DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation of a Deep Sparse Grid Layer in PyTorch

Simon Blöchinger

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation of a Deep Sparse Grid Layer in PyTorch

Implementierung mehrschichtiger dünner Gitter in PyTorch

Author:	Simon Blöchinger
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Dr. Felix Dietrich
Submission Date:	May 12, 2021

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

May 12, 2021 Simon Blöchinger

Acknowledgements

I want to acknowledge

- … my advisor, Dr. Felix Dietrich, for his helpful feedback and support throughout the writing of this thesis,
- … Zhen Zhang, who provided helpful insights about the codebase,
- … my friends and family for proofreading this thesis and providing support during the creation process.

Abstract

Sparse grids are useful for function approximation in high dimensions, because they reduce the impact of the "curse of dimensionality", as the number of grid points does not grow exponentially with the number of dimensions. This gives access to function approximation in higher dimensions than possible with full grids. Sparse grids represent functions as a linear combination of nonlinear basis functions.

A neural network can also represent a function as a linear combination of nonlinear basis functions. In contrast to sparse grids, however, the neural network learns the linear combination.

A combination of sparse grids and neural networks could lead to a more efficient use of resources and faster training of the neural network. Since there currently is a lack of frameworks that allow using sparse grids inside of neural networks, an implementation of a deep sparse grid layer is introduced in this thesis. The implementation uses the Python machine learning library PyTorch. With this implementation, it is possible for future researchers to start evaluating the advantages and disadvantages of sparse grids inside neural networks quicker. It provides a customizable sparse grid layer, which uses a sparse grid built with the combination technique, and can utilize PyTorch's parallel tensor computation on compatible GPUs.

Zusammenfassung

Dünne Gitter sind nützlich, um Funktionen in hohen Dimensionen zu approximieren, weil sie die Auswirkungen des „Fluchs der Dimensionalität" reduzieren, da die Anzahl der Gitterpunkte nicht exponentiell mit der Dimension ansteigt. Die Verwendung von dünnen Gittern erlaubt demnach die Approximation in höheren Dimensionen als mit vollen Gittern. Funktionen werden in dünnen und vollen Gittern als lineare Kombination von nichtlinearen Basisfunktionen dargestellt.

Ein neuronales Netz repräsentiert Funktionen auch mit einer linearen Kombination von nichtlinearen Basisfunktionen. Diese Kombination wird allerdings von dem neuronalen Netz gelernt.

Eine Verbindung zwischen dünnen Gittern und neuronalen Netzen könnte zu einer effektiveren Nutzung von Rechenressourcen und damit einer schnelleren Lerngeschwindigkeit des neuronalen Netzes führen. Da es momentan an Programmiergerüsten, die die Verwendung von dünnen Gittern innerhalb von neuralen Netzen erlauben, mangelt, führt diese Bachelorarbeit eine Implementierung mehrschichtiger dünner Gitter in der Python Programmbibliothek PyTorch ein. Diese Implementierung erlaubt zukünftigen Forschern, schneller und einfacher mit dem Studieren der Vor- und Nachteile der Verwendung von dünnen Gittern in neuronalen Netzen anzufangen. Die Implementierung stellt eine Schicht für ein neurales Netz zur Verfügung, die ein dünnes Gitter, das mit der Kombinationstechnik geschaffen wurde, verwendet. Parallele Berechnung auf Grafikprozessoren kann von der Schicht verwendet werden.

Contents

List of Notations

Notation	Description
$\mathbb{N}$	The natural numbers.
$\mathbb{N}_0$	The natural numbers including 0.
$\vec{v}$	The vector v.
v_i	The i-th element of the vector v.
$\vec{1}$	The vector consisting of ones.
$\mathcal{O}$	The big O notation.
l	The discretization level.
$\vec{l}$	The multi-dimensional vector of discretization levels.
Ω_l	The one-dimensional grid of level l.
$\bar{\Omega}$	The domain of the one-dimensional grid.
Ω_l^d	The d-dimensional grid of level l.
$\bar{\Omega}^d$	The domain of the d-dimensional grid.
h_l	The mesh width of a grid of level l.
$\vec{h}_l$	The multi-dimensional mesh width of level l.
$x_{l,i}$	The point with the index i in a grid of level l.
$\vec{x}_{\vec{l},\vec{i}}$	The grid point with the vector index $\vec{i}$.
$\Phi(x)$	The hat function.
$\Phi_{l,i}(x)$	The hat function on the grid point $x_{l,i}$.
$\Phi_{\vec{l},\vec{i}}(x)$	The hat function on the grid point $x_{\vec{l},\vec{i}}$.
V_l	The function space of a basis of level l.
$V_{\vec{l}}$	The function space of a basis with the discretization level vector $\vec{l}$.
W_l	The hierarchical increment space of level l.
$W_{\vec{l}}$	The hierarchical increment space with the discretization level vector $\vec{l}$.
I_l	The index set for the increment space W_l.
$I_{\vec{l}}$	The index set for the increment space $W_{\vec{l}}$.
$\hat{V}_n$	The sparse grid space of level n.
$u_l(x)$	The interpolation function of a basis of level l.
$u_{\vec{l}}(\vec{x})$	The interpolation function of a basis with the discretization level vector $\vec{l}$.
$\hat{u}_n(x)$	The interpolation function of a sparse grid of level n.
$\lvert\vec{l}\rvert_1$	The L-1 norm of the vector $\vec{l}$.
$\lvert\vec{l}\rvert_\infty$	The uniform norm of the vector $\vec{l}$.

1 Introduction

Nowadays, neural networks are used to solve a multitude of tasks that consist of learning, generalizing or clustering data. They can for example be used to identify emotions in pictures of faces [1] or interpret medical images [2]. They can also be used for dimension reduction on a function or regression [3].

Full grids can be used to approximate a function when a full sampling of the space is necessary. However, full grid approximation does not work well for functions in higher dimensions, since the number of points needed in a full grid grows exponentially with the dimension. Sparse grids are grids where most grid points are left empty. This results in the number of sparse grid points not growing exponentially with the number of dimensions. This gives access to function approximation in higher dimensions than normally possible. The intelligent layout of the sparse grid points (see Figure 1.1) allows for function approximation in higher dimensions with adequate approximation accuracy. Sparse grids are typically used when the use of full grids is not feasible anymore, but the space still has to be sampled completely, like solving partial differential equations or interpolation and approximation in medium dimensions [4]. Sparse grids represent functions as a linear combination of nonlinear basis functions.

A neural network can also represent a function as a linear combination of nonlinear basis functions. In contrast to sparse grids, however, the neural network needs to learn this combination. When training to represent a function, the neural network adjusts its weights to achieve a result with minimal error.

Combining sparse grids and neural networks could lead to faster training of neural networks. However, there currently is a lack of frameworks that allow using sparse grids inside of neural networks. This thesis introduces the implementation of a deep sparse grid layer for the Python machine learning library PyTorch. With this implementation, it will be possible for future researchers to start evaluating the advantages and disadvantages of sparse grids inside of neural networks quicker.

The mathematical background of full grids and sparse grids is explained in Sections 2.1 and 2.2, followed by an introduction to neural networks in Section 2.3. The library PyTorch, which is the Python machine learning library used to implement the deep sparse grid layer, is introduced in Section 2.4. Section 3 discusses the implementation of the deep sparse grid layer. The underlying codebase that is the basis for the sparse grid layer is introduced in Section 3.1. In Section 3.2 the representation of sparse grids in the sparse grid layer is explained. How the implemented layer can be customized and which features are implemented is discussed in Section 3.3. The algorithm used to generate sparse and full grids in the sparse grid layer is introduced in Section 3.4. In Section 3.5 the architecture of the implementation is presented. Unit tests for the sparse grid layer are introduced in Section 3.6. Finally, the capabilities of the deep sparse grid layer implementation for utilizing parallel GPU processing is evaluated in Section 3.8.

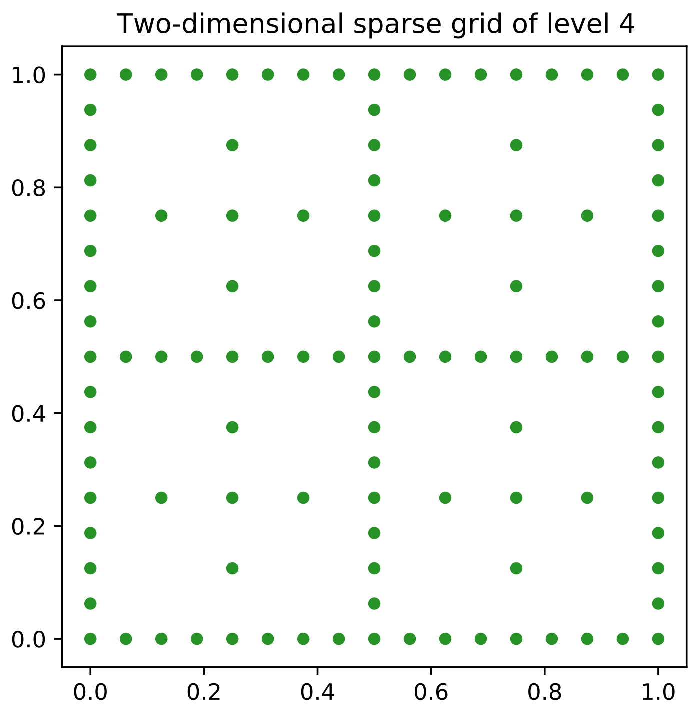


Figure 1.1: A two-dimensional sparse grid of level 4 with boundary points in the domain $[0, 1]^2$.

2 Theoretical Background

The mathematical background of full grids and sparse grids is given in Sections 2.1 and 2.2. Afterwards, neural networks are introduced in Section 2.3. The Python library PyTorch, that is the basis for the implementation of the deep sparse grid layer, is examined in Section 2.4.

