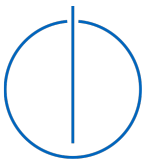
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis

Solving partial differential equations in high dimensional spaces using Deep Sparse Grids

Ahmet Ege Semiz



DEPARTMENT OF INFORMATICS

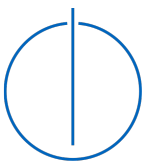
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis

Solving partial differential equations in high dimensional spaces using Deep Sparse Grids

Lösung von partiellen Differentialgleichungen in hochdimensionalen Räumen mit Deep Sparse Grids

Author:	Ahmet Ege Semiz
Supervisor:	Dr. Felix Dietrich
Advisor:	Dr. Felix Dietrich
Submission Date:	August 15th, 2022



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

Munich, August 15th, 2022 Ahmet Ege Semiz

Acknowledgments

I would like to thank my advisor and supervisor, Dr. Felix Dietrich, as his insight and feedback throughout the writing of this thesis has helped me immensely.

Abstract

Partial differential equations (or PDEs) are a type of problem that comes up in diverse areas of mathematics and engineering. From the heat equation to the wave equation, they are an obstacle that frequently needs to be tackled. While PDEs in low-dimensional base spaces are mostly well understood, those in high-dimensional base spaces are a lot more complicated yet still common, and require better methods to deal with.

PDEs are not unique in this regard, as problems such as regression, classification, and more also suffer from this so-called "curse of dimensionality", e.g. the difficulty spike while dealing with problems with higher dimensions. Some approaches to this problems are sparse grids and neural nets, both ways of approximating high-dimensional functions in a more efficient way. Sparse grids attempt do accomplish this by describing the function in terms of hierarchical basis functions, and leave out to ones that contribute less to the function sum. Neural nets, similarly approximates the function, but by using parameterized functions instead. Both techniques have been combined in the past as "Neural Sparse Grids" or "Deep Sparse Grids" in order to solve the previously mentioned problems.

In this thesis, we will attempt to use Deep Sparse Grids in order to solve a type of PDE, the heat equation, and for a specific case, namely, when the domain of the heat equation is a lower dimensional manifold embedded on a higher dimensional domain. This method will be derived, tested, and discussed.

Contents

Acknowledgments iii

Abstract iv

1 Introduction 1

2 State of the Art 2
2.1 Partial Differential Equations . . . 2
2.1.1 Variables and Initial Conditions . . . 2
2.1.2 The Heat Equation . . . 3
2.1.3 Solving PDEs . . . 4
2.2 Autoencoders . . . 4
2.2.1 Neural Networks . . . 4
2.2.2 Autoencoder . . . 6
2.2.3 Dimensional Reduction of PDEs . . . 6
2.3 Sparse Grids . . . 7
2.3.1 Nodal Basis Functions and Full Grids . . . 7
2.3.2 Hierarchical Basis Functions and Sparse Grids . . . 8
2.3.3 PDE Solving with Sparse Grids . . . 11

3 Solving partial differential equations in high dimensional spaces using Deep Sparse Grids 12
3.1 Solving the 1D Heat Equation . . . 12
3.2 Autoencoders and the 2D Case . . . 14
3.3 Domain Change and its Consequences . . . 17
3.4 Solving the Modified 1D Heat Equation . . . 18
3.5 Deep Sparse Grids and the Code . . . 20
3.6 Experiments and Results . . . 22
3.6.1 Example Problem . . . 22
3.6.2 Fake Encoder . . . 22
3.6.3 Real Encoder . . . 24

4 Conclusion **26**
4.1 Summary 26
4.1.1 Summary 26
4.1.2 Discussion 26
4.1.3 Outlook 26

List of Figures **28**

List of Tables **29**

Bibliography **30**

1 Introduction

Many complex processes in diverse areas of science involve multiple variables, and deal with how these variables change with relation to one another. These complex processes have been modelled and studied for decades, using partial differential equations [Pen05]. As science developed into more and more into more complicated settings, so did the models to study, and thus, the equations. In recent years, with the boom in machine learning and other areas of modelling with high dimensional structure, this increase in complexity has forced the partial differential equations to involve even more variables, and also increase in dimensionality. However, this process has resulted in the emergence of the "Curse of Dimensionality", that is, the higher the dimension of the problem, the more resources needed to solve the problem within the same error margins [Pfl10].

