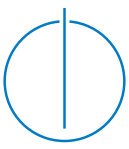
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Neural Network Hyperparameter Optimization with Sparse Grids

Maximilian Michallik



DEPARTMENT OF INFORMATICS

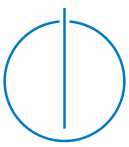
TECHNISCHE UNIVERSITÄT MÜNCHEN

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

1 Introduction

2 State of the Art

Machine Learning [1, 2] 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 labeled and unlabeled data. Additionally, there is reinforcement learning where the model learns by observing the environment [3]. Specific tasks are e.g. classification where input has to be assigned to specific classes, regression where 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 [4], the tsetlin machine [5], and decision trees [6]. One very important class of algorithms is *artificial neural networks* 2.1. After the introduction to neural networks, the hyperparameter optimization is presented with different techniques to improve machine learning models. In the following, sparse grids are presented which will be needed as foundation to hyperparameter optimization of neural networks with sparse grids.

2.1 Introduction to Neural Networks

Neural networks [7, 8] 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.

The output y is computed with

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

The network has n inputs x_i and weights w_i. θ is the activation threshold (also called bias), g is the activation function, and u is the activation potential [8].

This basis building block can then be used to build a more complex architecture with multiple layers. All neural networks have an input layer consisting of $n \in \mathbb{N}$

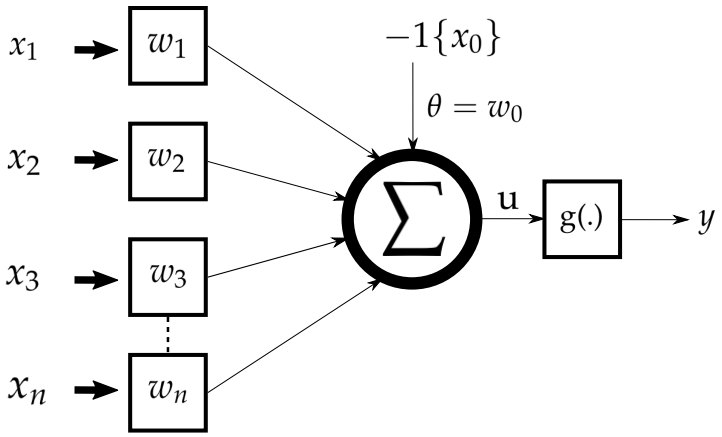


Figure 2.1: Neural network consisting of only one perceptron. The output is computed according to Equation 2.1.

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 determines 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 it can be seen in Figure 2.2.

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

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

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 η the learning rate. The choice of η is fixed before training and usually $0 < \eta < 1$. For the update of the weights of networks with multiple layers, refer to [8].

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

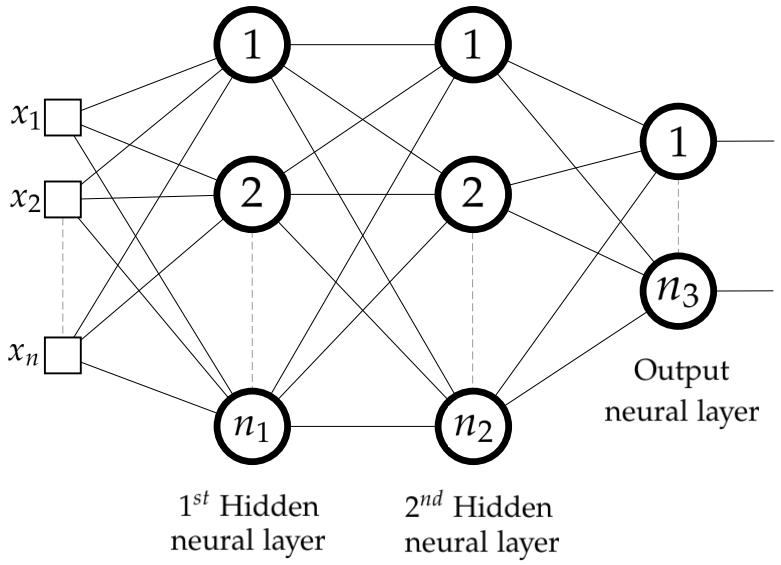


Figure 2.2: Neural network consisting of two hidden layers. The connections between the perceptrons are bidirectional. In the forward phase, the intermediate results are given to the neurons to the right and in the backward phase from right to left.

between layers can improve the performance of the network. Other things 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 [9]. There is also a specific architecture for grid structured data like images. For them, convolutional neural networks are used to extract features [10–12]. For further readings on different architecture choices, refer to [13].

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 the hyperparameters that have to be fixed before training. They are design decisions 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: η 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 λ_{min} from $\Lambda = \Lambda_1 \times \Lambda_2 \times ... \times \Lambda_N$ that minimizes the function f with N hyperparameters with

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

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

$$\forall i : M(x_i) = y_i. \tag{2.4}$$

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 [14].

All in all, the goal is get evaluation scores on the testing data set which can be achieved with Equation 2.3. [15–17]

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 the 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 [18] 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.

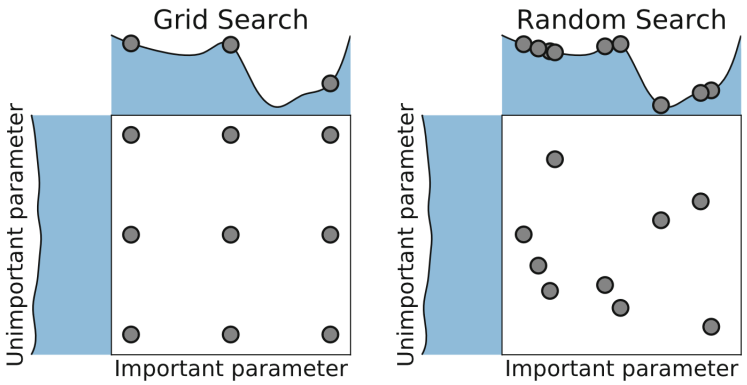


Figure 2.3: Comparison of grid search (left) and random search (right) in the two dimensional case. For both techniques, 9 different combinations are evaluated. In the left case, only 3 distinct values for each hyperparameter are set whereas there are 9 different values for each parameter in the random search. Taken from [15].

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) [19]. This is an iterative approach for optimizing the expensive black box function by modeling 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 [16] which is only suitable if the number of hyperparameters is not too high [20]. 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 [21]. Another possible technique for the surrogate model is using random forests [22]. 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 no much information is gathered are explored. There are many numerous different possibilities for this function [23] but the most used one is the *expected improvement* (EI) which is calculated with

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

If the model prediction y with configuration λ follows a normal distribution [15], it leads to

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

with ϕ and Φ being the standard normal density and standard normal distribution and f_{min} the best result so far.

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

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 λ^+. This point is evaluated and added to the archive. The overall result of the algorithm is the λ which is the hyperparameter configuration for the machine learning model with the overall best result.

