, the points are most distributed in the whole domain for  $\gamma = 1.0$  (left grids). This is due to the refinement criterion 3.7 which then selects the candidates in that way, that all points are refined as equally as possible which makes the distribution very homogeneous. In the center cases with  $\gamma = 0.75$ , the trade-off between selecting points that have low rank and ones that have not been refined that often is balanced. The second extreme case is with  $\gamma = 0.0$ . This always leads to the candidate point with the smallest rank (smallest function value). This can be seen in the grid because the points are not distributed at all. Note that you can directly see the difference between the uniform d-ball sampling strategy (3.23b, right grid) and the normal distribution sampling strategy (3.23c, right grid) because in the first case, the points are distributed uniformly in this circle and in the lower strategy, the density of points is higher in the center of the region around  $(1, 1)$ .

The concrete optimum found in each case is presented in Table 3.10.

Table 3.10: Resulting optimum found by the three different refinement strategies interval based refinement strategy (1), uniform d-ball sampling strategy (2), and normal distribution sampling strategy (3). The best of each refinement algorithm is marked in bold.

Alternative	Adaptivity parameter	Coordinates	Error
1	1.0	[0.4059, -0.04055]	4.566
	0.75	<b>[1.039, 1.080]</b>	<b>0.001519</b>
	0.0	[2.159, 4.618]	1.517
2	1.0	[-0.1135, -0.7218]	55.21
	0.75	[1.627, 2.625]	0.4376
	0.0	<b>[0.9542, 0.9028]</b>	<b>0.008060</b>
3	1.0	[1.305, 1.598]	1.208
	0.75	[1.465, 2.265]	1.651
	0.0	<b>[1.098, 1.194]</b>	<b>0.02369</b>

Table 3.10 shows that each of the algorithms finds a point very near to the optimum which is at  $(1, 1)$ . The table also indicates that each refinement strategy has a different best adaptivity parameter. For the second two alternatives, the most adaptive grid achieves the best result while a more homogeneous one is better for the first algorithm.

The experiments by now were only made with 3 distinct values for the adaptivity parameter and this only leads to assumptions that the evaluated parameters are the best in each case. To further analyze this parameter, the Rosenbrock and additionally,

the Rastrigin function were solved with 11 different adaptivity parameters between 0 and 1 with a step size of 0.1. The results for each of the three algorithms are presented in Figure 3.24.

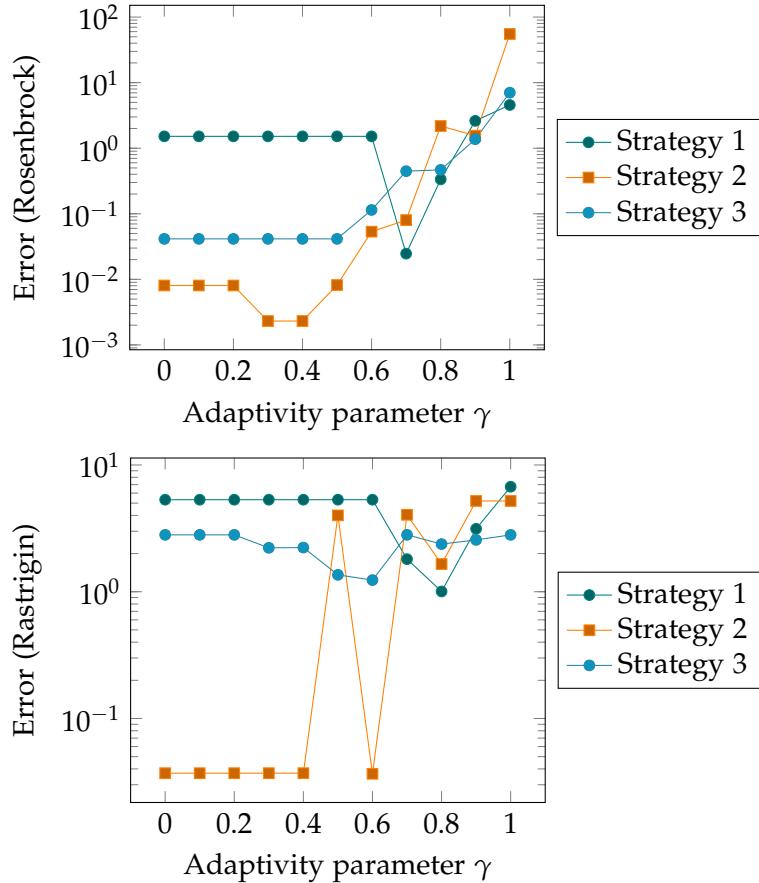


Figure 3.24: Error of the found optimum by the different refinement strategies for the Rosenbrock (top) and the Rastrigin (bottom) function.