Sparse grids were developed to solve this problem, and in fact, to solve partial differential equations, and they were quite successful [Zen91]. However, as the dimensionality of the data within these problems increases ever more, new methods must be developed to deal with them.

Luckily, in some cases, only the ambient dimension space of the data is high, and can be reduced without most loss. For this purpose, a special kind of neural network is used, the autoencoder. Autoencoders have been used in many diverse areas, to reduce the dimensionality of data, and have proven themselves useful [IC16].

A combination of an autoencoder to reduce the dimensionality of data, and a sparse grid to solve a partial differential equation on the latent space created by the encoder is the main concept developed in this thesis.

Section 2 will introduce the diverse yet interrelated concepts of partial differential equations (PDEs), autoencoders, and sparse grids. Afterwards, Section 3 will introduce the main topic of the thesis, that is, a Deep Sparse Grid structure involving an autoencoder and a sparse grid for solving a specific PDE, namely, the heat equation. We will derive a method to solve the equation using sparse grids, introduce the autoencoder to the system and study its effects, and finally, obtain and discuss results.

2 State of the Art

This section will serve as a brief introduction to the diverse background topics of this thesis, namely: partial differential equations in Section 2.1, autoencoders as a neural network for dimension reduction in Section 2.2, and sparse grids and all related interpolation tools such as hierarchical basis functions in Section 2.3.

2.1 Partial Differential Equations

Partial differential equations, or PDEs for short, are a special type of equation describing specific functions. The "differential" part indicates that such equations are constructed using the derivatives of functions, and the "partial" part indicates that the resulting functions and the relationships involve multiple variables. They are used to model a variety of real world phenomena, finding the solution allows one to understand and predict these phenomena better [Bor16].

2.1.1 Variables and Initial Conditions

In the traditional sense, and also for our purposes, the (multi)variables of the equations are time (t), and some spatial coordinates (x, y, z). So a time-dependent function based in 1 dimensions can be described as $u(t, x)$.

Aside from the equation itself, another important part of PDEs are the initial and boundary conditions. As the solutions of these equations are usually a family of functions, these families generally need to be narrowed down. The most common way of doing this is by using initial conditions, that is, the value of the function at points where t equals zero. Other conditions such as nonzero values or values of derivatives are also common.

Unless stated otherwise, we will use f_x to indicate the partial derivative of a function f with respect to the variable x.

2.1.2 The Heat Equation

An example of a PDE is the so-called heat equation, which can be written as

$$\frac{\partial u}{\partial t} = \alpha \Delta u. \tag{2.1}$$

Here, the Δ stands for the Laplace operator, a sum of all spacial partial second derivatives. In other words, the function u must satisfy the property where its partial derivative with respect to time is equal to the sum of its spacial partial derivatives, times a constant α. In 3 dimensions, the equation would like

$$\frac{\partial u}{\partial t} = \alpha(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}). \tag{2.2}$$

But in this thesis, we will focus on the one dimensional case,

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{2.3}$$

.
The PDE must be solved in a domain, for simplicity, we will assume that

$$u : [0, 1] \to \mathbb{R}. \tag{2.4}$$

Also, as mentioned in the previous subchapter, PDEs are generally to be solved using some conditions. As for the heat equation, we will assume the so-called Dirichlet boundary conditions [Lan15], or

$$u(x, 0) = u_0, \quad u(0, t) = u(1, t) = 0. \tag{2.5}$$

So, we will have some initial input vector u_0, which will describe the state of the domain at $t = 0$. In addition, the two boundary points at the edge of the domain will always be 0.

The heat equation, as can be understood by its name, is a way of simulating the heat flow on some kind of surface or object. For example, the one dimensional heat equation is a way of representing the heat flow on an uniform rod, where α is a product of certain physical coefficients, although the derivation and exact implications of this equivalency will not be the topic of this thesis.

2.1.3 Solving PDEs

There are many ways of solving PDEs. One way is solving the PDE analytically, that is, finding an explicit expression for the solution. However, solving the PDE analytically is not always possible.

In this thesis, we will attempt to solve the heat equation numerically, generally using finite differences. Instead of obtaining a closed form representation of the resulting functions, we will use an algorithm that can generate the values of this function at specific points.

For this, we will use the weak formulation of the heat equation [Glo11], which can be written as

$$\int_\Omega u_t \cdot v dx = \int_\Omega u_{xx} \cdot v dx. \tag{2.6}$$

Here, Ω is our domain, and v is a so-called test function. A solution satisfying this equation for all test functions v will also satisfy the original equation. Using other techniques, this integral equality can be turned into a system of equations, the exact details of which will be explained later.