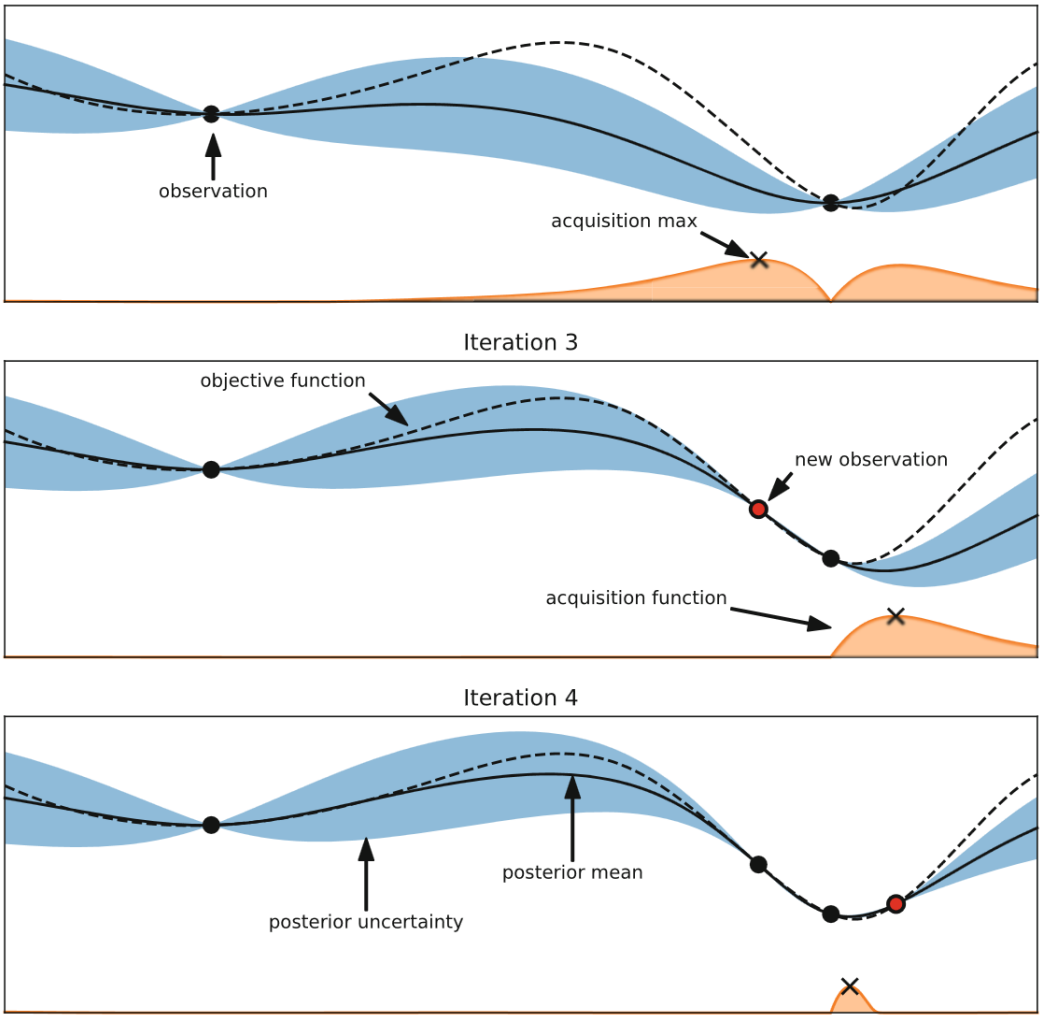


Figure 2.4: Schematic iteration steps of the bayesian optimization. The maximum of the acquisition function determines the next function evaluation (red dot in the middle). The goal is to find the minimum of the dashed line. The blue band is the uncertainty of the function. Taken from [15].

2.2.4 Other Techniques

There are also other techniques for finding the best hyperparameters. Multi-fidelity optimization [15] aims to probe the learning of model on a task with reduced complexity such as a subset of the data or less 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 performs the best ones. This can be done iteratively like it is done in *successive halving* for hyperparameter optimization [24]. The algorithm is very simple. It starts to evaluate all different combinations with very small budget. The best half of the candidates are then evaluated in the next iteration with double budget and so on until only one combination is left. In [25], 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 then evaluated with higher budget. Also neural networks can be used for the optimization which was done by the authors in [26]. Also, covariance matrix adaptation evolution strategy was implemented as an alternative to bayesian optimization in [27].

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 $f : \Omega \to R$ 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

$$x_{l,i} = i * h_l,\ i = 0, ..., 2^l, \tag{2.7}$$

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

$$\varphi_{l,i} : [0,1] \to \mathbb{R}. \tag{2.8}$$

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

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

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:

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

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

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

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

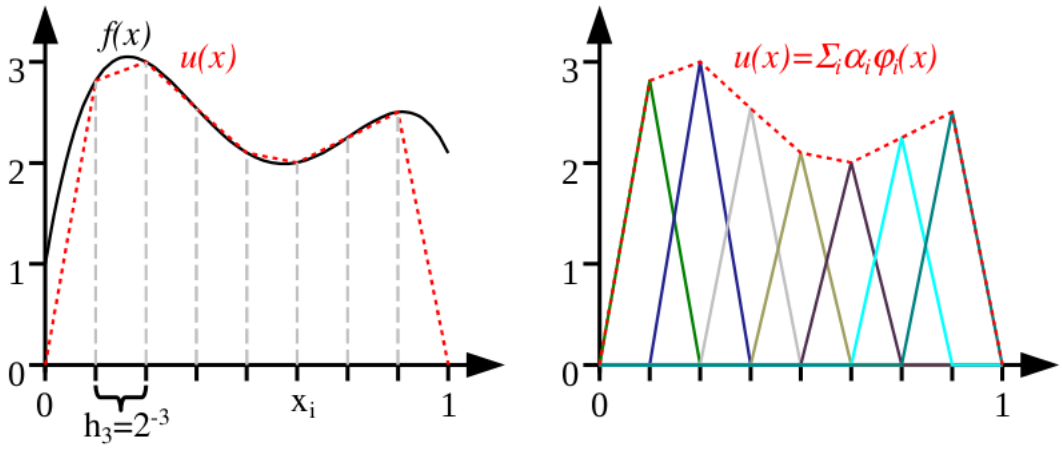


Figure 2.5: Interpolation of the function f (black line) by its interpolant u (red, dashed) in the nodal basis. Level of the grid is 3 and hat functions are used. Taken from [28].

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 $I_l^h = \{i \in \mathbb{N} | 1 \leq i \leq i \leq s^l - 1, i \text{ odd}\}$. The hierarchical subspaces are then

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

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

$$V_l = \bigoplus_{i \leq l} W_i. \tag{2.13}$$

An example can be seen in Figure 2.6.

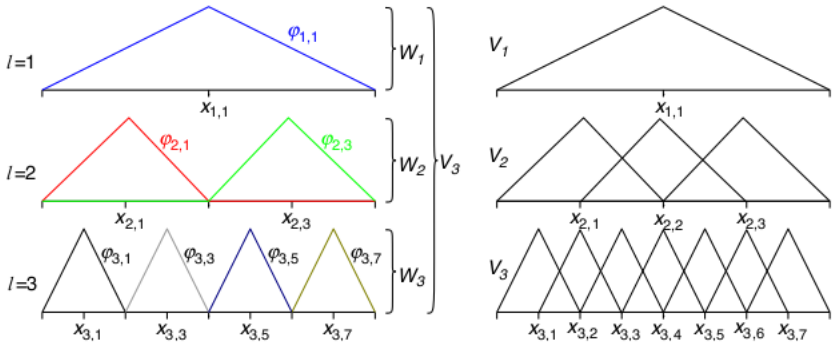


Figure 2.6: Hierarchical subspaces up to level 3 on the left. On the right, nodal spaces up to level 3. The combination of W_1 up to W_3 is the same space as V_3. Taken from [28].

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

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

An example can be seen in Figure 2.7.

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 $\vec{l} = (l_1, ..., l_d) \in \mathbb{N}_0^d$. The index set is then

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

and the subspaces

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

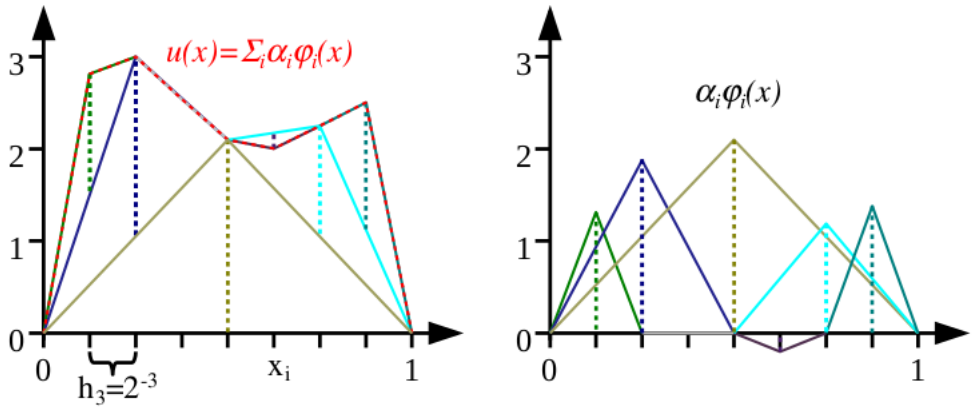


Figure 2.7: Interpolation of the function f (black line) by its interpolant u (red, dashed) in the hierarchical basis. Level of the grid is 3 and hat functions are used. Taken from [28].

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

$$V_n = \bigoplus_{|\vec{l}|_\infty \leq n} W_l \tag{2.17}$$

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

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

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.

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

$$V_{\vec{l}} = \bigoplus_{\vec{m}=0}^{\vec{l}} W_{\vec{m}} \tag{2.19}$$

with $W_{\vec{l}} = \text{span}\left\{\varphi_{\vec{l},\vec{i}} | \vec{i} \in I_{\vec{l}}\right\}$, $I_{\vec{l}} = I_{l_1} \times ... \times I_{l_d}$ holds for the basis with hat functions can be found in [30].