The results in Figure 3.24 confirm that each strategy has its own "sweet spot" for the adaptivity parameter. For the interval-based refinement (Strategy 1), this is at about 0.7 – 0.8. For the second one which is the uniform d-ball sampling strategy, good results are achieved especially for values about 0.3 – 0.4. The last strategy based on normal distribution sampling achieves good results with  $\gamma = 0.6$ . One second observation we can make is that the perfect value for this parameter also depends on the optimization problem. However, these values are always in a small interval. This is the reason why we will set the parameter fixed for each of the refinement strategies. For the first one,

we set  $\gamma_1 = 0.75$ , for the second one  $\gamma_2 = 0.35$  and for the last strategy, we use the value  $\gamma_3 = 0.6$ .

### Number of initial points

The second parameter we want to analyze is the number of initial points. Intuitively, a high value for this parameter makes the algorithm behave more similarly to the normal random search because more random points on the whole domain are sampled and fewer refinement steps are done. We, therefore, optimize the three test functions Rosenbrock, Rastrigin, and Eggholder with a different number of initial points and compare the Errors of the optimum found. The overall budget is 200 and the results are shown in Figure 3.25. In each case, 10 runs are evaluated and the average error is plotted.

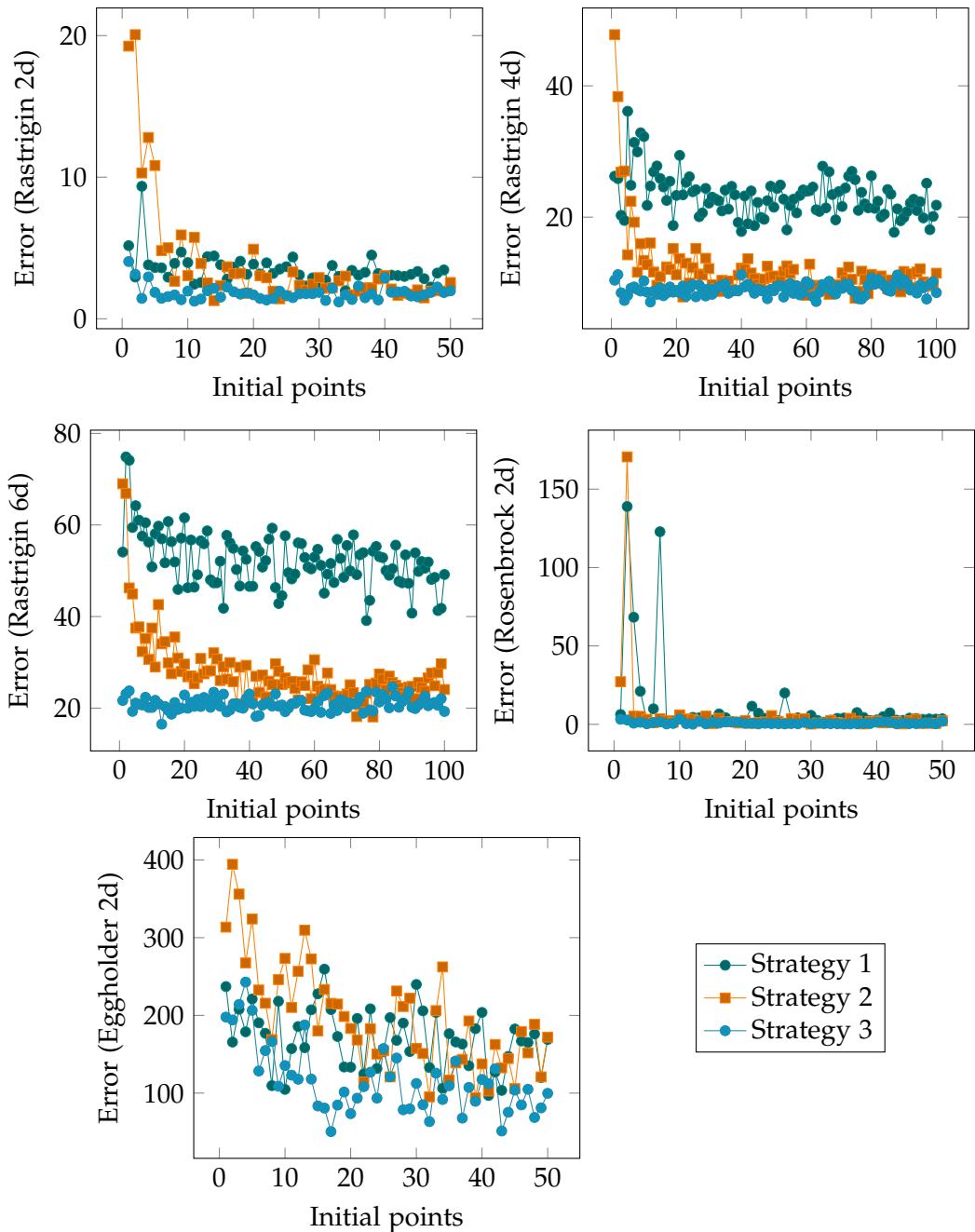


Figure 3.25: Error of the optimum found by the algorithm depending on the number of initial points. The budget is 200 in all two dimensional cases and 400 in the 4- and 6-dimensional problem. Each point is the average of 10 evaluations.