2.1 Full Grids

In order to define sparse grids, full grids are introduced first. For simplicity's sake, a one-dimensional equidistant grid Ω_l of level $l \in \mathbb{N}$ in the interval $\bar{\Omega} = [0, 1]$ is considered first. Figure 2.1a shows a one-dimensional full grid. Then, these principles will be extended to multiple dimensions in Section 2.1.4. A two-dimensional full grid is shown in Figure 2.1b.

The grid Ω_l consists of $(2^l + 1)$ equidistant points if it contains the boundary points and of $(2^l - 1)$ equidistant points if not [5]. The distance h_l between the points of a grid with level l, also called mesh width, is defined as

$$h_l := 2^{-l}. \tag{2.1}$$

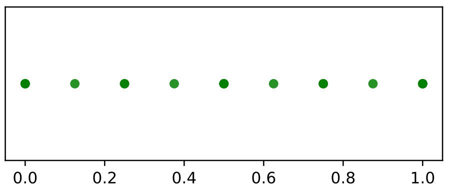
A point $x_{l,i}$ in relation to its index i and the level l of the grid is defined as

$$x_{l,i} := i \cdot h_l, \ \ 0 \le i \le 2^l \tag{2.2}$$

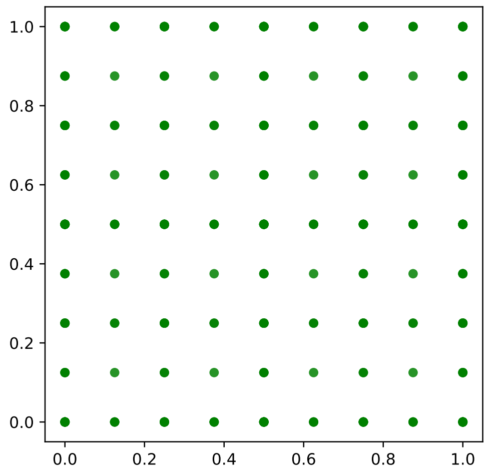
if the grid contains boundary points and as

$$x_{l,i} := i \cdot h_l, \ \ 1 \le i \le 2^l - 1 \tag{2.3}$$

if not.



(a) One-dimensional full grid of level 3.



(b) Two-dimensional full grid of level $(3,3)^\top$.

Figure 2.1: Full grids with boundary points in the domain $[0,1]$.

In the Sections 2.1 and 2.2 a grid without boundary points is considered.

2.1.1 Hat Function

A one-dimensional basis function is used to construct the full grid. The hat function,

$$\Phi(x) := \max\{1 - |x|,\ \ 0\}, \tag{2.4}$$

is a simple one-dimensional basis function [6]. Figure 2.2 shows a plot of the hat function in the interval $[-3,3]$.

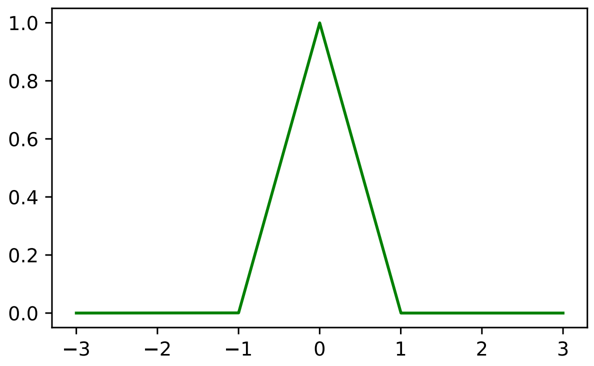


Figure 2.2: The hat function in the interval $[-3, 3]$.

By dilating and translating this function, the hat function can be defined in relation to the grid point $x_{l,i}$:

$$\Phi_{l,i}(x) := \Phi\left(\frac{x - x_i}{h_i}\right). \tag{2.5}$$

This general hat function can then be used to construct a basis.

2.1.2 One-dimensional Nodal Basis

A nodal basis of level l uses a span of level l basis functions to construct its function space. An example of a nodal basis can be seen in Figure 2.3.

Using the basis function from (2.5), the function space V_l of the nodal basis is defined as

$$V_l := \text{span}\left\{\Phi_{l,i} : 1 \leq i \leq 2^l - 1\right\}. \tag{2.6}$$

This space contains all functions that can be represented by the nodal basis [6].

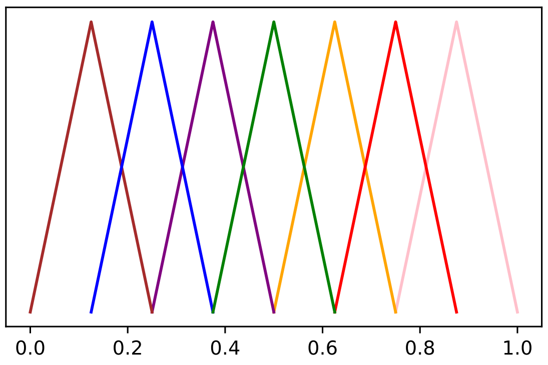


Figure 2.3: A nodal basis of level 3 in the domain $[0, 1]$.

An interpolation function $u(x)$ can be constructed by multiplying the basis functions with a basis-specific weight $v_{l,i}$ and summing up all results:

$$u(x) = \sum_{i \in I_l} v_{l,i} \cdot \Phi_{l,i}(x). \tag{2.7}$$

Since the basis functions are static, the interpolation function $u(x)$ can be represented by its weights.

2.1.3 One-dimensional Hierarchical Basis

A hierarchical basis can also be constructed in a similar way to the nodal basis. The hierarchical basis uses multiple hierarchical increment spaces W_l to build its function space V_l [6]. Each increment space W_l contains the odd basis functions of level l:

$$W_l := \text{span}\left\{\Phi_{l,i} : i \in I_l\right\}, \tag{2.8}$$

$$I_l := \left\{i \in \mathbb{N} : 1 \leq i \leq 2^l - 1, \;\; i \text{ odd}\right\}. \tag{2.9}$$

The sum of the increment spaces W_l of all levels l is the resulting function space

$$V_l = \bigoplus_{k \leq l} W_k, \tag{2.10}$$

which is the same space as the nodal function space. An example of a hierarchical basis can be seen in Figure 2.4.

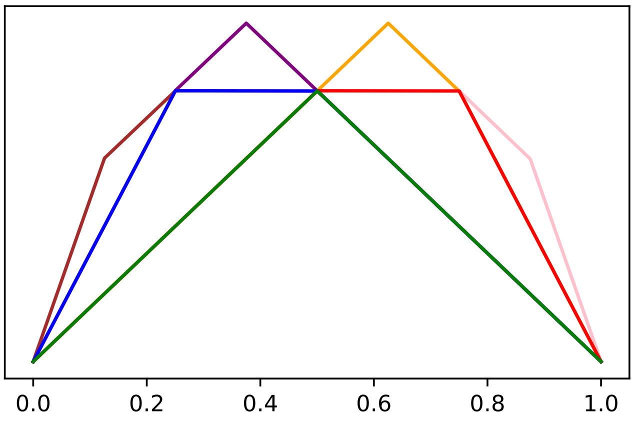


Figure 2.4: A hierarchical basis of level 3 in the domain $[0, 1]$.

A function $u(x)$ inside the function space V_l can be represented by the basis functions as the sum of all hierarchical basis functions in all increment spaces, multiplied by their respective weights $v_{k,i}$:

$$u(x) = \sum_{k=1}^{l} \sum_{i \in I_k} v_{k,i} \cdot \Phi_{k_i}(x). \tag{2.11}$$

The interpolation function $u(x)$ can be uniquely represented by its weights using hierarchical basis function as well.

2.1.4 Multi-dimensional Basis

Tensor product construction can be utilized to extend the one-dimensional grid to d dimensions. The d levels l of the d-dimensional grid Ω_l^d are combined into the d-dimensional vector $\vec{l}$:

$$\vec{l} := (l_1,\ l_2,\ \ldots,\ l_d) \in \mathbb{N}^d. \tag{2.12}$$

The grid Ω_l^d is now in the d-dimensional interval $\bar{\Omega}^d := [0, 1]^d$. The mesh width is defined as

$$\vec{h}_l := (h_{l,1},\ h_{l,2},\ \ldots,\ h_{l,d}) := 2^{-\vec{l}}. \tag{2.13}$$

The grid points $\vec{x}_{\vec{l},\vec{i}}$ of the grid Ω_l^d are defined as

$$\vec{x}_{\vec{l},\vec{i}} := (x_{l_1,i_1},\ \ldots,\ x_{l_d,i_d}),\ \ \vec{1} \le \vec{i} \le 2^{\vec{l}-\vec{1}}, \tag{2.14}$$

where for all vectors above, all arithmetic operations are component-wise [5]. Figure 2.5 shows a full grid with $\vec{l} = (3,3)^\top$.

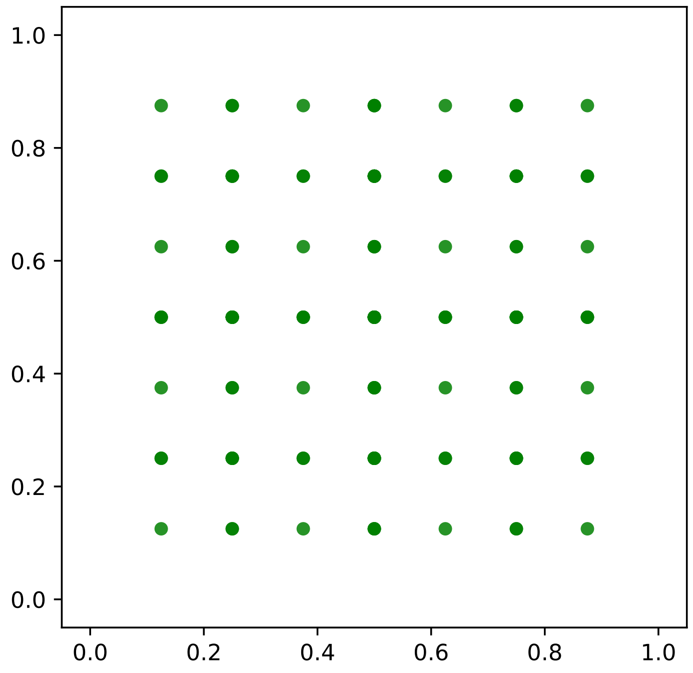


Figure 2.5: A two-dimensional full grid of level $(3,3)^\top$ (without boundary points).