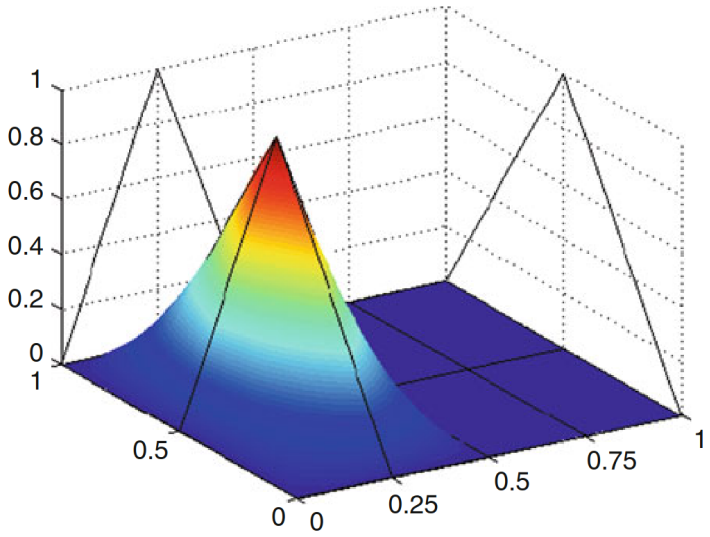


Figure 2.8: Example of a basis function in two dimensions. It is constructed with the tensor product of two 1d hat functions. Taken from [29].

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 [31, 32] by reducing this number. The first technique to achieve this is by just leaving out subspaces. The resulting sparse function space is given by

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

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

An interpolant u_n of a function f is then constructed by

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

where the $\alpha_{\vec{l},\vec{i}}$ are the coefficients of the basis functions [33].

A second approach for sparse grids exists. The so-called *combination technique* [34] 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.

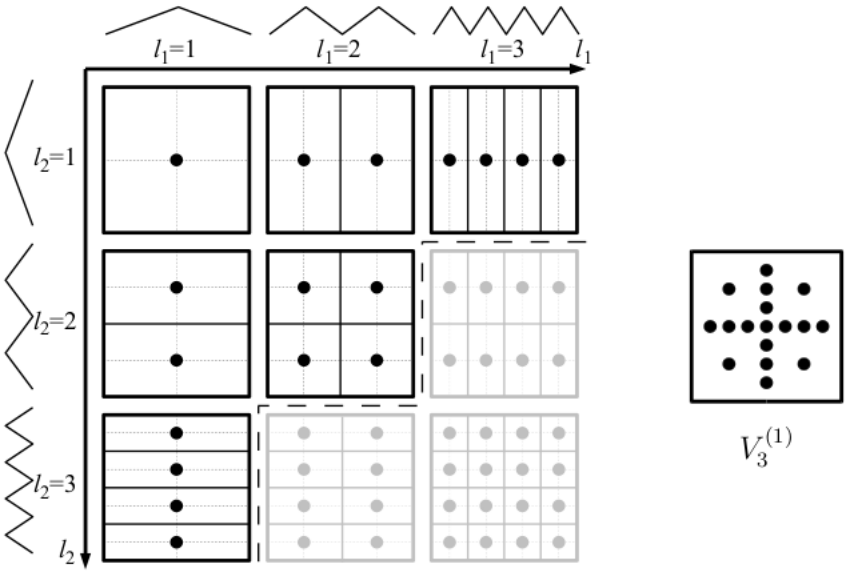


Figure 2.9: Two dimensional example of a sparse grid with $n = 3$. Left, the subspaces $W_{\vec{l}}$ can be seen and on the right is the resulting sparse grid. Taken from [29].

The combined solution can be computed with

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

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

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

with $I_{l,q} = \left\{ \vec{l} \in \mathbb{N}_0^d : \left|\left|\vec{l}\right|\right|_1 = l + d - 1 - q \right\}$ [35]. 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 [36], more grid points are added in the dimensions of higher relevance.

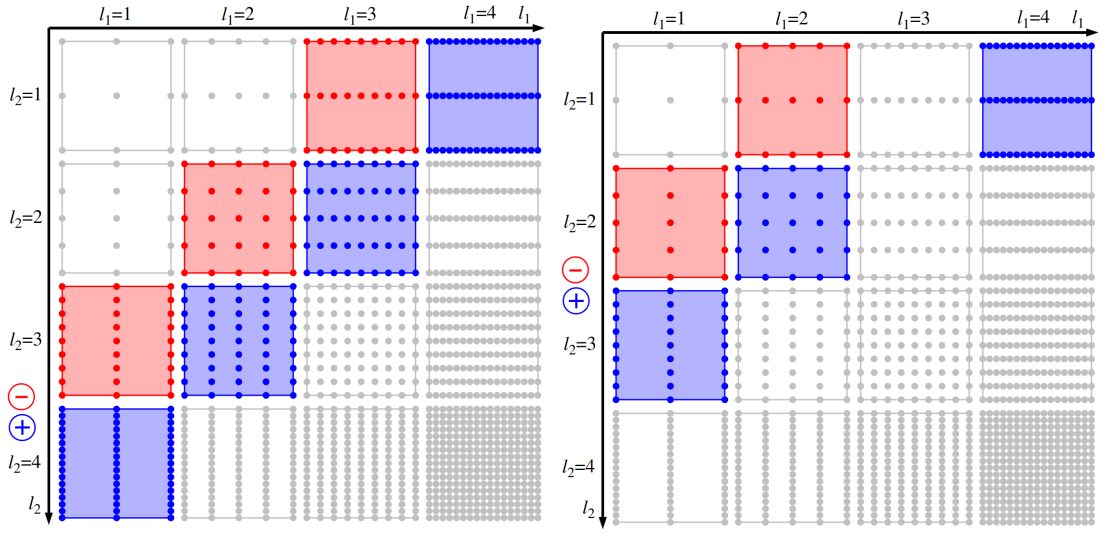


Figure 2.10: Example of the 2-dimensional combination technique. Here the blue regular grids are added and the red ones are subtracted. On the left, the normal combination technique can be seen and on the right is an dimension-adaptive version. Taken from [28].

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 [35] 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 points and interpolation accuracy [28].

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	$\mathcal{O}\left(h_n^2\right)$	$\mathcal{O}\left(2^{nd}\right)$
Sparse grid	$\mathcal{O}\left(h_n^{-1}\left(\log h_n^{-1}\right)^{d-1}\right)$	$\mathcal{O}\left(h_n^2\left(\log h_n^{-1}\right)^{d-1}\right)$
Combination technique	$\mathcal{O}\left(d\left(\log h_n^{-1}\right)^{d-1}\right) \times \mathcal{O}\left(h_n^{-1}\right)$	$\mathcal{O}\left(h_n^2\left(\log h_n^{-1}\right)^{d-1}\right)$

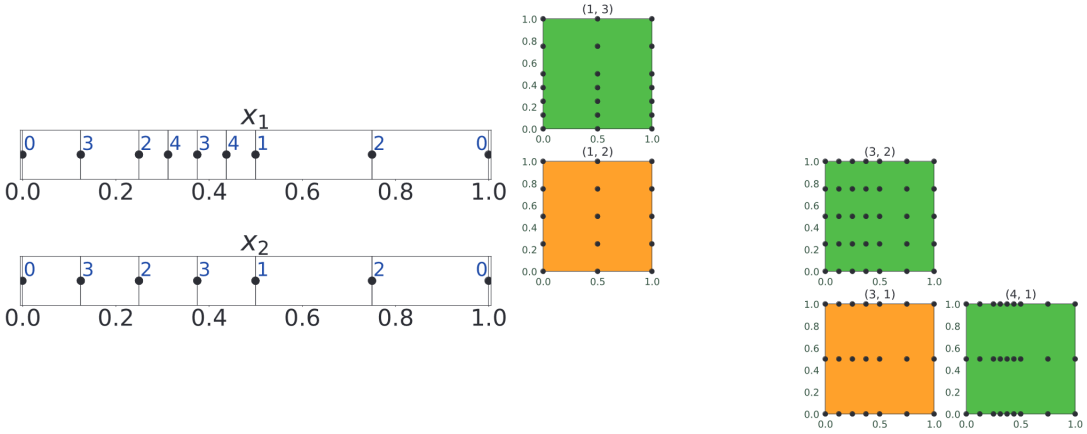


Figure 2.11: Example of the spatially adaptive combination technique in two dimensions. Taken from [35].