For the Rastrigin function, the problem was solved for two (Figure 3.25, top left), four (Figure 3.25, top right), and six (Figure 3.25, center left) dimensions. In all three cases, similar behavior can be observed for each refinement strategy. For the first algorithm, the error decreases with increasing number of initial points. This is the same for the second strategy but with the difference that it is increasing much faster and seems to be saturated with a sufficiently high number of initial points. For strategy 3, a lower value compared to the other strategies results in smaller errors of the optimum. As depicted in the upper left plot, a minimal number is also not perfect for this parameter. Good results are achieved with 5 to 15 initial points for the two-dimensional case.

For the Rosenbrock and Eggholder function, the behavior is the same for all alternative refinement strategies. The error decreases with a higher number of initial points. Note that these two test functions are very hard to optimize and a generally high number of function evaluations would lead to better results. In this case, the budget was only 200.

For the behavior on the Rastrigin function, we can conclude that strategy 1 is not the best because the results are getting better when the number of refinements decreases. This means that the algorithm is not performing well. The results for the second and third refinement strategies are very promising as they perform well already for a small number of initial points which means that the refinements are finding good optima.

For the following experiments, we will set the number of initial points with  $d$  hyperparameters to:

- Strategy 1:  $10 \cdot d$
- Strategy 2:  $8 \cdot d$
- Strategy 3:  $7 \cdot d$

#### Number of refinements per iteration

The next parameter that is influencing the performance of the algorithm is the number of refinements per iteration. Intuitively, a higher value for this parameter makes the algorithm less adaptive because the point that has to be refined is chosen fewer times. To analyze this parameter, again the three functions (Rastrigin in 2, 4, and 6 dimensions and Rosenbrock and Eggholder in 2 dimensions) are optimized with a budget of 200 in the case of the 2-d problems and 400 for 4 and 6 dimensions. The resulting errors which are the averages of ten runs with up to 50 refinements per iteration are presented

in Figure 3.26.