For each grid point the d-dimensional hat function is defined as

$$\Phi_{\vec{l},\vec{i}}(x) := \prod_{j=1}^{d} \Phi_{l_j,i_j}(x_j). \tag{2.15}$$

The function space

$$V_{\vec{l}} := \text{span}\left\{\Phi_{l,i} : \vec{1} \leq \vec{i} \leq 2^{\vec{l}} - \vec{1}\right\} \tag{2.16}$$

is again defined as a span of basis functions.

To represent a function $u_{\vec{l}}(\vec{x})$ in the multi-dimensional function space $V_{\vec{l}}$, the nodal basis functions are multiplied with their respective weights $v_{\vec{l},\vec{i}}$, similar to the one-dimensional case:

$$u_{\vec{l}}(\vec{x}) = \sum_{\vec{i}\in I_{\vec{l}}} v_{\vec{l},\vec{i}} \cdot \Phi_{\vec{l},\vec{i}}(\vec{x}). \tag{2.17}$$

The hierarchical increment spaces $W_{\vec{l}}$ are defined as

$$W_{\vec{l}} := \text{span}\left\{\Phi_{\vec{l},\vec{i}} : \quad \vec{i} \in I_{\vec{l}}\right\}, \text{ with} \tag{2.18}$$

$$I_{\vec{l}} := \left\{\vec{i} \in \mathbb{N}^d : \quad \vec{1} \le \vec{i} \le 2^{\vec{l}} - \vec{1}, \quad i_j \text{ odd for all } 1 \le j \le d\right\}. \tag{2.19}$$

The sum of the increment spaces $W_{\vec{l}}$ of all levels $\vec{l}$ still is the same as the function space $V_{\vec{l}}$ of the nodal basis:

$$V_{\vec{l}} = \bigoplus_{\vec{k}\le\vec{l}} W_{\vec{k}}. \tag{2.20}$$

A function $u_{\vec{l}}(\vec{x})$ can be represented by the sum of all hierarchical basis functions in all increment spaces, multiplied by their respective weights $v_{\vec{k},\vec{i}}$:

$$u_{\vec{l}}(\vec{x}) = \sum_{\vec{1}\le\vec{k}\le\vec{l}} \sum_{\vec{i}\in I_{\vec{k}}} v_{\vec{k},\vec{i}} \cdot \Phi_{k_i}(\vec{x}). \tag{2.21}$$

An interpolation function $u_{\vec{l}}(\vec{x})$ can be represented in the multi-dimensional grid just by its weights as well.

2.2 Sparse Grids

When approximating functions in increasing dimensions, the number of points in a full grid increases exponentially. This is called "the curse of dimensionality". Sparse grids are grids that have a large number of empty grid points. Because of that, the actual number of points in a sparse grid does not increase exponentially while the quality of the approximation does not decrease significantly [6, 7]. Table 2.1 shows a comparison of the grid points and accuracy of full and sparse grids.

Table 2.1: Comparison between full grids and sparse grids in relation to the dimension d and the mesh width h [4].

	Full grid	**Sparse grid**
Grid Points	$\mathcal{O}\left(h^{-d}\right)$	$\mathcal{O}\Big(h^{-1}\log\left(h^{-1}\right)^{d-1}\Big)$
Accuracy	$\mathcal{O}\left(h^{2}\right)$	$\mathcal{O}\Big(h^{2}\log\left(h^{-1}\right)^{d-1}\Big)$

One way of defining sparse grids is by selectively adding hierarchical subspaces from the full grids space (see Figure 2.6). The result of the added hierarchical subspaces can be seen in Figure 2.7. Sparse grids achieve their smaller number of points with little loss in accuracy by selectively adding subspaces from the full grid space (see Figure 2.6). Two norms are used to distinguish the hierarchical subspaces:

$$|\vec{l}|_1 := \sum_{j=1}^{d} |l_j|, \tag{2.22}$$

$$|\vec{l}|_\infty := \max_{1\leq j\leq d} |l_j|. \tag{2.23}$$

To gain the maximum accuracy for the least amount of points, sparse grid spaces $\hat{V}_n$ of levels $\vec{l} = (n, \ldots, n)^\top$ in d dimensions are defined as

$$\hat{V}_n := \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}}. \tag{2.24}$$

Full grid spaces do not selectively choose hierarchical subspaces but use all subspaces from the corresponding full grid space:

$$V_n := \bigoplus_{|\vec{l}|_\infty \leq n} W_{\vec{l}}. \tag{2.25}$$

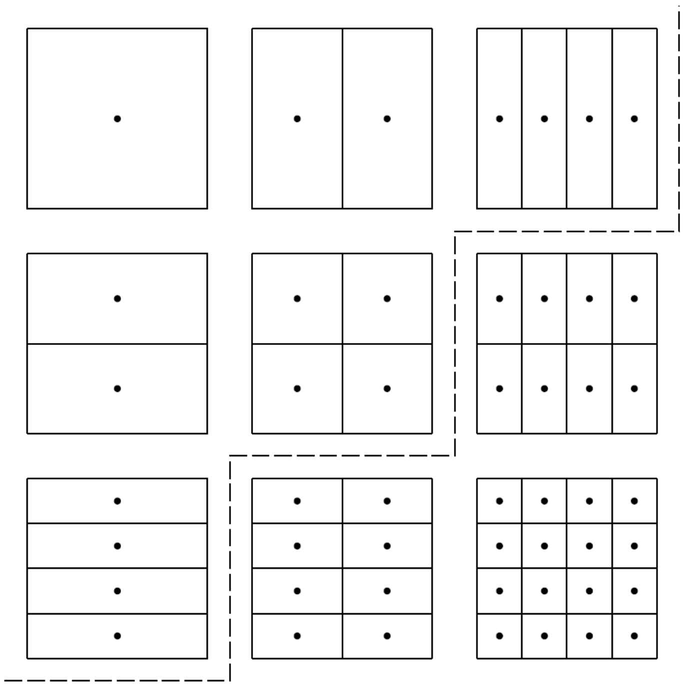


Figure 2.6: Subspaces $W_{\vec{l}}$ for levels $|\vec{l}_\infty| \leq 3$. Together the subspaces form the full grid space V_3. The sparse grid space $\hat{V}_3$ consists of the full grid spaces that satisfy $|\vec{l}|_1 \leq 3 + 2 - 1 = 4$ (see (2.24)), in this figure shown above the dashed line [taken from 6].

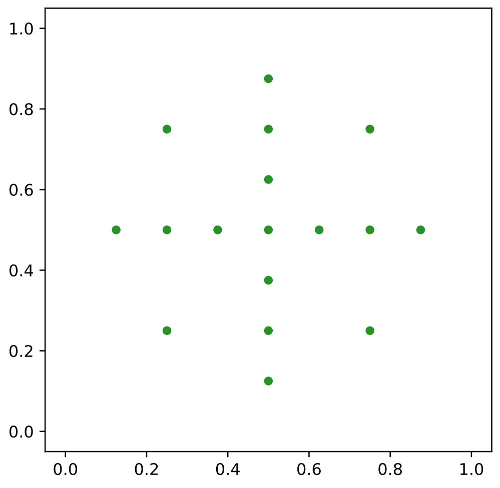


Figure 2.7: Two-dimensional sparse grid of level 3.

The combination technique is another technique used for generating sparse grids. Instead of adding hierarchical subspaces from the full grid space, the combination technique linearly adds and subtracts coarser full grids to represent a sparse grid. The advantage of constructing a sparse grid out of complete full grids instead of hierarchical subspaces is that existing solving methods for full grids can be used on the constructed sparse grid. The disadvantage is that the combination technique uses additional points. An example of the combination technique can be seen in Figure 2.8, where the technique is used to create a two-dimensional sparse grid of level 3.

The sparse grid interpolation function $\hat{u}_l(x)$, also called sparse grid solution, is given by

$$\hat{u}_n(x) := \sum_{n \leq |\vec{l}|_1 \leq n+d-1} (-1)^{n+d-|\vec{l}|_1-1} \binom{d-1}{|\vec{l}|_1 - n} u_l(x). \tag{2.26}$$

The coefficient

$$c = (-1)^{n+d-|\vec{l}|_1-1} \binom{d-1}{|\vec{l}|_1 - n} \tag{2.27}$$

of a grid determines whether the grid is added or subtracted.