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 [28, 32, 37] 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 [30]. 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

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

with $\chi_{[0,1[}$ being the characteristic function of the half-open unit interval [38]. 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 $\xi = (\xi_0, ..., \xi_{m+p})$ be an increasing sequence of real numbers called *knot sequence*. For $k = 0, ..., m-1$, the non-uniform B-spline is defined by

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

This definition and the proof that the hierarchical splitting also holds for using the B-splines for restricted functions can be found in [30].

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 gradient-free method, respectively. In the following, algorithms of both types are presented [30].

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

$$x_{k+1} = x_k + \delta_k d_k \tag{2.26}$$

where δ_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 $d_k \propto -\left(\nabla_x^2 f(x_k)\right)^{-1} \nabla_x f(x_k)$. 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 [30]. There, the optimal point has to be found while constraining another function g with $x_{opt} = argmin f(x)$, such that $g(x) \geq 0$.

One application of the optimization with sparse grids is presented in [39]. 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 [40]. One application is dealing with induction motor parameter estimation [41]. There, the search space is discretized using the hyperbolic cross points (HPC). In one dimension, the points of level k are defined with

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

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 $k = 5$.

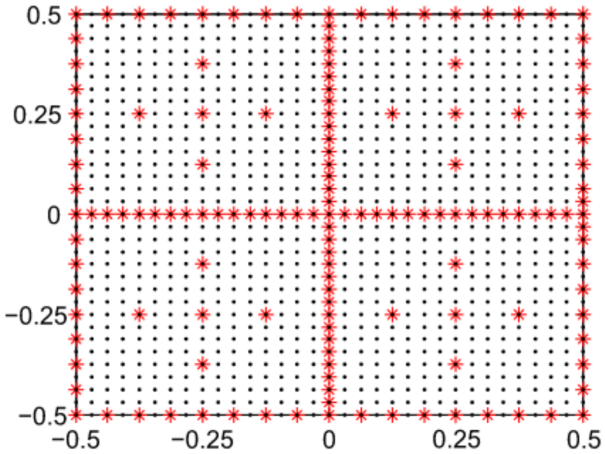


Figure 2.12: HPC of level $k = 5$ (red stars) and full grid (black dots). A full grid would have $33^2 = 1089$ and this grid has 147 points. Taken from [41].

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 $\mathcal{O}\left(N^{-2} \cdot (\log N)^{d-1}\right)$ 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 [42] 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 [40, 41].

Another method for optimization is presented in [43]. 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 [44] 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 [45] 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.

3 Hyperparameter Optimization with Sparse Grids

3.1 Methodology

3.1.1 Adaptive Grid Search with Sparse Grids

- Evaluating neural networks as function with input of the hyperparameters
- General goals