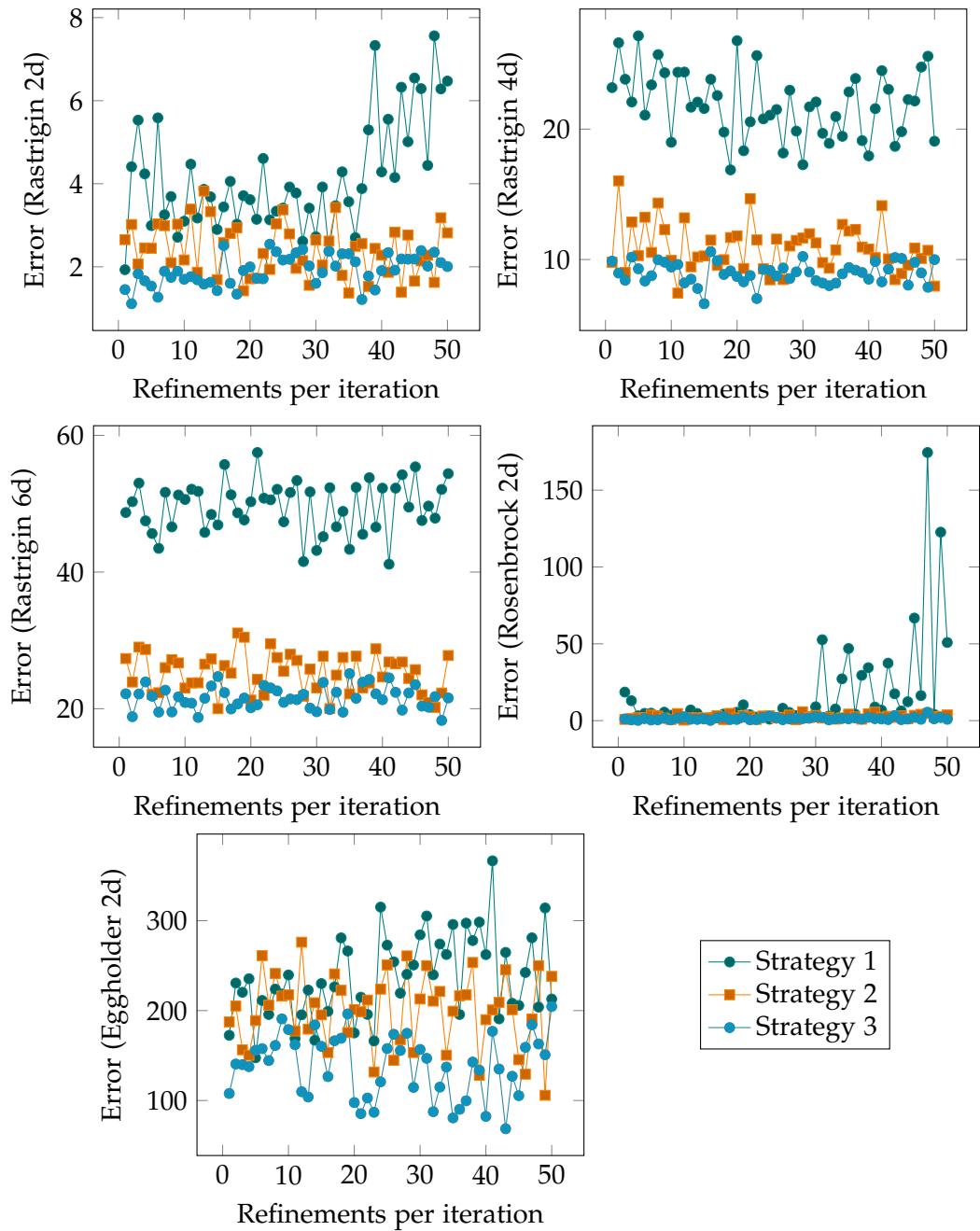


Figure 3.26: Error of the optimum found by the algorithm depending on the number of points added per refinement step. The budget is 200 in all two dimensional cases and 400 in the 4- and 6-dimensional problems. Each point is the average of 10 evaluations.

As depicted in Figure 3.26, this parameter does not have the biggest impact on the performance. However, especially for the first refinement strategy, smaller values sometimes lead to better results as can be seen on the top left (Figure 3.26) and center right (Figure 3.26).

In the following, we set the number of points to be added in each iteration to  $2 \cdot d$  with  $d$  being the dimension of the problem. This is the same number of points that are added in each iteration in the sparse grid generation algorithm.

In summary, the third strategy almost always performs best. This can be seen in Figure 3.25 and Figure 3.26.

### 3.5.3 Hyperparameter Optimization

First experiments -> Problem from Figure 3.9.

### 3.5.4 Comparison with other Techniques

- compare all 5 techniques

## **4 Conclusion and Outlook**

# List of Figures

2.1	Neural network consisting of only one perceptron. The output is computed according to Equation 2.1. . . . .	4
2.2	Neural network consisting of two hidden layers. The connections between the perceptrons are bidirectional. In the forward phase, the intermediate results are given to the neurons to the right and in the backward phase from right to left. . . . .	5
2.3	Comparison of grid search (left) and random search (right) in the two dimensional case. For both techniques, 9 different combinations are evaluated. In the left case, only 3 distinct values for each hyperparameter are set whereas there are 9 different values for each parameter in the random search. Taken from [17]. . . . .	8
2.4	Schematic iteration steps of the bayesian optimization. The maximum of the acquisition function determines the next function evaluation (red dot in the middle). The goal is to find the minimum of the dashed line. The blue band is the uncertainty of the function. Taken from [17]. . . . .	11
2.5	Interpolation of the function $f$ (black line) by its interpolant $u$ (red, dashed) in the nodal basis. Level of the grid is 3 and hat functions are used. Taken from [30]. . . . .	13
2.6	Hierarchical subspaces up to level 3 on the left. On the right, nodal spaces up to level 3. The combination of $W_1$ up to $W_3$ is the same space as $V_3$ . Taken from [30]. . . . .	14
2.7	Interpolation of the function $f$ (black line) by its interpolant $u$ (red, dashed) in the hierarchical basis. Level of the grid is 3 and hat functions are used. Taken from [30]. . . . .	15
2.8	Example of a basis function in two dimensions. It is constructed with the tensor product of two 1d hat functions. Taken from [31]. . . . .	16
2.9	Two dimensional example of a sparse grid with $n = 3$ . Left, the subspaces $W_{\vec{l}}$ can be seen and on the right is the resulting sparse grid. Taken from [31]. . . . .	17