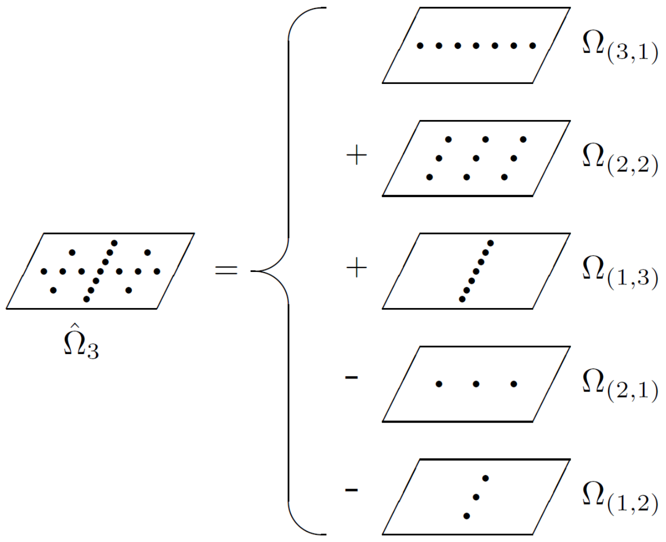


Figure 2.8: A two-dimensional sparse grid of level 3 is combined from full grids. The coefficient c of the grids $\Omega_{(3,1)}$, $\Omega_{(2,2)}$ and $\Omega_{(1,3)}$ is 1, which means that these grids are added. The coefficient c of the grids $\Omega_{(2,1)}$ and $\Omega_{(1,2)}$ is -1, which means that these grids are subtracted [taken from 6].

2.3 Neural Networks

Artificial neural networks are computational models that possess the ability to learn, generalize or cluster data. They are loosely inspired by biological neural networks and consist of a set of simple processing units called neurons. These neurons communicate with each other via weighted connections [8].

Neural networks can perform various tasks, from diagnosing patients with dementia using neural imaging [2] to reading emotions using facial image analysis [1] to playing StarCraft II, a complex real-time strategy game with incomplete information [9].

2.3.1 Types of Neural Networks

There are different types of neural networks, two important types are feed-forward networks and recurrent networks. In feed-forward networks the data flow is unidirectional. No information is transmitted backwards and there are no loops in the network. Figure 2.9a shows a feed-forward network. Recurrent networks on the other hand do transmit data backwards. This means that the network influences itself. The output of a network can either be stable, when the network reaches an equilibrium, or dynamic, when the network does not converge to a stable state and instead periodically changes [8]. Figure 2.9b shows a recurrent network.

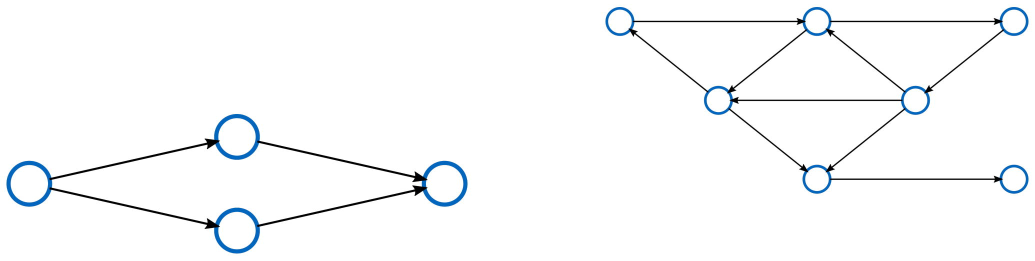

(a) Feed-forward network. (b) Recurrent network.

Figure 2.9: Comparison between a feed-forward network and a recurrent network.

The focus of this thesis will be on feed-forward networks.

2.3.2 Layers

The neurons in a feed-forward network are usually divided into layers. Neurons of layer n only influence neurons of layer $n+1$. Neurons on the same level do not influence each other. Figure 2.10 shows a feed-forward network with an input layer, a hidden layer and an output layer.

The neurons on the first layer are influenced by the input layer, which is not counted in the number of layers of a neural network. The last layer, which is also called output layer, does not influence other neurons in the neural network. All layers between the input and the output layer are called hidden layers. Most neural networks consist of multiple layers. While single-layer networks without hidden layers exist, they are much more restricted than multi-layer networks [8].

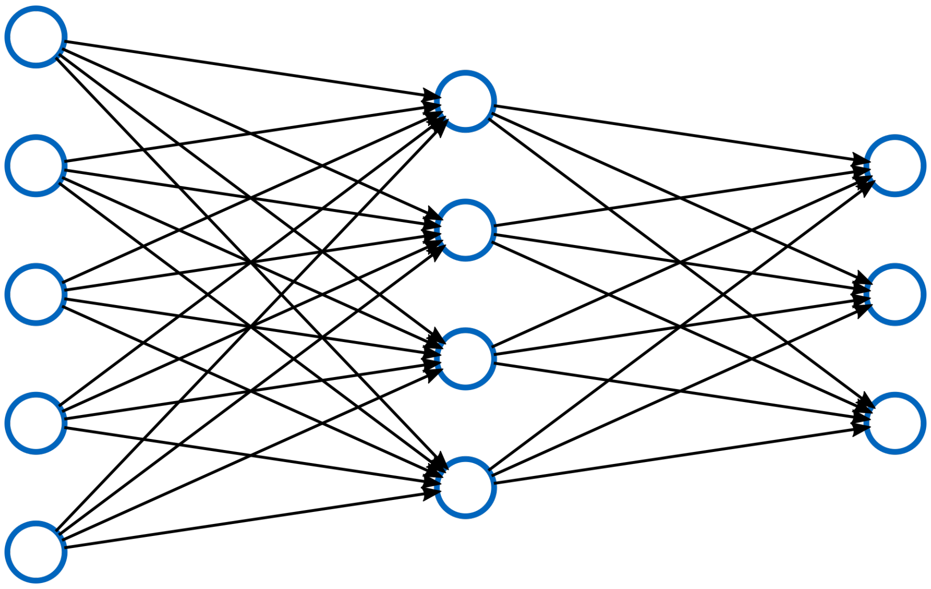


Figure 2.10: Feed-forward neural network with 2 layers.

2.3.3 Forward Pass

To calculate the result of a neural network, which is the states of the neurons in the output layer, a forward pass is executed. In this forward pass the states of all neurons get evaluated. The forward pass starts at the input layer, moves forwards through the network, and ends at the output layer.

Each neuron can assume an activation state s between zero and one. The weight of the connection from neuron j to neuron k is w_{jk}. For each neuron there can also be a bias Θ [8]. The state of the neuron is calculated as the sum of all weighted connections to the neuron plus the bias of the neuron:

$$s_k(t) = \sum_j w_{jk}(t)y_j(t) + \Theta_k(t). \tag{2.28}$$

To calculate the output y of a neuron an activation function $\alpha(x)$ can be used. This activation function can introduce non-linearity to the neural network and help detect certain features [10]. The activation function gets the state of the neuron as input and returns the output of the neuron:

$$y_k(t) = \alpha_k(s_k(t)). \tag{2.29}$$

The output of the neurons is evaluated layer by layer until the output layer is reached. The output y of the neurons of the output layer is the output of the neural network.

2.3.4 Activation Functions

Different activation functions can be used for different purposes. Two popular activation functions are the sigmoid activation function and the rectified linear unit (ReLU) activation function [10].

The sigmoid activation function, shown in Figure 2.11a, is defined as

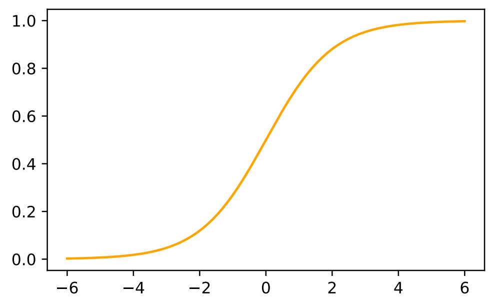
$$f(x) = \frac{1}{(1 + e^{-x})}. \tag{2.30}$$

It is a non-linear smooth activation function that is derivable. It is mainly used in the output layer.

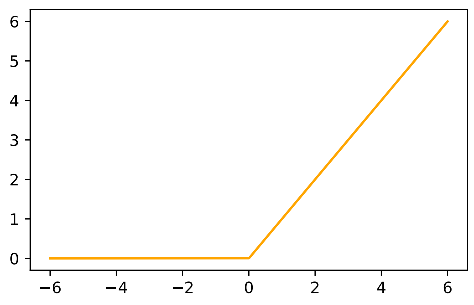
The ReLU activation function, shown in Figure 2.11b, is defined as

$$f(x) = \max(0, x). \tag{2.31}$$

It is a nearly linear activation function which means it is often used in linear models.



(a) Sigmoid activation function.



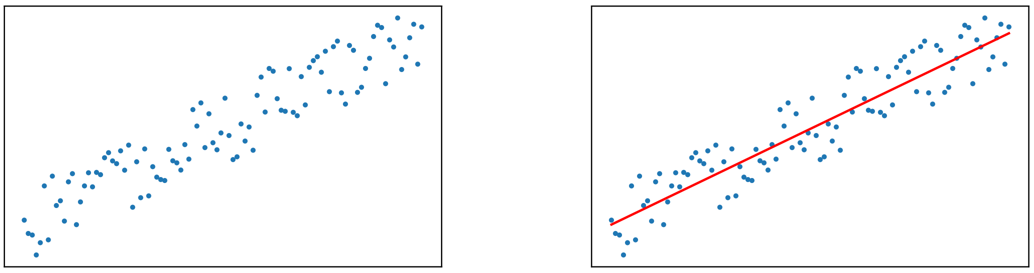
(b) ReLU activation function.

Figure 2.11: Comparison of the sigmoid and the ReLU activation function in the interval $[-6, 6]$.

2.3.5 Problems Solvable with Machine Learning

Problems that can be solved by machine learning can be divided into two groups: supervised and unsupervised learning [11].