3.1.2 Implementation

The implementation was done in Python and two main class 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 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 [46]. 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 4 concrete implementation, one for each of grid-, random search, bayesian optimization and the sparse grid search. In all cases, a dataset object has to be given. Additionally, the machine learning model as a function depending on the hyper parameters is required. For the sparse grid search, the functionality of the *SG++* [47] 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 that has to be defined. This is a dictionary with information about the names and the types of the 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.

```
def to_standard(lower, upper, value):
  return (value - lower) / (upper - lower)

def from_standard(lower, upper, value):
  return value * (upper - lower) + lower

def to_standard_log(lower, upper, value):
  a = math.log10(upper / lower)
  return math.log10(value / lower) / a

def from_standard_log(lower, upper, value):
  a = math.log10(upper/lower)
  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.

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 [47]:

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 [48]

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

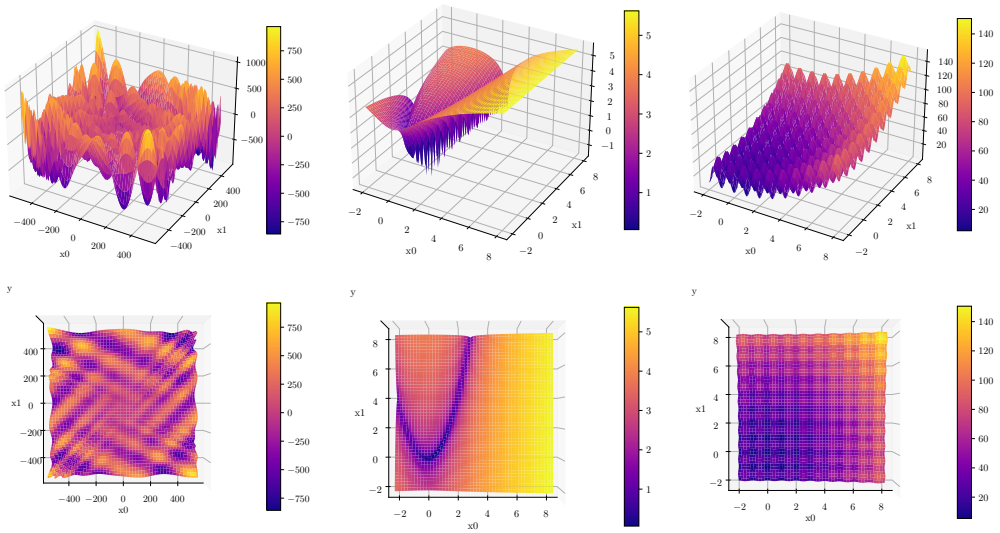
The second function (Rosenbrock) [49] is calculated with

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

and the third one (Rastrigin) [49] is defined with

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

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



(a) Eggholder. (b) $log_{10}(f)$ of Rosenbrock. (c) Rastrigin.

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 [30] is evaluated. In each iteration, the grid point $x_{l,i}$ that minimizes

the product

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

is refined. In this case, $r_{l,i} = |\{(l', i') \in K | f(x_{l',i'}) \leq f(x_{l,i})\}| \in \{1, ..., |K|\}$ 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 γ. 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

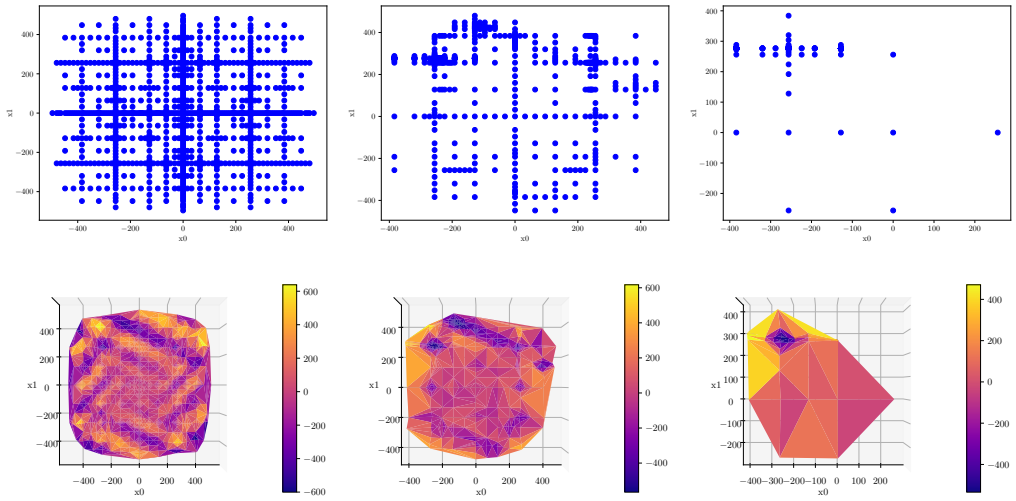
In the following, the behavior of the sparse grid generation is analyzed with the help of the three test functions. In each case, 3 different values for the adaptivity parameter $\gamma \in \{0.0, 0.5, 1.0\}$ are used and the resulting sparse grid with the triangulated function values is plotted. The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

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.

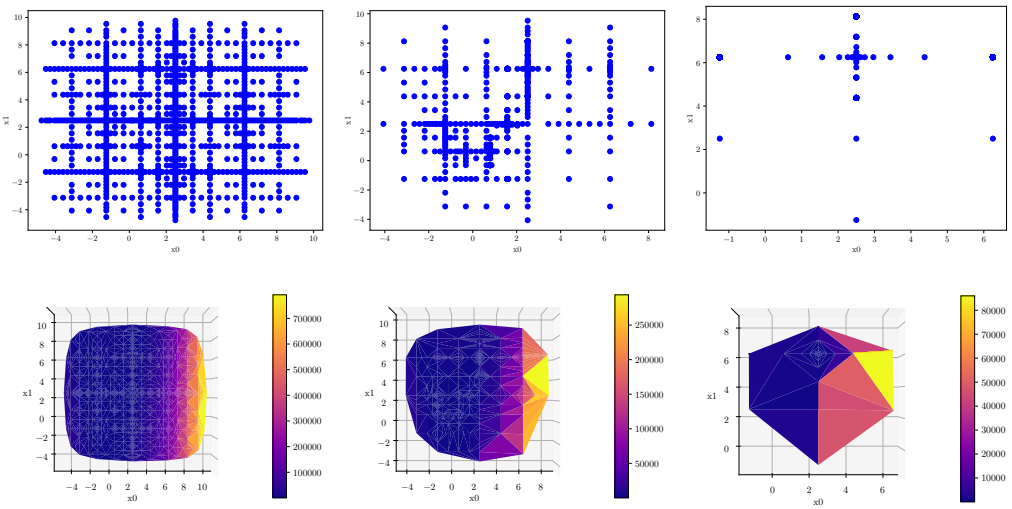
Note that in all three cases, the exact same number of grid points are evaluated. In this case it is very hard for the sparse grid to find the real optimum because it is located at the border of the domain and it does not use grid points at the border. Also, the oscillatory nature of the function makes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



(a) $\gamma = 1.0$ (b) $\gamma = 0.5$ (c) $\gamma = 0.0$

Figure 3.2: Sparse grid generation depending on the adaptivity parameter γ 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.

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.



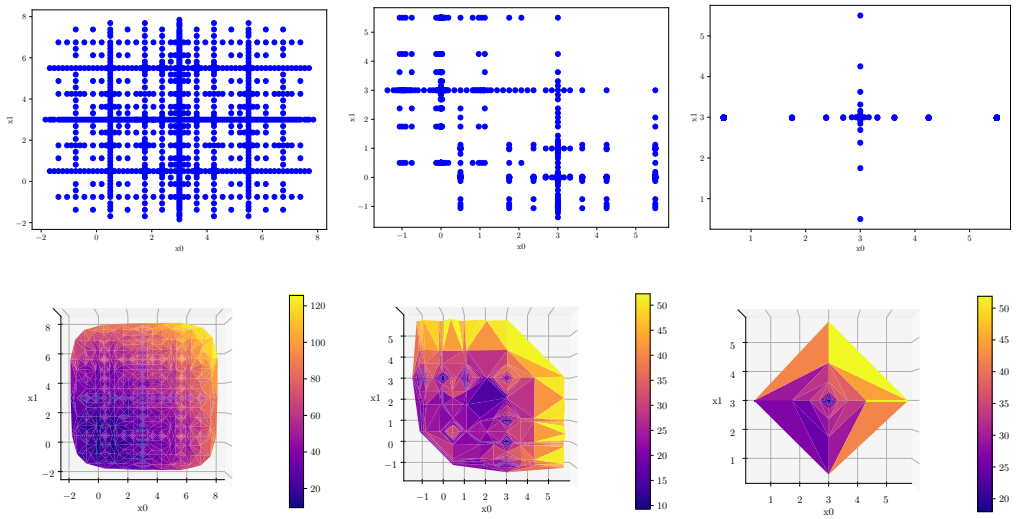
(a) $\gamma = 1.0$ (b) $\gamma = 0.5$ (c) $\gamma = 0.0$

Figure 3.3: Sparse grid generation depending on the adaptivity parameter γ 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 γ 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 γ, 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).



(a) $\gamma = 1.0$ (b) $\gamma = 0.5$ (c) $\gamma = 0.0$