2.10 Example of the 2-dimensional combination technique. Here the blue regular grids are added and the red ones are subtracted. On the left, the normal combination technique can be seen and on the right is an dimension-adaptive version. Taken from [30]. . . . .	18
2.11 Example of the spatially adaptive combination technique in two dimensions. Taken from [37]. . . . .	19
2.12 HPC of level $k = 5$ (red stars) and full grid (black dots). A full grid would have $33^2 = 1089$ and this grid has 147 points. Taken from [43]. . .	23
3.1 Test functions used for evaluating the sparse grid optimization. Each one is plotted with 200 samples in each dimension. Note that the function values of the Rosenbrock function are transformed with $\log_{10}(f(x))$ for better visualization. . . . .	29
3.2 Sparse grid generation depending on the adaptivity parameter $\gamma$ for the Eggholder function. In all cases, the same number of grid points is used. Here, in each of the 249 iterations, 4 new grid points are added resulting in a overall number of 997 function evaluations. . . . .	30
3.3 Sparse grid generation depending on the adaptivity parameter $\gamma$ for the Rosenbrock function. In all cases, the same number of grid points is used. Here, in each of the 249 iterations, 4 new grid points are added resulting in a overall number of 997 function evaluations. . . . .	31
3.4 Sparse grid generation depending on the adaptivity parameter $\gamma$ for the Rastrigin function. In all cases, the same number of grid points is used. Here, in each of the 249 iterations, 4 new grid points are added resulting in a overall number of 997 function evaluations. . . . .	32
3.5 Influence of the adaptivity parameter on the error (difference to the actual optimal value) of the optimum found by the sparse grid. . . . .	34
3.6 Optimization error for different algorithms depending on the number of grid points in two dimensions. The plots show the results with degree 2 (top left), degree 3 (top right) and degree 5 (bottom). The Rastrigin function was used. . . . .	36
3.7 Resulting optimal points from local (red points) and global (black points) optimization algorithm. In the background, the contour of the interpolated function is depicted with the corresponding sparse grid points in white. The left one has 5 grid points, the one in the center 77, and the right one 997. The degree of the B-splines on the sparse grid is 2. . . .	37
3.8 Error of the optimization depending on the number of grid points used for different dimensions of the Rastrigini function. . . . .	38

3.9	2-layer (with 30 neurons each) fully connected network evaluation on the diamonds dataset depending on the number of epochs and learning rate of the Adam optimizer. The result is plotted with $\log_{10}(\text{Result})$ for better visualization. . . . .	40
3.10	Analysis of sparse grid with machine learning evaluation for different adaptivity parameters (0.85 3.10a and 0.5 3.10b), each with 29 and 77 grid points. . . . .	41
3.11	Optimization steps of gradient descent algorithm for 5 (top left), 9 (top right), 29 (bottom left) and 49 (bottom right) grid points. In the background of each plot, the contour of the interpolated function is shown. The function evaluated is the same as depicted in Figure 3.9. . . . .	43
3.12	Comparison of optimization algorithms on the sparse grid with increasing number of grid points in two and three dimensions. Epochs, learning rate, and the batch size of a two-layer neural network on the diamonds dataset for regression are optimized. Result is the mean absolute percentage error. . . . .	45
3.13	Sparse grid optimization with the interpolated function in the background, the white grid points, and the optimizer steps in red. Four different budgets are used (30: top left, 50: top right, 100: bottom left, 200: bottom right. Only three different configurations with values for batch size from 100, 400, 2000 and values for epochs from 1, 5, 13. . . . .	47
3.14	Comparison of grid-, random search, bayesian optimization and sparse grid search for the datasets shown in Table 3.5. Two hyperparameters, epochs and learning rate of the neural network's optimizer, were optimized. . . . .	50
3.15	Comparison of the grids generated for the house_sales dataset. The best configuration found $x_{opt}$ is marked with a white cross. . . . .	52
3.16	Comparison of grid search, random search, bayesian optimization and sparse grid search. The model optimized consists of two layers with 40 neurons each. The number of epochs, batch size, and learning rate are optimized. . . . .	54
3.17	Comparison of grid search, random search, bayesian optimization and sparse grid search. The number of epochs, batch size, and learning rate are optimized as well as the number of layers and number of neurons per layer. . . . .	55
3.18	Hyperparameter optimization of a convolutional neural network on the MNIST dataset with different algorithms. The hyperparameter intervals are presented in Table 3.6. . . . .	57