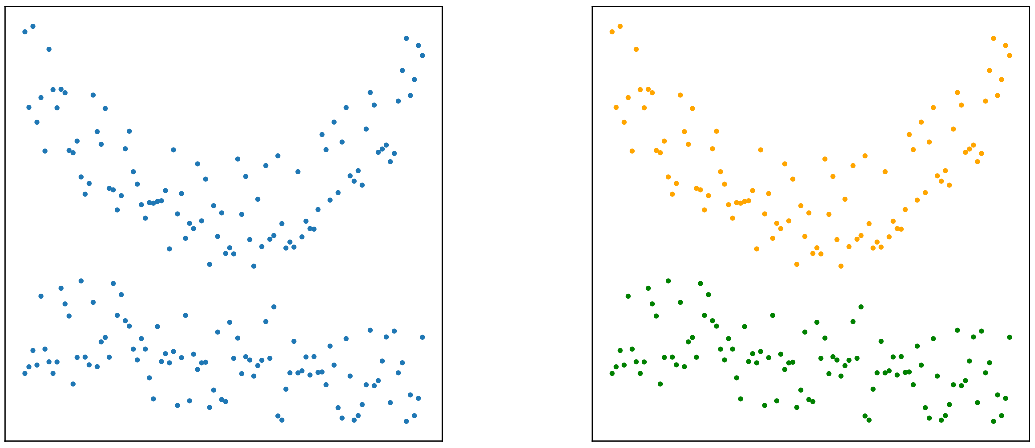
In supervised learning problems, the neural network learns a function that best predicts the output of incoming data by learning from input-output pairs of data. The output that the network should predict is already known. Examples of supervised learning problems are classification, where the network classifies input data into known classes, and regression, where the network learns a function from sample points. An example of regression can be seen in Figure 2.12.



(a) Input data points. (b) Predicted function.

Figure 2.12: Regression example.

Unsupervised learning is done without a priori knowledge about the data. The network gets unlabeled data without instructions on what to predict. It then tries to find patterns in the data. An example of unsupervised learning problems is clustering, which is the division of unknown data into clusters of similar data points. The number of clusters and the features that are used in the clustering process are not known beforehand. An example of this can be seen in Figure 2.13.



(a) Input data points. (b) Clustered data.

Figure 2.13: Clustering example.

2.3.6 Loss Function

A loss function is used to measure how well a model performs. It quantifies the difference between the value predicted by the model and the expected value. This quantification is then used to tune the model's parameters to better predict the input data [12]. Because the loss function compares to an expected value, it is only usable in supervised learning. This tuning of the model's parameters is explained in Section 2.3.7.

There are different loss functions for different purposes. A simple loss function used for example in regression problems is the quadratic loss function, also called squared error (SE) function. It uses the squared distance between prediction y and target T as loss value. It is defined as

$$L(y) = B \cdot (y - T)^2, \tag{2.32}$$

where B is a constant.

The mean value of the quadratic loss function over a batch of data is called the mean squared error (MSE) function. It is defined as

$$\text{MSE}(y) = \frac{1}{n} \sum_{i=0}^{n} B \cdot (y_n - T_n)^2 \tag{2.33}$$

for n different data points.

2.3.7 Backpropagation

After training data was passed forward through the neural network, the loss of this forward pass is calculated. In case of a batch of training data, a mean loss function is used. Then the weights and biases of the neurons and connections in the neural network get adjusted to minimize the loss. This is done by passing the loss backwards through the network. The adjustment of the parameters to minimize the loss are done layer by layer. An algorithm such as gradient descent can be used to find a local minimum of the loss function for different weights and biases. The weights and biases can then be optimized [8].

2.4 PyTorch

PyTorch is an open-source Python library that is used for deep learning. It provides tensors that support calculations both with central processing units (CPUs) and graphical processing units (GPUs). Using a GPU allows for faster parallel calculations. The tensors save a history of changes, which provides built-in differentiation.

The library provides a wide range of ready-to-use layers that can easily be added to custom models. It also provides optimizers and loss functions used for backpropagation [13].

The implementation discussed in Section 3 defines a deep sparse grid layer in PyTorch. The steps required for creating a custom layer in PyTorch and using the layer in a PyTorch model are introduced in Sections 2.4.1 and 2.4.2.

2.4.1 Model Definition

To define a new model in PyTorch, the `torch.nn.Module` class is extended. The programmer then has to define the layers. In addition, the function that is executed with each forward pass needs to be defined as well. A code example can be found below in Source Code 2.1 [14].

```python
class CustomModel(torch.nn.Module):
    def __init__(self, dim_in, dim_h1, dim_out):
        super(CustomModel, self).__init__()
        self.linear1 = torch.nn.Linear(dim_in, dim_h1)
        self.linear2 = torch.nn.Linear(dim_h1, dim_out)

    def forward(self, x):
        h1 = F.relu(self.linear1(x))
        y  = torch.sigmoid(self.linear2(h1))
        return y

my_model = CustomModel(3, 2, 1)
```

Source Code 2.1: Definition of a model with two linear layers.

The model consists of two linear layers. The first linear layer uses the ReLU activation function, while the second linear layer uses the sigmoid activation function.

2.4.2 Layer Definition

To define a custom layer in PyTorch, the `torch.nn.Module` class also needs to be extended. Inside the layer class a forward function needs to be defined that gets executed with every forward pass. The custom layer can then be used in any model. A code example can be found below (see Source Code 2.2).

```python
class CustomLinearLayer(torch.nn.Module):
    def __init__(self, in_features: int, out_features: int) \
            -> None:
        super(CustomLinearLayer, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(
            torch.Tensor(out_features, in_features))


    def forward(self, x):
        return torch.nn.functional.linear(x, self.weight)


class CustomModel(torch.nn.Module):
    def __init__(self, dim_in, dim_h1, dim_out):
        super(CustomModel, self).__init__()
        self.linear1 = torch.nn.Linear(dim_in, dim_h1)
        self.custom_linear = CustomLinearLayer(dim_h1, dim_out)

    def forward(self, x):
        h1 = F.relu(self.linear1(x))
        y  = torch.sigmoid(self.custom_linear(h1))
        return y

my_model = CustomModel(3, 2, 1)
```

Source Code 2.2: Definition of a custom layer and instantiation of it in a custom model.

In this code example a custom linear layer is defined. The forward pass of the custom layer is a function call to a built-in function in the PyTorch library. The custom linear layer is then used after a linear layer in the defined model.

3 Implementation of a Deep Sparse Grid Layer in PyTorch

Sparse grids perform best in ambient space dimensions larger than three when an exponential increase of points is not feasible anymore, but the space still has to be sampled completely. The sparsity does not only scale better in high dimensions but can also help with pattern recognition by dropping unnecessary features, making the system more robust in return [15].

Sparse grids in combination with machine learning can also be useful for density estimation, since it needs an exponentially growing number of grid points compared to data points, whereas the number of sparse grid points does not grow as fast [16].

The scope of this thesis is the implementation of a deep sparse grid layer in PyTorch, using the already existing codebase as a starting point.

The features of the codebase will be introduced in Section 3.1. Afterwards, an overview of the representation of sparse grids in the deep sparse grid layer implementation is given in Section 3.2. The features and options of the layer implementation are summarized in Section 3.3. In Section 3.4 the grid generation algorithm is explained. The architecture of the implementation and its modularization is discussed in Section 3.5. Unit tests, which ensure the functionality of the implementation of the deep sparse grid layer and its components, are outlined in Section 3.6. Finally, in Section 3.8, the topic of using parallel computation from GPUs with the implementation of the deep sparse grid layer is discussed.

3.1 Deep Sparse Grids Codebase

The already existing codebase implements a neural network with a sparse grid layer for Tensorflow, a Python library similar to PyTorch that is also used for machine learning.

An algorithm to create a d-dimensional sparse grid without boundary points in the domain $[0,1]^d$ using the combination technique is also already implemented in the codebase. To create a sparse grid with the combination technique, a set of full grids is multiplied with a set of coefficients and then added together.

The sparse grid representation in the codebase stores the sparse grid in three variables. The grid points of all full grids that make up the sparse grid are stored together in the variable `positions`. When representing a d-dimensional sparse grid with p grid points, the array `positions` has the shape (p, d). The grid coefficients of the set of full grids are stored for every point. This means that the variable called `combi_coefficients` contains the grid coefficient for every grid point, which makes it a one-dimensional array of length p. The mesh widths of the set of full grids are also stored for every grid point in every dimension, which results in an array with the shape (p, d) called `scales`.

3.2 Sparse Grid Representation in the Deep Sparse Grid Layer

Sparse grids in the deep sparse grid layer are also generated using the combination technique. The representation of sparse grids is very similar to the representation in the codebase. Three variables with the same functions and shapes as in the codebase are used. To make the naming more consistent with the purpose of the variables, two of them have been renamed. The variable that contains the positions of all grid points is still called `positions`. The variable containing the grid coefficients for every grid point is called `grid_coefficients`. The mesh widths for each grid point are stored in `mesh_widths`.

Full grids are represented using the same variables. The positions of the points are stored the same way the positions of sparse grid points are stored. Since full grids are just a single full grid and not a combination of multiple full grids, no grid coefficients are needed. This means that the grid coefficient of all grid points is 1. The mesh width for all grid points is equal as well. It only depends on the domain and level of the full grid.

3.3 Features of the Deep Sparse Grid Layer

The `sparsegrid_layer` module allows its user to create deep sparse grid layers, which can be used in PyTorch neural networks. When using the layer in a PyTorch model, the sparse grid layer can be customized to best accomplish its task.

To allow easy comparisons between sparse and full grids, the grid type can be chosen when instantiating the layer. The generation of sparse grids using the combination technique is supported. Full grids can be generated as well. The generation of both grid types is also accessible from outside of the sparse grid layer, which makes it possible to use the grid generation algorithm outside of a neural network.

Both grid types can be generated with or without boundary points. The use of boundary points increases the number of points that are used in the grid, which makes calculations slower. However, it can be beneficial to use boundary points when the data in the neural network has boundary conditions that are not zero.

The level of the grid can be specified as well, which is used to define the level of detail that the grid has. A higher level means more grid points, but higher accuracy as well.