2.2 Autoencoders

An autoencoder, a type of neural network, is a key component of this thesis, and will be the tool used to reduce the dimensionality of our PDE domains. Before they could be explained in detail, neural networks must be introduced.

2.2.1 Neural Networks

A neural network, or NN for short, is a computational structure consisting of artificial neurons, which are connected to each other in certain way, and all have inputs and outputs in a domain, generally in $[0, 1]$. These neurons are arranged in layers, where the inputs of a neuron consist of all the outputs of the neurons of the previous layer [VU96]. See 2.1 for an example. For the output o_i of a neuron i, we have

$$o_i = a\left(\sum_j w_{ij} o_j + b_i\right). \tag{2.7}$$

Where j are the neurons of the previous layer, w_{ij} are weights of the connections, and b_i is the bias of neuron i. a is an activation function, which keeps the output within the

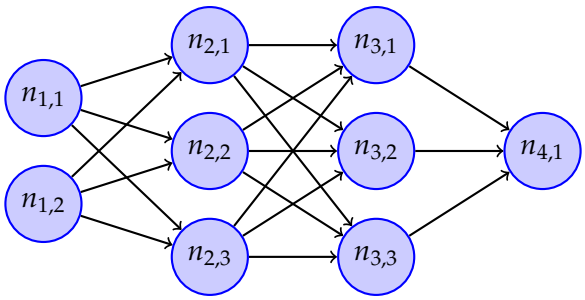


Figure 2.1: A neural network with a 2D input, 1D output, and 2 hidden layers.

interval $[0, 1]$. An example of an activation function is

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

also called the Sigmoid function.

The layers of a neural network consist of an input an layer, an output layer, and the hidden layers. Between each layer, the above calculation occurs, where in the end, we obtain the output of the whole neural network based on the input. This whole process is called a forward pass. The number of neurons in a neural network's input layer is also called its input dimension, and describes the shape of its input. Similarly, a NN also has an output dimension.

The main point of neural networks is that they can be trained to solve a specific problem. The idea is to prepare inputs and correct outputs of the problem, calculate the outputs of the neural network, and then compare these, using a so-called loss function. An example is the Mean Squared Error, which can be calculated as

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (o_i - \hat{o}_i)^2. \tag{2.9}$$

Where n is the number of outputs, o_i is the correct output, and $\hat{o}_i$ is the output of the neural network.

Based on the result of this function, we can adjust the weights and the biases of the neural network to decrease the value of the loss function, and thus reduce the error rate of the neural network. This is done using a technique called back propagation, the details of which will not be explained very in depth in this thesis.

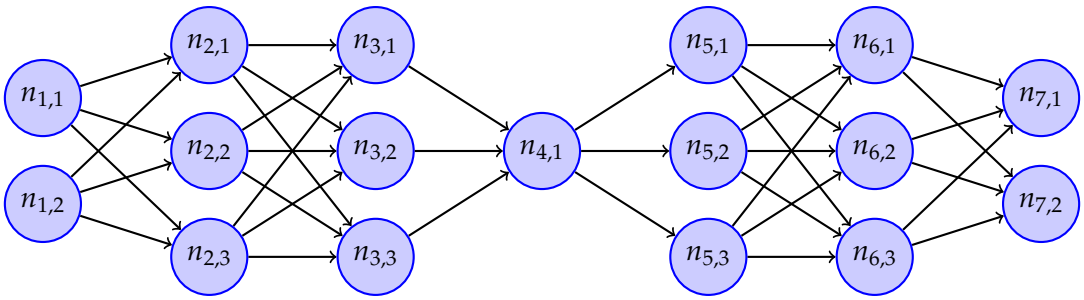


Figure 2.2: An autoencoder with 2D inputs/outputs, and a 1D encoding domain.

For the purposes of building and training a neural network, the Python library TensorFlow will be used [Mar+15].

2.2.2 Autoencoder

An autoencoder is a special type of neural network [IC16]. It consists of two neural networks adjoined, such that the output of the first neural network becomes the input of the second neural network. The input dimension of one NN is equal to the output dimension of the other, and vice versa. This means that the input dimension and the output dimension of the whole structure are the same. See 2.2 for an example.