Figure 3.4: Sparse grid generation depending on the adaptivity parameter γ 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 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

$$Error = f(x^*_{opt}) - f(x_{opt}) \tag{3.5}$$

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.

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 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

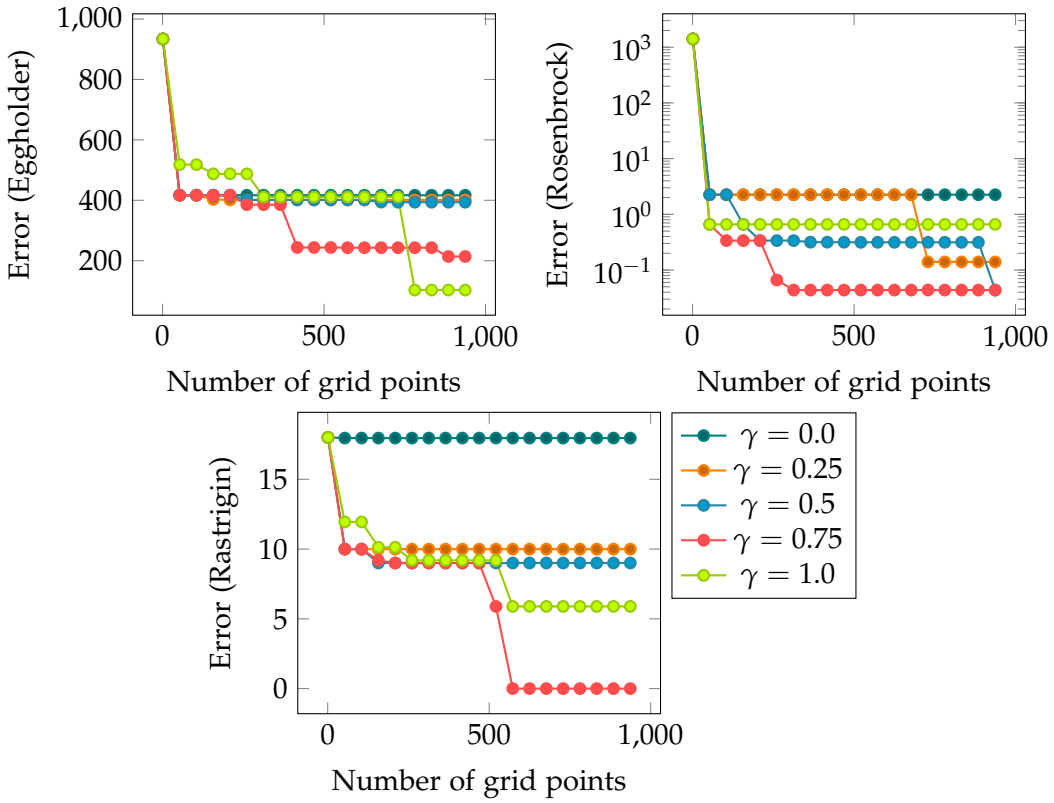


Figure 3.5: Influence of the adaptivity parameter on the error (difference to the actual optimal value) of the optimum found by the sparse grid.

between optimum found by algorithm and actual optimal function value) behaves with higher number of grid points of the underlying sparse grid. As a local optimization algorithm, the 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.

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.

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 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 me of them are really 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

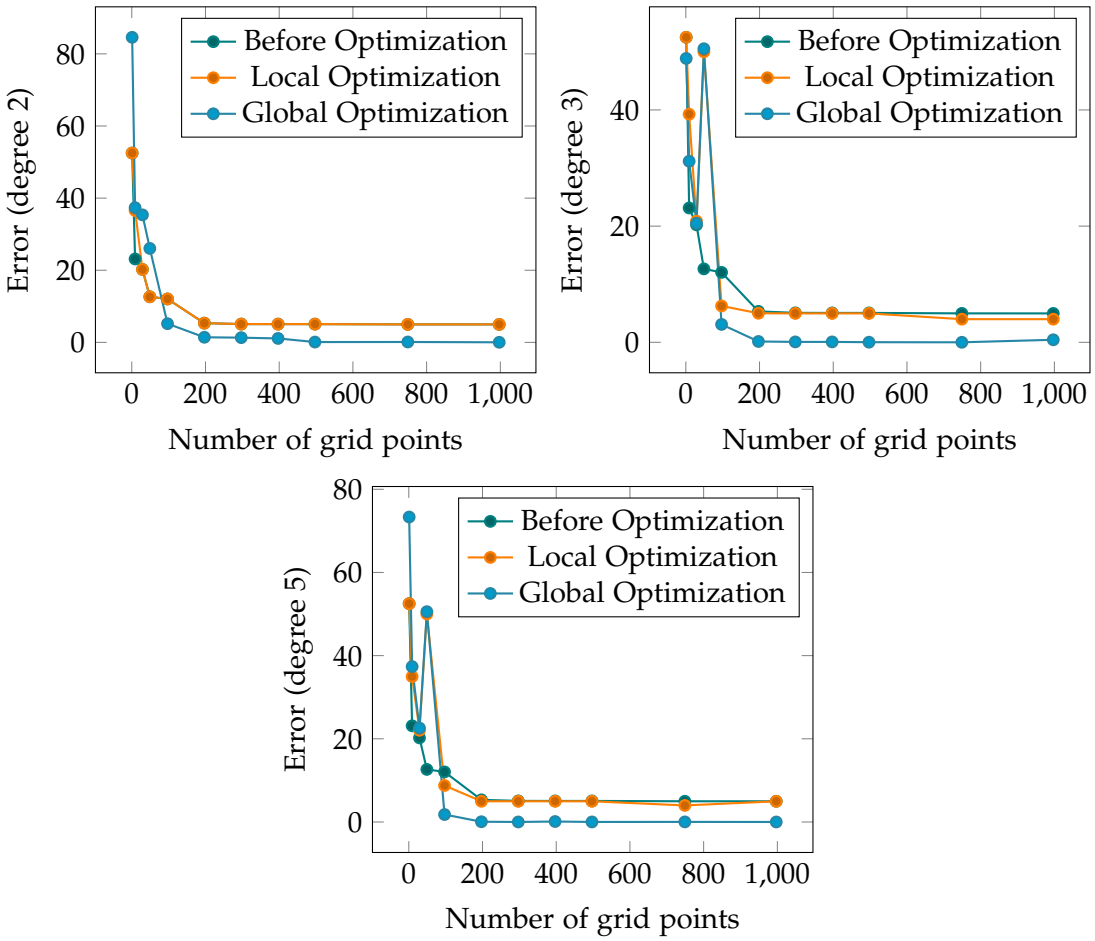


Figure 3.6: Optimization error for different algorithms depending on the number of grid points in two dimensions. The plots show the results with degree 2 (top left), degree 3 (top right) and degree 5 (bottom). The Rastrigin function was used.

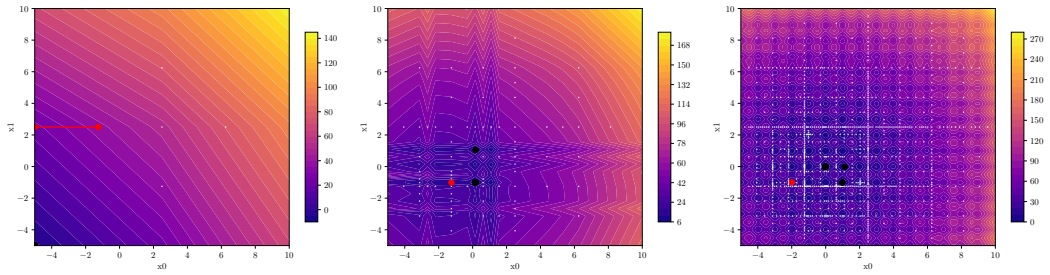


Figure 3.7: Resulting optimal points from local (red points) and global (black points) optimization algorithm. In the background, the contour of the interpolated function is depicted with the corresponding sparse grid points in white. The left one has 5 grid points, the one in the center 77, and the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

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	**(-0.014,-0.002)**	0.151	**0.042**

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.

The input of the previous experiments were all just in two dimensions. 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.

The results show that with increasing dimensionality of the problem, more grid

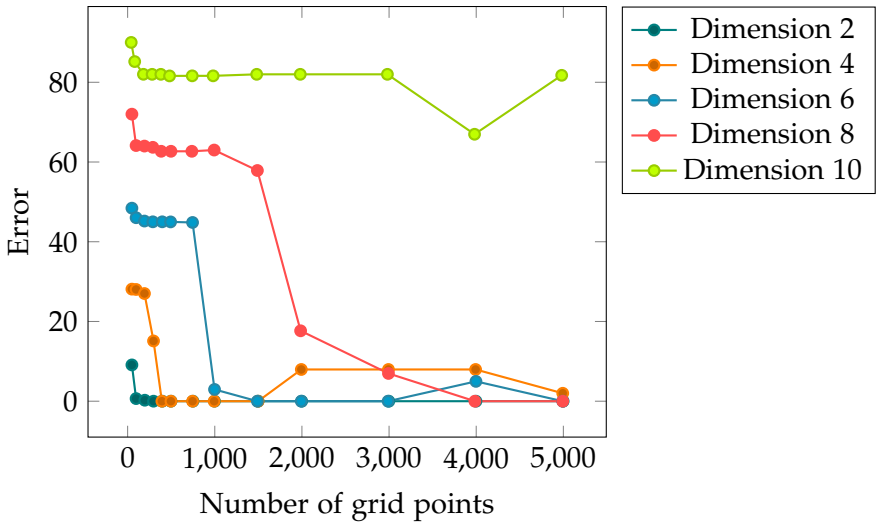


Figure 3.8: Error of the optimization depending on the number of grid points used for different dimensions of the Rastrigini function.

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 [46]. 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 is 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

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y^i - y^i_{pred}}{y^i}\right| \tag{3.6}$$

where y^i is the target value and y^i_{pred} 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}(learning rate) can be seen in Figure 3.9.

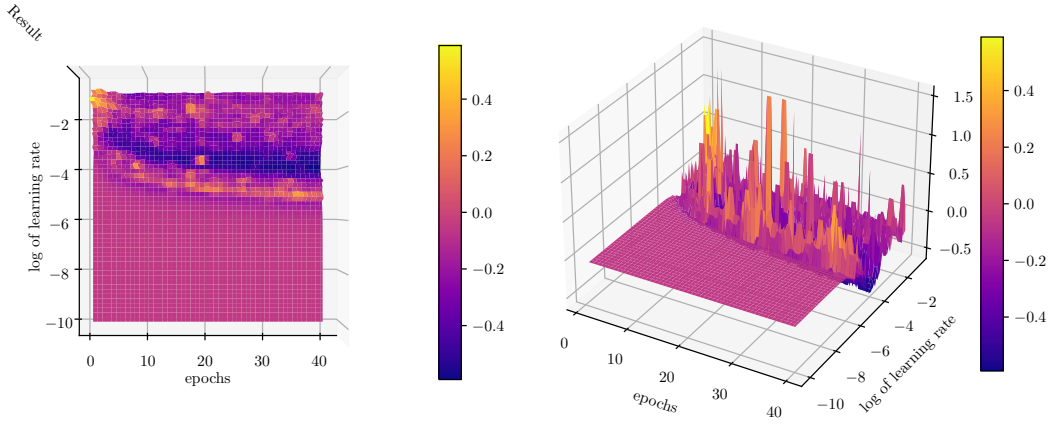


Figure 3.9: 2-layer (with 30 neurons each) fully connected network evaluation on the diamonds dataset depending on the number of epochs and learning rate of the Adam optimizer.

In this plot, for each of epochs 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 slow 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 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 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.

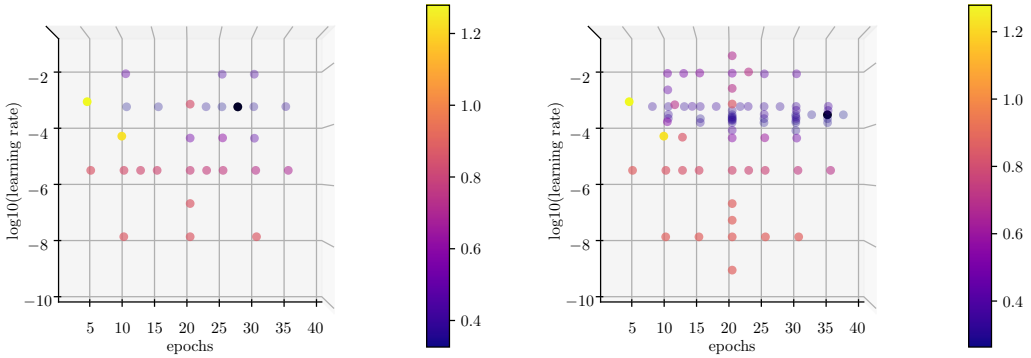
In the first case with $\gamma = 0.85$, more grid points are generated in the upper half of the domain. A similar behavior can be observed for a higher number of grid points (3.10a, right). Especially in regions where the learning rate is about 0.0002 to 0.0003 (scale marked at 0.7), many grid points 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 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.