When generating a new grid the grid domain can be chosen as well. By default, the grid will be generated in the domain $[0, 1]^d$. It is possible to choose different grid domains for each dimension.

Basis functions on each grid point are used to construct the grid. The basis function used in the sparse grid layer can be specified as well. The default basis function is the hat function (2.5). Currently, only the hat function is implemented. However, a function wrapper exists that allows to easily define further basis functions.

The weights of the sparse grid layer can be optimized with an optimizer from the PyTorch library together with the other layers in a model. It is, however, also possible to optimize the weights of the sparse grid layer with a custom optimizer. Optimizing the weights during a training step is also possible. A PyTorch implementation of a regularized least squares solver implemented by Zhen Zhang also comes with the deep sparse grid layer. This solver can be used to optimize the weights of the layer.

3.4 Grid Generation

The grid generation of the deep sparse grid layer is done by the `grid_generation` module. The generation function is accessible outside of the sparse grid layer, which means it can be used to generate sparse grids and full grids for other use cases as well. It is possible to generate sparse and full grids of different levels and dimensions in a chosen domain with or without boundary points.

The grid generation is done in two parts.

First, a list of full grids, called scheme, is generated, together with a list of grid coefficients. Each full grid is represented by a d-tuple specifying the level of the full grid in each dimension d.

To generate sparse grids, the combination technique (see Section 2.2) is used. This technique constructs a sparse grid out of multiple full grids, which are multiplied with a coefficient and added together. All full grids that satisfy the condition

$$n \leq |l|_1 \leq n + d - 1, \tag{3.1}$$

where n is the level and d the dimension of the sparse grid, are added to the scheme. The coefficient of the full grid is calculated with (2.27).

To generate full grids, no addition of multiple grids needs to happen. The scheme simply contains the final full grid. The corresponding coefficient is 1.

Source Code 3.1 shows the algorithm used to generate the grid scheme and coefficients in the deep sparse grid layer implementation.

```
Function create_combi_scheme(dim, level)

    # create all possible level vectors that could be part of the scheme
    level_list = list from 1 to level
    level_lists = list of level_list for i=1 to dim
    level_vectors = cartesian_product(level_lists)

    # add all fitting level vectors to the scheme
    scheme = empty list
    For level_vector in level_vectors:
        If level <= L1_norm(level_vector) <= level + dim - 1
            Append level_vector to scheme
        Endif
    Endfor

    # calculate the coefficients for all level_vectors in the scheme
    coefficients = empty list
    For level_vector in scheme:
        q = level + dim - 1 - L1_norm(level_vector)
        f = (dim - 1) choose (L1_norm(level_vector) - level)
        coefficient = (-1)^q * f
        Append coefficient to coefficients
    Endfor

    Return (scheme, coefficients)
Endfunction
```

Source Code 3.1: Generation of the grid scheme and grid coefficients used to build a sparse grid using the combination technique.

After the creation of a scheme and grid coefficients, the actual grid generation starts. For each full grid specified in the scheme the following steps are executed:

1. The coordinates of the full grid are generated and saved to an array called positions.
2. The grid coefficient of the full grid is expanded into an array called grid coefficients, which is the same length as positions.
3. The mesh width of the full grid is expanded into an array called mesh widths, which is the same size as positions.

The arrays representing each full grid are then merged into one array representing the final grid.

3.5 Layer Architecture

The main class in the `sparsegrid_layer` module is called `SparsegridLayer` and extends `torch.nn.Module`, the main building block for layers in PyTorch. It contains the layer's weights and implements the forward pass of the layer, which multiplies the result of the sparse grid with the layer's weights. All other functionality is implemented in different modules. This makes extending the sparse grid layer function easier. It also provides easier access to functions like the grid generation algorithm, which are used in the sparse grid layer.

To calculate the result of the sparse grid, the `basis_functions` module is used. It contains all basis functions that can be used in the sparse grid layer. Custom basis functions can also be added here.

To generate a grid, the `generate_grid` module is used. It implements the generation of sparse and full grids as described in Section 3.4.

Useful math functions are located in the `math_util` module. The regularized least squares solver for PyTorch implemented by Zhen Zhang is located here.

The module `plot_grid` implements a function that plots a sparse- or full grid in up to two dimensions, which can be helpful for debugging purposes. Figure 3.1 showcases a plot produced by the module. The plot features a two-dimensional sparse grid of level 4, that was built using the combination technique. The red dots indicate where subtracted grid points are. The green dots are the added grid points. Red grid points are plotted with 25% opacity while green grid points are plotted with 75% opacity. Multiple grid points in the same spot result in a point that appears more opaque.

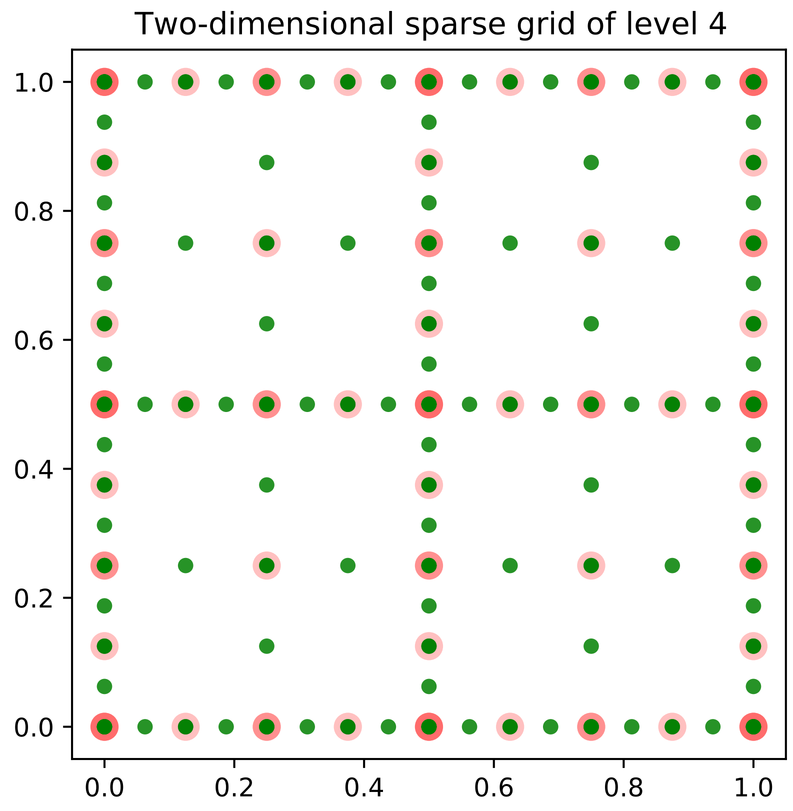


Figure 3.1: Sparse grid plotted by the plot grid module. Green and red dots represent added and subtracted grid points in the sparse grid built with the combination technique. The more opaque a point is, the more grid points are in the same spot.

```
sparsegrid_layer
+-- sparsegrid_layer
    +-- __init__.py
    +-- basis_functions.py
    +-- grid_generation.py
    +-- math_util.py
    +-- plot_grid.py
    +-- sparsegrid_layer.py
+-- test
    +-- __init__.py
    +-- test_basis_functions.py
    +-- test_grid_generation.py
    +-- test_math_util.py
    +-- test_plot_grid.py
    +-- test_sparsegrid_layer.py
```

Figure 3.2: Directory structure of the sparse grid layer.

Figure 3.2 showcases the directory structure of the sparse grid layer and its components. The main modules are located in the folder `sparsegrid_layer`. The unit tests are found separately in the folder `test`.

3.6 Unit Tests

To ensure the functionality of the deep sparse grid layer and all of its components, unit tests are employed. These tests are used to ensure that all parts of the code still function properly after implementing new features or rewriting code. They also check that unallowed inputs get handled correctly. The unit tests are found in the subfolder `test`, as seen in Figure 3.2. The unit tests used in the implementation are listed in Table 3.1.

Table 3.1: List of the unit tests used to test the deep sparse grid layer and its components.

Test method name	Function
Basis functions	
`test_parameter_validation`	Tests for correct input validation.
`test_hat_function`	Tests the hat function implementation.
Grid generation	
`test_parameter_validation`	Tests for correct input validation.
`test_combi_scheme`	Tests the generation of the combi grid scheme.
`test_combi`	Tests the generation of combi grids.
`test_full_scheme`	Tests the generation of the full grid scheme.
`test_full`	Tests the generation of full grids.
Math util	
`test_lstsq`	Tests the regularized least squares solver.
Plot grid	
`test_parameter_validation`	Tests for correct input validation.
Sparsegrid layer	
`test_parameter_validation`	Tests for correct input validation.
`test_forward_pass`	Tests basic functionality of the forward pass.

The implementation passes all unit tests. The tests for the regularized least squares solver are implemented by Zhen Zhang.

3.7 Computational Experiments

Section 3.8 performs experiments regarding the ability of the deep sparse grid layer implementation to use parallel computation.

The models used in the experiments perform regression on the three-dimensional swiss roll dataset from the scikit-learn Python library [17]. The number of sample points is 8000 and the noise is 0.001.

The training is done over 200 epochs with a batch size of 32. The Adam optimizer implemented in the PyTorch library is used with a learning rate of 0.005.

Four different PyTorch models are used in the experiments. The input sample size of all models is 3 and the output sample size is 1. The sparse grid layers used in the models all use a level 6 sparse grid in the domain $[0, 1]^d$ with boundary points built with the combination technique. Model [1] consists of three linear layers that encode the three-dimensional function, all with an input sample size of 3, followed by one two-dimensional sparse grid layer. Model [2] uses only one linear layer and one sparse grid layer. The linear layer is used to reduce the dimension of the input from three to two dimensions. The sparse grid layer once again uses a two-dimensional sparse grid. Model [3] consists only of one sparse grid layer, which uses a three-dimensional sparse grid. Model [4] does not use a sparse grid layer. It only uses two linear layers. To make better comparisons with model [3], which uses a high number of grid points, the hidden layer has an input sample size of 15188, which is also the number of grid points in a three-dimensional sparse grid of level 6 with boundary points created with the combination technique.