3.19	Resulting accuracy with increasing budget for the grid search, random search, bayesian optimization and sparse grid optimization. The MNIST dataset and a small convolutional neural network were used. . . . .	59
3.20	Example for the interval based refinement strategy. The point being refined is indicated with the blue cross and the lines show the upper and lower boundary in each direction. The sampling is done uniformly in the blue area. . . . .	63
3.21	Example for the uniform d-ball sampling strategy. The point being refined is indicated with the blue cross and the circle is the area where the uniform sampling is done. . . . .	64
3.22	Example for the normal distribution sampling strategy. The point being refined is indicated with the blue cross and contour represents the likelihood of samples. . . . .	65
3.23	Generated points for optimizing the Rosenbrock function with interval based refinement strategy (3.23a), uniform d-ball sampling strategy (3.23b), and normal distribution sampling strategy (3.23c). In each case, the left one has adaptivity parameter $\gamma = 1.0$ which distributes the points most homogeneous. In the center with $\gamma = 0.75$ , they are concentrated on few areas and in each right case where $\gamma = 0.0$ , the grid is most adaptive and almost all points are in the same region. . . . .	68
3.24	Error of the found optimum by the different refinement strategies for the Rosenbrock (top) and the Rastrigin (bottom) function. . . . .	70
3.25	Error of the optimum found by the algorithm depending on the number of initial points. The budget is 200 in all two dimensional cases and 400 in the 4- and 6-dimensional problem. Each point is the average of 10 evaluations. . . . .	72
3.26	Error of the optimum found by the algorithm depending on the number of points added per refinement step. The budget is 200 in all two dimensional cases and 400 in the 4- and 6-dimensional problems. Each point is the average of 10 evaluations. . . . .	75

# List of Tables

2.1	Comparison of sparse grids, full grids, and the combination technique in terms of number of grid points and the accuracy. . . . .	19
3.1	Three test functions and their properties. . . . .	28
3.2	Overview over the exact solutions found by the local and global optimizer for the same problem as in Figure 3.7. The actual optimum is at (0, 0), with function value 0 (see Table 3.1). . . . .	38
3.3	Best hyperparameter configuration found by the sparse grid with different adaptivity parameters and number of grid points. The best result is found with $\gamma = 0.85$ and 77 grid points. . . . .	42
3.4	Exact values for the optima found by the sparse grid points and the gradient descent algorithm. The values for $x_{min}^{grid}$ are the configurations evaluated by the sparse grid during the generation and the coordinates in column $x_{min}^{opt}$ are found by the optimizer. In bold, the best function values of the optimum found by the sparse grid and gradient descent algorithm, respectively. . . . .	44
3.5	Overview over the datasets used for the first comparison of the optimization algorithms. They are all available on OpenML [60]. . . . .	49
3.6	Hyperparameters with their type and interval for the nine dimensional space. The values for the Int-Interval are discretized with Python's <i>int()</i> function. The learning rate is sampled logarithmic and the dropout probability can take continuous values between 0 and 1. . . . .	56
3.7	Overview over the best configurations found by the different algorithms grid search (GS), random search (RS), bayesian optimization (BO) and sparse grid optimization (SG). The configuration is given as tuple (epochs, batch size, learning rate, number conv layers, number fully connected layers, kernel size, pool size, neurons per fc layer, dropout probability). . . . .	58
3.8	Architecture used in [62] for MNIST image classification. . . . .	58
3.9	Best configurations and corresponding test accuracy found by the four algorithms for the MNIST dataset. . . . .	60

## *List of Tables*

# Bibliography

- [1] S. M. Malakouti, "Babysitting hyperparameter optimization and 10-fold-cross-validation to enhance the performance of ml methods in predicting wind speed and energy generation," *Intelligent Systems with Applications*, vol. 19, p. 200248, 2023, ISSN: 2667-3053. doi: <https://doi.org/10.1016/j.iswa.2023.200248>.
- [2] N. Gorgolis, I. Hatzilygeroudis, Z. Istenes, and L.-G. Gyenne, "Hyperparameter optimization of lstm network models through genetic algorithm," in *2019 10th International Conference on Information, Intelligence, Systems and Applications (IISA)*, IEEE, 2019, pp. 1–4.
- [3] H. Wang, Z. Lei, X. Zhang, B. Zhou, and J. Peng, "Machine learning basics," *Deep learning*, pp. 98–164, 2016.
- [4] B. Mahesh, "Machine learning algorithms-a review," *International Journal of Science and Research (IJSR). [Internet]*, vol. 9, pp. 381–386, 2020.
- [5] T. O. Ayodele, "Types of machine learning algorithms," *New advances in machine learning*, vol. 3, pp. 19–48, 2010.
- [6] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.
- [7] O.-C. Granmo, "The tsetlin machine—a game theoretic bandit driven approach to optimal pattern recognition with propositional logic," *arXiv preprint arXiv:1804.01508*, 2018.
- [8] L. Rokach and O. Maimon, "Decision trees," *Data mining and knowledge discovery handbook*, pp. 165–192, 2005.
- [9] C. M. Bishop, "Neural networks and their applications," *Review of scientific instruments*, vol. 65, no. 6, pp. 1803–1832, 1994.
- [10] I. N. Da Silva, D. Hernane Spatti, R. Andrade Flauzino, L. H. B. Liboni, S. F. dos Reis Alves, I. N. da Silva, D. Hernane Spatti, R. Andrade Flauzino, L. H. B. Liboni, and S. F. dos Reis Alves, *Artificial neural network architectures and training processes*. Springer, 2017.
- [11] L. R. Medsker and L. Jain, "Recurrent neural networks," *Design and Applications*, vol. 5, pp. 64–67, 2001.