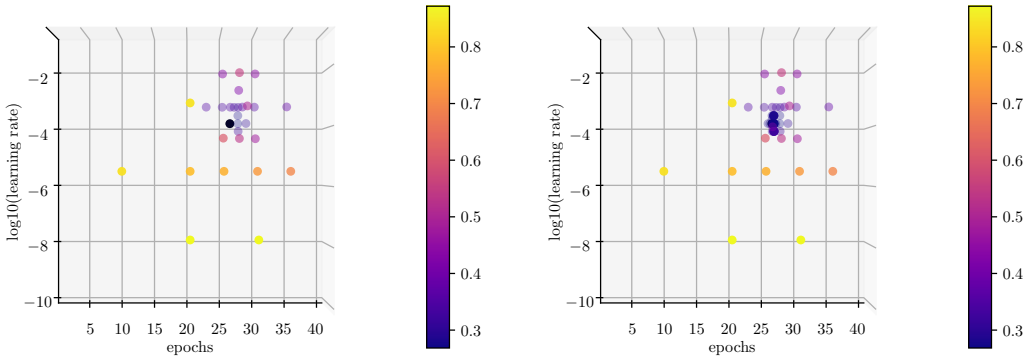
However, as shown in Table 3.3, the best value is found with the configuration $\gamma = 0.85$ and number of grid points of 100. This means that lower adaptivity leads to better results.

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.

γ	N	Epochs	Learning rate	Result
0.85	29	27	0.00056	0.32599
0.85	77	35	0.00029	**0.26339**
0.5	29	26	0.00015	0.26837
0.5	77	26	0.00015	0.26837



(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.

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.

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	$(27.81, 5.623 * 10^{-4})$	**0.326**	0.326
49	$(20.5, 2.129 * 10^{-4})$	**0.267**	$(21.72, 2.094 * 10^{-2})$	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

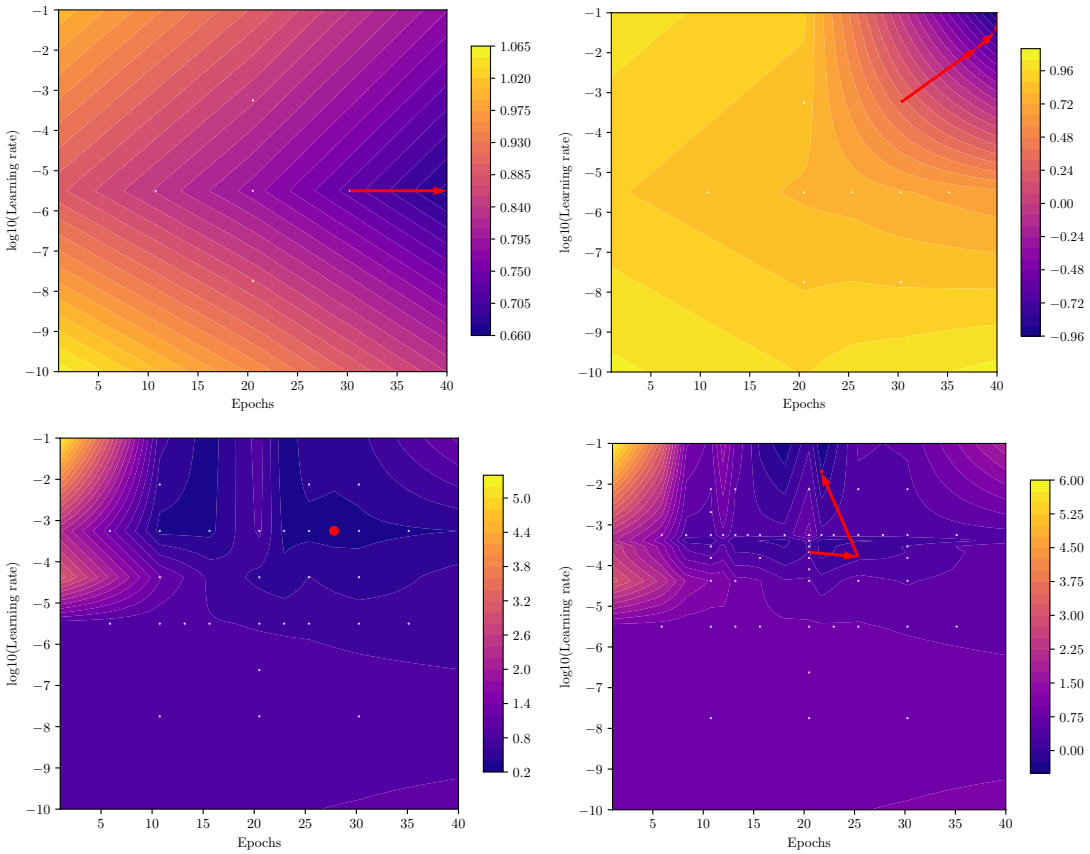


Figure 3.11: Optimization steps of gradient descent algorithm for 5 (top left), 9 (top right), 29 (bottom left) and 49 (bottom right) grid points. In the background of each plot, the contour of the interpolated function is shown. The function evaluated is the same as depicted in Figure 3.9.

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 becing 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.

For both cases, the solution found by the sparse grid without 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.

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 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

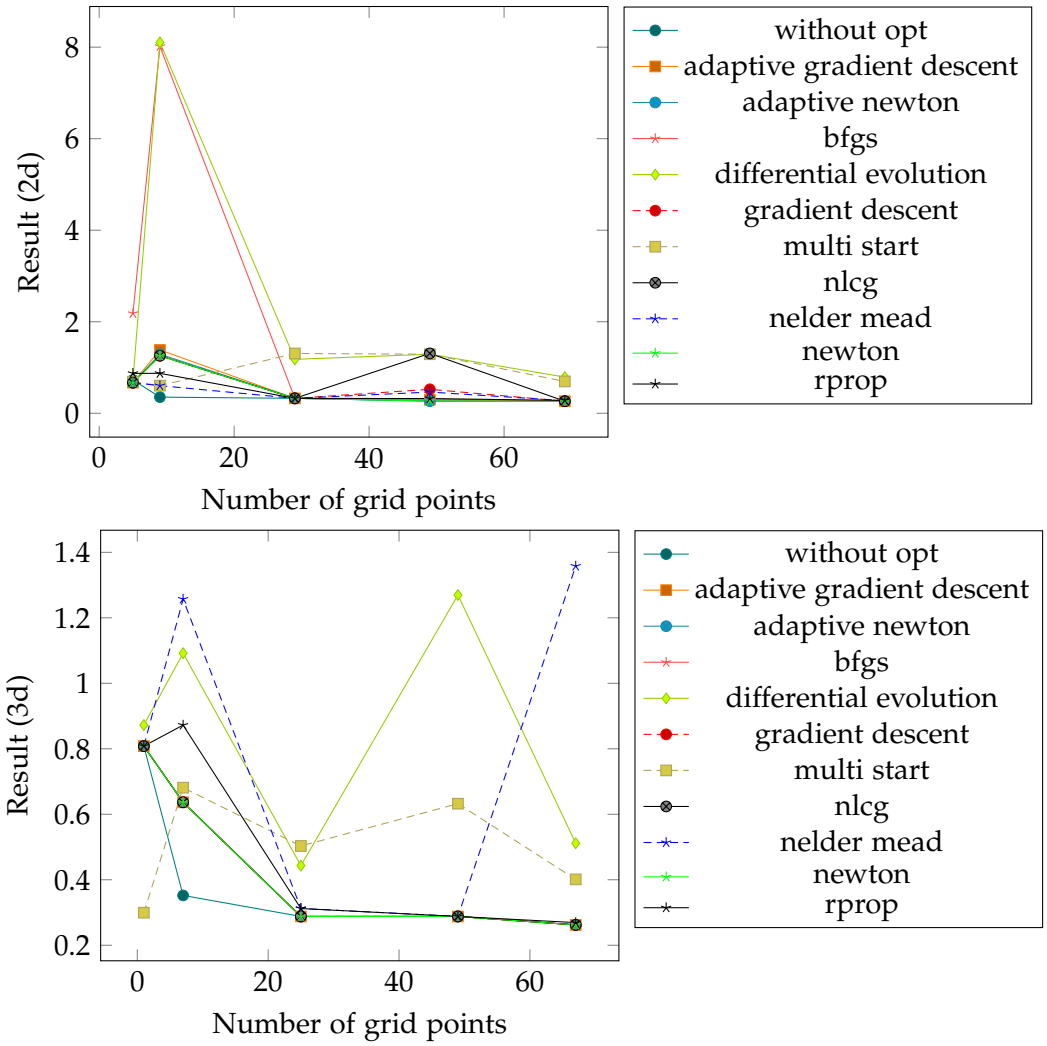


Figure 3.12: Comparison of optimization algorithms on the sparse grid with increasing number of grid points in two and three dimensions. Epochs, learning rate, and the batch size of a two-layer neural network on the diamonds dataset for regression are optimized. Result is the mean absolute percentage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

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 $budget - 2$ because we have to evaluate the point of the local and global optimization for comparison.

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 over the datasets.

Table 3.5: Overview over the datasets used for the first comparison of the optimization algorithms.

	Number of features		
Dataset	numeric	categorical	Number of instances
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 [50]. 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 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 adaptivity value 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 logarithmic from 10^{-10} and 10^{-1}. The resulting performance can be seen in Figure 3.13.

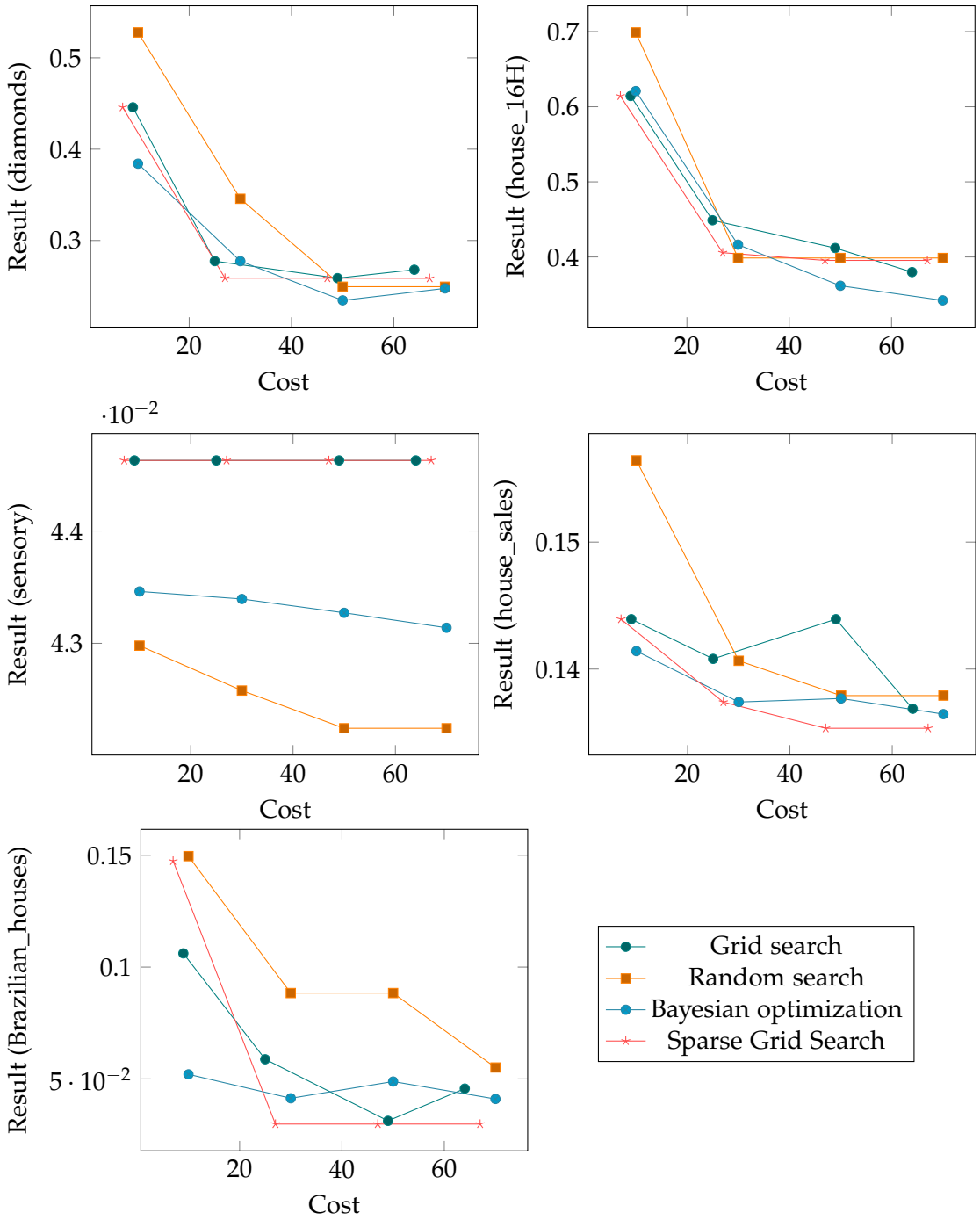


Figure 3.13: Comparison of grid-, random search, bayesian optimization and sparse grid search for the datasets shown in Table 3.5. Two hyperparameters, epochs and learning rate of the neural network's optimizer, were optimized.

4 Conclusion and Outlook

List of Figures

List of Tables

Bibliography

[1] H. Wang, Z. Lei, X. Zhang, B. Zhou, and J. Peng, "Machine learning basics," *Deep learning*, pp. 98–164, 2016.

[2] B. Mahesh, "Machine learning algorithms-a review," *International Journal of Science and Research (IJSR).[Internet]*, vol. 9, pp. 381–386, 2020.

[3] T. O. Ayodele, "Types of machine learning algorithms," *New advances in machine learning*, vol. 3, pp. 19–48, 2010.

[4] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.

[5] 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.

[6] L. Rokach and O. Maimon, "Decision trees," *Data mining and knowledge discovery handbook*, pp. 165–192, 2005.

[7] C. M. Bishop, "Neural networks and their applications," *Review of scientific instruments*, vol. 65, no. 6, pp. 1803–1832, 1994.