3.8 GPU Optimization

The library PyTorch provides tensors that can use the GPU of a system for parallel computations. To utilize the GPU of a system, a specific library from the GPU's manufacturer needs to be used. For these comparisons, the Nvidia GPU GTX 1060 in combination with Nvidia's CUDA library[1] is used. The CPU used is the AMD Ryzen 3 1300X.

The arrays used to represent the sparse (or full) grid inside the sparse grid layer are implemented as PyTorch tensors. While tensors, in general, can be calculated in parallel using a GPU, it is unclear if the current representation of the sparse (or full) grid in conjunction with the implementation of the basis functions can benefit from using CUDA, because it is more complex than the multiplication of just two tensors. All functions used during training are functions from the PyTorch library that support parallel tensor calculations. However, the implementation was not tailored specifically for parallel computation.

To analyze how much the current implementation of the sparse grid layer benefits from parallel computations, the performance gain from using CUDA computations over CPU computations is analyzed and compared to the linear layer, which is implemented in the PyTorch library. To compare the layers, different PyTorch model configurations are used. These configurations and the dataset are explained in Section 3.7. The time spent on the training while using CPU calculations and while using GPU calculations utilizing the CUDA library is compared. To get more reliable results, each training is done five times. A comparison of the quality of the models is not part of the scope of this thesis.

[1]CUDA Toolkit Documentation: `https://docs.nvidia.com/cuda/`

The execution time (in seconds) of model [1], which uses three linear layers and one two-dimensional sparse grid layer, is plotted in Figure 3.3. The three linear layers are used to reduce the three-dimensional input to two dimensions so that a two-dimensional sparse grid can be used.

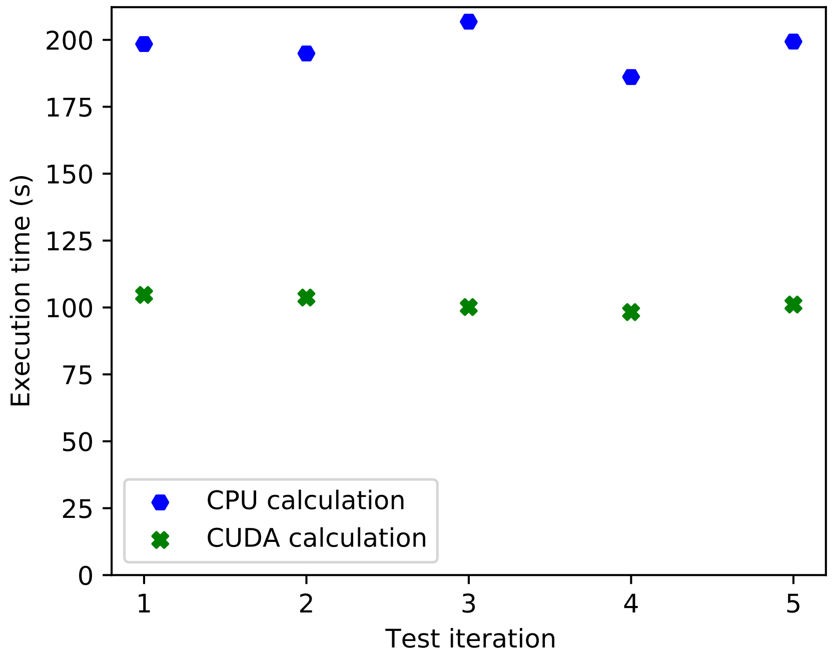


Figure 3.3: The execution time (in seconds) for model [1], consisting of three linear layers and one sparse grid layer using a two-dimensional sparse grid.

It is noticeable that the training utilizing CUDA calculations is about 48% faster than using CPU calculations on average. However, since the model consists of three linear layers and only one sparse grid layer, it is possible that the computation acceleration observable in Figure 3.3 is not because of the sparse grid layer.

The next experiment is done with model [2], which uses only one linear layer and one sparse grid layer. The linear layer is used to reduce the dimension of the input to two dimensions. This means that the sparse grid layer can once again use a two-dimensional sparse grid.

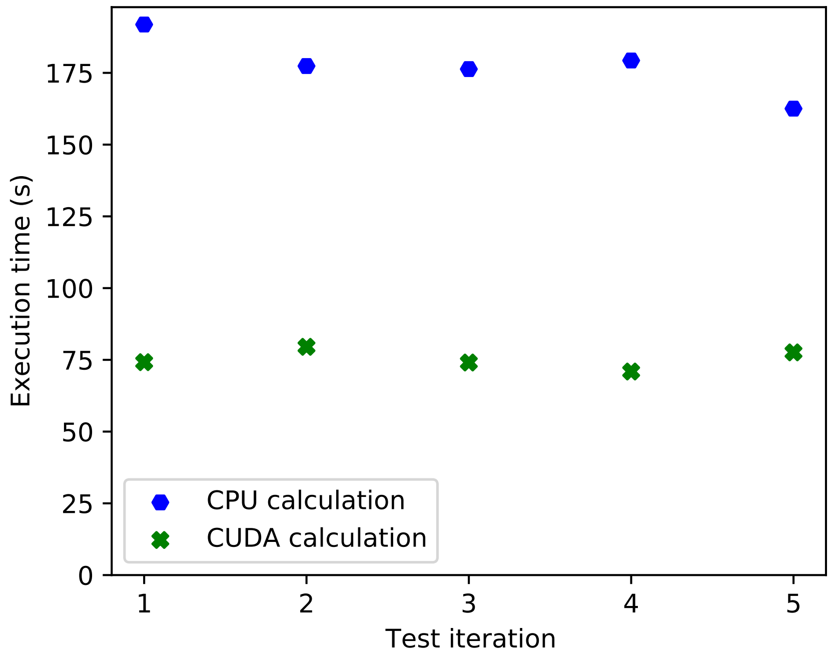


Figure 3.4: The execution time (in seconds) for model [2], consisting of one linear layer and one sparse grid layer using a two-dimensional sparse grid.

Figure 3.4 shows that the execution time of the training done with CUDA calculations is around 57% faster than the execution time of the training done with CPU calculations on average. This could mean that the current implementation of the sparse grid layer can benefit from CUDA calculations. However, since a linear layer is still used in the model, the difference in execution time could be entirely a result of the linear layer.

To ensure that only the sparse grid layer implementation is tested, a model consisting only of one sparse grid layer is used next (model [3]). This time the sparse grid layer uses a three-dimensional sparse grid of level 6 with boundary points, since the input is three-dimensional as well. This means that the sparse grid consists of much more points than before. For context, a two-dimensional sparse grid of level 6 with boundary points built using the combination technique only has about 1500 points, while a three-dimensional sparse grid with otherwise the same properties consists of about 15000 points. This means that the execution is going to be slower. However, it could also mean that parallel CUDA computation could have a bigger impact on the execution time for this model than for the past models.

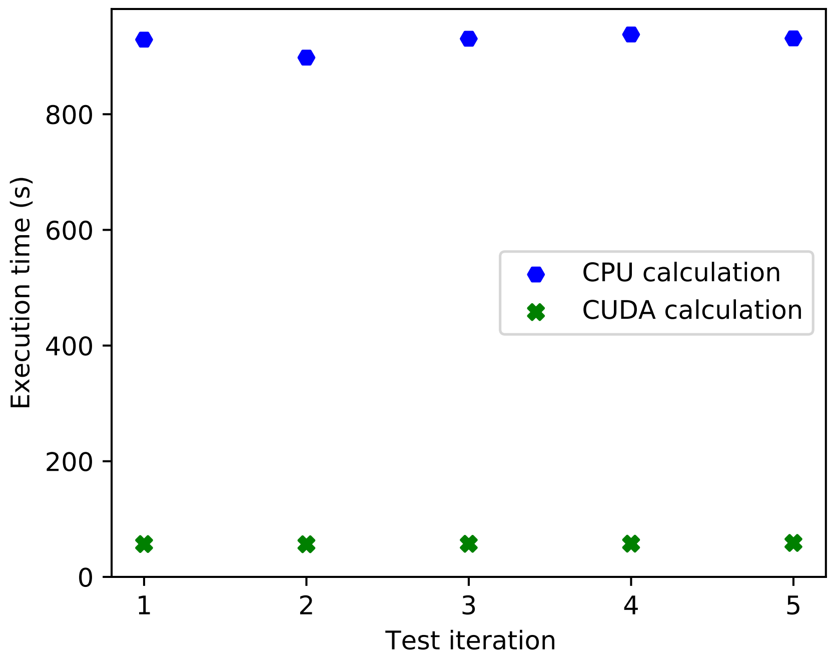


Figure 3.5: The execution time (in seconds) for model [3], consisting of one three-dimensional sparse grid layer.

Figure 3.5 shows that calculations using CUDA are around 93% faster than calculations using the CPU. The execution time for using CPU calculations is much slower than before. This is a result of the increase in sparse grid points. In spite of the increase in execution time using CPU computation, the CUDA execution time is similar compared to the models before. This is because using CUDA allows for parallel processing of the sparse grid points. Since this model only uses a sparse grid layer and no other layers, this means that its current implementation does benefit from parallel GPU computation.

The last experiment is done on model [4], which uses only two linear layers and no sparse grid layer. The hidden layer uses a similar number of points to the three-dimensional sparse grid layer.