## Bibliography

---

- [12] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, 2021.
- [13] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, *et al.*, "Recent advances in convolutional neural networks," *Pattern recognition*, vol. 77, pp. 354–377, 2018.
- [14] K. O’Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.
- [15] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.
- [16] P. Cunningham, M. Cord, and S. J. Delany, "Supervised learning," *Machine learning techniques for multimedia: case studies on organization and retrieval*, pp. 21–49, 2008.
- [17] M. Feurer and F. Hutter, "Hyperparameter optimization," *Automated machine learning: Methods, systems, challenges*, pp. 3–33, 2019.
- [18] B. Bischl, M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, *et al.*, "Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, e1484, 2021.
- [19] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.
- [20] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of machine learning research*, vol. 13, no. 2, 2012.
- [21] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, 2012.
- [22] R. Andonie, "Hyperparameter optimization in learning systems," *Journal of Membrane Computing*, vol. 1, no. 4, pp. 279–291, 2019.
- [23] E. C. Garrido-Merchán and D. Hernández-Lobato, "Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes," *Neurocomputing*, vol. 380, pp. 20–35, 2020.
- [24] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*, Springer, 2011, pp. 507–523.

## Bibliography

---

- [25] J. Wilson, F. Hutter, and M. Deisenroth, "Maximizing acquisition functions for bayesian optimization," *Advances in neural information processing systems*, vol. 31, 2018.
- [26] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *Artificial intelligence and statistics*, PMLR, 2016, pp. 240–248.
- [27] G. I. Diaz, A. Fokoue-Nkoutche, G. Nannicini, and H. Samulowitz, "An effective algorithm for hyperparameter optimization of neural networks," *IBM Journal of Research and Development*, vol. 61, no. 4/5, 9:1–9:11, 2017. doi: 10.1147/JRD.2017.2709578.
- [28] S. C. Smithson, G. Yang, W. J. Gross, and B. H. Meyer, "Neural networks designing neural networks: Multi-objective hyper-parameter optimization," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2016, pp. 1–8.
- [29] I. Loshchilov and F. Hutter, "Cma-es for hyperparameter optimization of deep neural networks," *arXiv preprint arXiv:1604.07269*, 2016.
- [30] D. M. Pflüger, "Spatially adaptive sparse grids for high-dimensional problems," Ph.D. dissertation, Technische Universität München, 2010.
- [31] J. Garcke, "Sparse grids in a nutshell," in *Sparse grids and applications*, Springer, 2013, pp. 57–80.
- [32] J. Valentin, "B-splines for sparse grids: Algorithms and application to higher-dimensional optimization," *arXiv preprint arXiv:1910.05379*, 2019.
- [33] C. Zenger and W. Hackbusch, "Sparse grids," in *Proceedings of the Research Workshop of the Israel Science Foundation on Multiscale Phenomenon, Modelling and Computation*, 1991, p. 86.
- [34] H.-J. Bungartz and M. Griebel, "Sparse grids," *Acta numerica*, vol. 13, pp. 147–269, 2004.
- [35] M. Obersteiner and H.-J. Bungartz, "A spatially adaptive sparse grid combination technique for numerical quadrature," in *Sparse Grids and Applications-Munich 2018*, Springer, 2022, pp. 161–185.
- [36] M. Griebel, M. Schneider, and C. Zenger, "A combination technique for the solution of sparse grid problems," 1990.
- [37] M. Obersteiner and H.-J. Bungartz, "A generalized spatially adaptive sparse grid combination technique with dimension-wise refinement," *SIAM Journal on Scientific Computing*, vol. 43, no. 4, A2381–A2403, 2021.