[8] 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.

[9] L. R. Medsker and L. Jain, "Recurrent neural networks," *Design and Applications*, vol. 5, pp. 64–67, 2001.

[10] 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.

[11] 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.

[12] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[13] 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.

[14] 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.

[15] M. Feurer and F. Hutter, "Hyperparameter optimization," *Automated machine learning: Methods, systems, challenges*, pp. 3–33, 2019.

[16] 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.

[17] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[18] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization.," *Journal of machine learning research*, vol. 13, no. 2, 2012.

[19] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, 2012.

[20] R. Andonie, "Hyperparameter optimization in learning systems," *Journal of Membrane Computing*, vol. 1, no. 4, pp. 279–291, 2019.

[21] 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.

[22] 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.

[23] J. Wilson, F. Hutter, and M. Deisenroth, "Maximizing acquisition functions for bayesian optimization," *Advances in neural information processing systems*, vol. 31, 2018.

[24] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *Artificial intelligence and statistics*, PMLR, 2016, pp. 240–248.

[25] 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.

[26] 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.

[27] I. Loshchilov and F. Hutter, "Cma-es for hyperparameter optimization of deep neural networks," *arXiv preprint arXiv:1604.07269*, 2016.

[28] D. M. Pflüger, "Spatially adaptive sparse grids for high-dimensional problems," Ph.D. dissertation, Technische Universität München, 2010.

[29] J. Garcke, "Sparse grids in a nutshell," in *Sparse grids and applications*, Springer, 2013, pp. 57–80.

[30] J. Valentin, "B-splines for sparse grids: Algorithms and application to higher-dimensional optimization," *arXiv preprint arXiv:1910.05379*, 2019.

[31] 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.

[32] H.-J. Bungartz and M. Griebel, "Sparse grids," *Acta numerica*, vol. 13, pp. 147–269, 2004.

[33] 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.

[34] M. Griebel, M. Schneider, and C. Zenger, "A combination technique for the solution of sparse grid problems," 1990.

[35] 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.

[36] M. Hegland, "Adaptive sparse grids," *Anziam Journal*, vol. 44, pp. C335–C353, 2002.

[37] H.-J. Bungartz, "Finite elements of higher order on sparse grids," Ph.D. dissertation, Technische Universität München, 1998.

[38] K. Höllig and J. Hörner, *Approximation and modeling with B-splines*. SIAM, 2013.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] S. Chen and X. Wang, "A derivative-free optimization algorithm using sparse grid integration," 2013.

[45] S. Sankaran, "Stochastic optimization using a sparse grid collocation scheme," *Probabilistic engineering mechanics*, vol. 24, no. 3, pp. 382–396, 2009.

[46] 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.

[47] 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.

[48] D. Whitley, S. Rana, J. Dzubera, and K. E. Mathias, "Evaluating evolutionary algorithms," *Artificial intelligence*, vol. 85, no. 1-2, pp. 245–276, 1996.

[49] X.-S. Yang, *Engineering optimization: an introduction with metaheuristic applications*. John Wiley & Sons, 2010.

[50] 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.