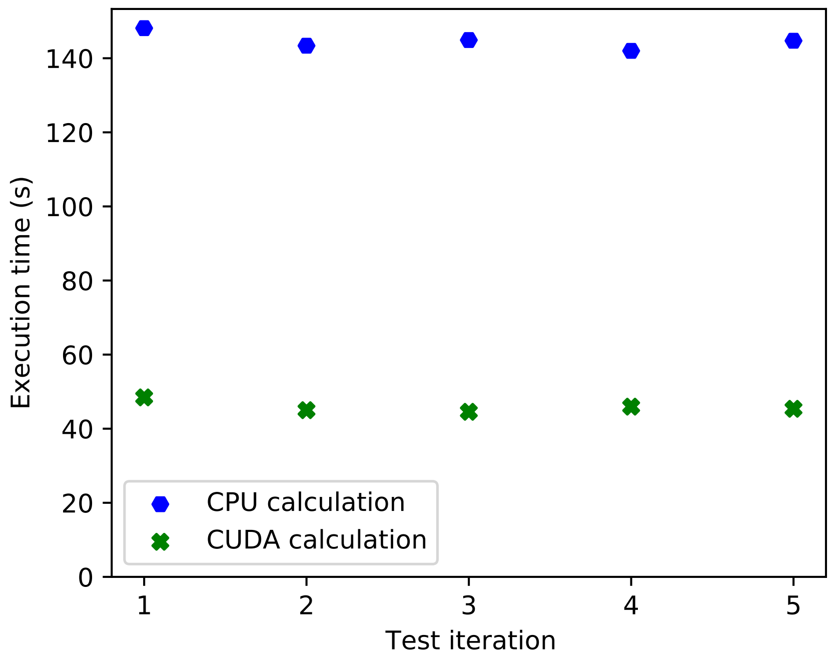
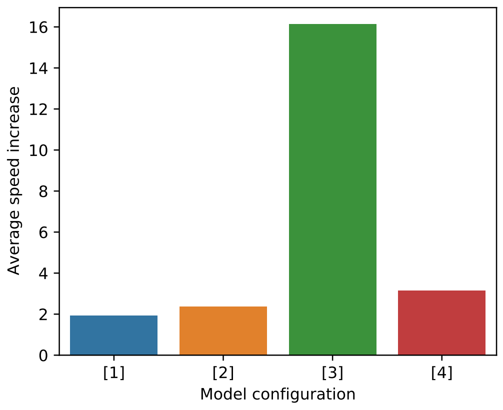


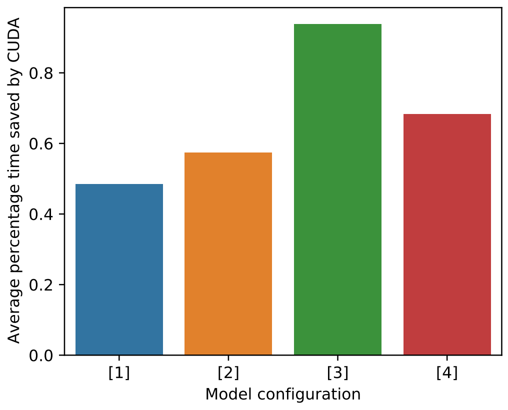
Figure 3.6: The execution time (in seconds) for model [4], consisting of two linear layers.

The use of CUDA calculations also affects this model, as shown in Figure 3.6. On average, the training using CUDA calculations was around 68% faster than the training using only CPU calculations. The linear layer implemented in the PyTorch library also benefits from parallel computation, as expected. The execution time is lower than in the model utilizing only the sparse grid layer though. This might be because the sparse grid layer is more complex than a linear layer, even when using a similar number of points.

Two metrics are introduced to compare the use of CUDA calculations with CPU calculations: The average speed increase when using CUDA (shown in Figure 3.7a) and the average time saved by using CUDA (shown in Figure 3.7b).



(a) Average speed increase with CUDA calculations.



(b) Average percentage time saved by CUDA calculations.

Figure 3.7: Comparison of CUDA and CPU calcuations in respect to the four models: three linear into one sparse grid layer(model [1]), one linear into one sparse grid layer(model [2]), only one three-dimensional sparse grid layer(model [3]) and two linear layers using a similar number of points as the three-dimensional sparse grid layer(model [4]).

Model [3] benefited the most from using CUDA computation on average. The training was over 16-times faster when using CUDA computation compared to CPU computation. This corresponds to an average percentage time saved of over 90%. The sparse grid used in model [3] uses a large number of grid points, which explains why model [3] benefits so much from parallel computation. The hidden layer in model [4] uses a similar number of grid points as model [3] but does not see as much of a speed increase as model [3] does. This could be explained by the complexity of the sparse grid layer, which is higher than the complexity of a linear layer. Since the sparse grid layer is more complex it could gain more from parallel computation. Model [1] and model [2] also benefit from CUDA computation and are around two times faster with parallel computation. These models do not consist of as many points as models [3] and [4], which could mean that they can benefit less from parallel computation.

4 Conclusion

This section summarizes the achieved results and looks into possible future work done on the implementation.

4.1 Summary

This bachelor's thesis discussed the implementation of the deep sparse grid layer and provided an introduction to sparse grids and machine learning. The mathematical background of full grids was provided in Section 2.1. In Section 2.2 this concept was extended to sparse grids. The basics of neural networks were discussed in Section 2.3. Afterwards the Python machine learning library PyTorch was introduced in Section 2.4.

In Section 3 the implementation of the deep sparse grid layer was presented, starting with the codebase, which is the foundation of the deep sparse grid layer, in Section 3.1. The representation of sparse grids in the final implementation was explained in Section 3.2. In Section 3.3 the features of the deep sparse grid layer were listed and the options explained. The grid generation algorithm for full grids and sparse grids was discussed in Section 3.4. The general layer architecture and module separation was shown in Section 3.5. In Section 3.6 the unit tests used to verify the functionality of the deep sparse grid layer were listed. Lastly, in Section 3.8, the parallel processing capabilities utilizing GPU computation of the deep sparse grid layer implementation were evaluated.

4.2 Discussion

Until now, there was a lack of frameworks providing easy access to sparse grids in neural networks. With the completion of the deep sparse grid layer, an entry point to researching the possibilities of the combination of sparse grids and neural networks was created. Future researchers can start realizing their ideas quicker and can easily extend the existing sparse grid layer in case they need specific features that are not yet implemented.

The current sparse grid layer implementation supports the use of sparse grids inside a neural network. The generation and use of both full grids and sparse grids is supported, which enables comparisons between the two grid types. The sparse grids used are generated using the combination technique. Both grid types can be generated with and without boundary points. The sparse grid layer can be tailored to the data that is processed within. It is also possible to specify a domain for the used grid. Different domains for the different dimensions of the grid are supported. The basis function used to construct the grid can be customized as well.

The operations done on the sparse grid layer benefit from parallel GPU computing. When using a larger number of grid points, increases of over 90% were observed in tests. This makes studying neural networks with detailed sparse grid layers possible in a reasonable time frame.

4.3 Outlook

In the future, the implementation of the deep sparse grid layer could be extended further. The existing options could be extended, like supporting more basis functions or implementing the generation of other grid types. Adaptive sparse grids, which dynamically adjust the sparse grid to be denser at certain points, could be added as a new feature. This would allow using the grid points of the sparse grid more efficiently. More tests evaluating how the sparse grid layer performs in extreme cases should be done as well.

The current implementation was built putting the convenience of the user first. Because of this, adding new features and options took priority over optimizing the implemented algorithms. It is probably possible to optimize the algorithms used in the sparse grid layer to get faster computation. Tailoring the implementation to parallel computation might further increase the benefit of using CUDA and increase performance for large grids. There might be a performance gain using a different, more performance-oriented, data structure to represent the sparse grids as well.

List of Figures

List of Tables

List of Source Code

Bibliography

[1] Tuhin Kundu and Chandran Saravanan. Advancements and recent trends in emotion recognition using facial image analysis and machine learning models. In *2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*, pages 1–6. IEEE, 2017.

[2] Md Rishad Ahmed, Yuan Zhang, Zhiquan Feng, Benny Lo, Omer T Inan, and Hongen Liao. Neuroimaging and machine learning for dementia diagnosis: Recent advancements and future prospects. *IEEE reviews in biomedical engineering*, 12:19–33, 2018.

[3] Miguel A Carreira-Perpinán. A review of dimension reduction techniques. *Department of Computer Science. University of Sheffield. Tech. Rep. CS-96-09*, 9:1–69, 1997.

[4] Jochen Garcke, Michael Griebel, and Michael Thess. Data mining with sparse grids. *Computing*, 67(3):225–253, 2001.

[5] Thomas Gerstner and Michael Griebel. Sparse grids. *Encyclopedia of Quantitative Finance*, 2008.

[6] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. *Acta Numerica*, pages 1–123, 2004.

[7] Michael Griebel, Michael Schneider, and Christoph Zenger. A combination technique for the solution of sparse grid problems. 1990.

[8] Ben Kröse, Ben Krose, Patrick van der Smagt, and Patrick Smagt. An introduction to neural networks. 1993.

[9] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. 2017.

[10] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. 2018.

[11] Kenneth Rose. Deterministic annealing for clustering, compression, classification, regression, and related optimization problems. *Proceedings of the IEEE*, 86(11):2210–2239, 1998.

[12] Fred A Spiring. The reflected normal loss function. *The Canadian Journal of Statistics/La Revue Canadienne de Statistique*, pages 321–330, 1993.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. 2019.

[14] Torch Contributors. Pytorch documentation. `https://pytorch.org/docs/stable/index.html`, 2019. Accessed: 2021-04-24.

[15] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[16] Benjamin Peherstorfer, Dirk Pflüge, and Hans-Joachim Bungartz. Density estimation with adaptive sparse grids for large data sets. In *Proceedings of the 2014 SIAM international conference on data mining*, pages 443–451. SIAM, 2014.

[17] 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*, 12:2825–2830, 2011.