## Bibliography

---

- [38] M. Hegland, "Adaptive sparse grids," *Anziam Journal*, vol. 44, pp. C335–C353, 2002.
- [39] H.-J. Bungartz, "Finite elements of higher order on sparse grids," Ph.D. dissertation, Technische Universität München, 1998.
- [40] K. Höllig and J. Hörner, *Approximation and modeling with B-splines*. SIAM, 2013.
- [41] J. Valentin, M. Sprenger, D. Pflüger, and O. Röhrle, "Gradient-based optimization with b-splines on sparse grids for solving forward-dynamics simulations of three-dimensional, continuum-mechanical musculoskeletal system models," *International journal for numerical methods in biomedical engineering*, vol. 34, no. 5, e2965, 2018.
- [42] E. Novak and K. Ritter, "Global optimization using hyperbolic cross points," *State of the art in global optimization: computational methods and applications*, pp. 19–33, 1996.
- [43] F. Duan, R. Živanović, S. Al-Sarawi, and D. MBA, "Induction motor parameter estimation using sparse grid optimization algorithm," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 4, pp. 1453–1461, 2016.
- [44] M. Saska, I. Ferenczi, M. Hess, and K. Schilling, "Path planning for formations using global optimization with sparse grids," in *Proc. of The 13th IASTED International Conference on Robotics and Applications (RA 2007)*, 2007.
- [45] M. Hülsmann and D. Reith, "Spagrow—a derivative-free optimization scheme for intermolecular force field parameters based on sparse grid methods," *Entropy*, vol. 15, no. 9, pp. 3640–3687, 2013.
- [46] S. Chen and X. Wang, "A derivative-free optimization algorithm using sparse grid integration," 2013.
- [47] S. Sankaran, "Stochastic optimization using a sparse grid collocation scheme," *Probabilistic engineering mechanics*, vol. 24, no. 3, pp. 382–396, 2009.
- [48] C. Hamzaçebi and F. Kutay, "A heuristic approach for finding the global minimum: Adaptive random search technique," *Applied Mathematics and Computation*, vol. 173, no. 2, pp. 1323–1333, 2006.
- [49] H. A. Schilling and S. L. Harris, *Applied numerical methods for engineers using MATLAB*. Brooks/Cole Publishing Co., 1999.
- [50] S. Masri, G. Bekey, and F. Safford, "A global optimization algorithm using adaptive random search," *Applied Mathematics and Computation*, vol. 7, no. 4, pp. 353–375, 1980, ISSN: 0096-3003. doi: [https://doi.org/10.1016/0096-3003\(80\)90027-2](https://doi.org/10.1016/0096-3003(80)90027-2).

## Bibliography

---

- [51] S. Andradóttir and A. A. Prudius, "Adaptive random search for continuous simulation optimization," *Naval Research Logistics (NRL)*, vol. 57, no. 6, pp. 583–604, 2010. doi: <https://doi.org/10.1002/nav.20422>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.20422>.
- [52] M. Schumer and K. Steiglitz, "Adaptive step size random search," *IEEE Transactions on Automatic Control*, vol. 13, no. 3, pp. 270–276, 1968.
- [53] Z. B. Tang, "Adaptive partitioned random search to global optimization," *IEEE Transactions on Automatic Control*, vol. 39, no. 11, pp. 2235–2244, 1994.
- [54] R. Kelahan and J. Gaddy, "Application of the adaptive random search to discrete and mixed integer optimization," *International Journal for Numerical Methods in Engineering*, vol. 12, no. 2, pp. 289–298, 1978.
- [55] D. Gall, "A practical multifactor optimization criterion," *Recent Advances in Optimization Techniques*, pp. 369–386, 1966.
- [56] M. Feurer, J. N. van Rijn, A. Kadra, P. Gijsbers, N. Mallik, S. Ravi, A. Müller, J. Vanschoren, and F. Hutter, "Openml-python: An extensible python api for openml," *arXiv:1911.02490*, 2019.
- [57] J. Valentin and D. Pflüger, "Hierarchical gradient-based optimization with b-splines on sparse grids," in *Sparse Grids and Applications-Stuttgart 2014*, Springer, 2016, pp. 315–336.
- [58] D. Whitley, S. Rana, J. Dzubera, and K. E. Mathias, "Evaluating evolutionary algorithms," *Artificial intelligence*, vol. 85, no. 1-2, pp. 245–276, 1996.
- [59] X.-S. Yang, *Engineering optimization: an introduction with metaheuristic applications*. John Wiley & Sons, 2010.
- [60] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "Openml: Networked science in machine learning," *SIGKDD Explorations*, vol. 15, no. 2, pp. 49–60, 2013. doi: [10.1145/2641190.2641198](https://doi.org/10.1145/2641190.2641198).
- [61] 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, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [62] J. Wu, X.-Y. Chen, H. Zhang, L.-D. Xiong, H. Lei, and S.-H. Deng, "Hyperparameter optimization for machine learning models based on bayesian optimization," *Journal of Electronic Science and Technology*, vol. 17, no. 1, pp. 26–40, 2019, issn: 1674-862X. doi: <https://doi.org/10.11989/JEST.1674-862X.80904120>.

## Bibliography

---

- [63] R. Harman and V. Lacko, "On decompositional algorithms for uniform sampling from n-spheres and n-balls," *Journal of Multivariate Analysis*, vol. 101, no. 10, pp. 2297–2304, 2010, issn: 0047-259X. doi: <https://doi.org/10.1016/j.jmva.2010.06.002>.