Autoencoders are used in dimensional reduction [IC16]. For example, an autoencoder can be used to reduce the dimension of an image, to compress it. A monochrome image with 16x16 pixels can be understood as a 256-dimensional data point, which we might attempt to reduce to 64 dimensions, and then reconstruct. So, the input/output dimension of the autoencoder would be 256, and the middle dimension between the NNs would be 64.

The first NN is called the encoder, and the second NN is called the decoder. The encoder can be used to compress the input data and obtain the coded data, whereas the decoder can decompress the code and get the original input. So ideally, the encoder and the decoder should be inverses of each other.

2.2.3 Dimensional Reduction of PDEs

As mentioned in the previous section, PDEs generally involve multivariable functions, another way of imagining the multiple variables is by examining them in multiple dimensions. For example, a heat equation $u(t, x, y, z)$ can be visualized, and explained,

in a three-dimensional space, and also time as a possible fourth dimension.

This approach is useful in specific cases, one that we will consider in this paper is if the interval that the PDE is defined on is a lower-dimensional manifold. What this means is that, the PDE is actually defined on a, for example, one-dimensional interval, but this interval is "twisted" and "moved around" on a 2-dimensional plane. In such cases, reducing the dimensionality of the PDE helps immensely, as it also reduce the work and increases accuracy.

An autoencoder can be used to compress the domain of the original PDE, after which it will be solved on the encoded domain, and then we can decode the solution. The exact process will be explained later.

2.3 Sparse Grids

Sparse grids are the second key component of our PDE solver, and constitute the way in which the PDE will actually be solved. However, nodal / hierarchical basis functions and full grids must be explained before we can get to sparse grids.

2.3.1 Nodal Basis Functions and Full Grids

Let u be a 1-dimensional scalar-valued function defined on the domain $[0,1]$ For simplicity, we will also assume that $u(0) = u(1) = 0$. The goal is to approximate u using a sum of n weighted basis functions, where

$$u(x) \approx \sum_{i=1}^{n} c_i \cdot \phi_i(x) = u'(x). \tag{2.10}$$

Here, ϕ are the basis functions, and c are the weights [Pfl10]. u' is our interpolant. To do this, we divide the domain into mesh-lengths of equal size $h = n^{-1}$.

The same approach can be used for a d-dimensional function, for this, the domain must now be divided into d-dimensional segments. This structure is also called a full grid [Pfl10]. Afterwards, the d-dimensional basis functions would need to be calculated from their one dimensional counterparts. So we would have

$$\phi_{\vec{i}}(\vec{x}) \prod_{j=1}^{d} \phi_{i_j}(x_j), \tag{2.11}$$

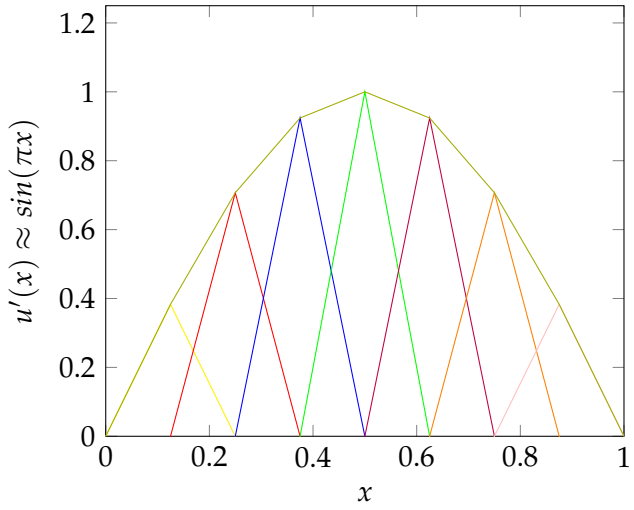


Figure 2.3: Interpolation of the function $sin(\pi x)$ by a nodal basis

and then the interpolant can be constructed as before using

$$u'(\overrightarrow{x}) \approx \sum_{i=1}^{n} c_i \cdot \phi_i(x). \tag{2.12}$$

One possibility is to use nodal basis functions, with

$$\phi_i(x) = 1 - |\frac{x}{h} - i|, \quad c_i = u(\frac{i}{n+1}). \tag{2.13}$$

While these are simple to understand and use, a problem occurs in higher dimensions: The number of points to interpolate increases exponentially with respect to the dimension. This is also called the curse of dimensionality. To combat this, we will introduce hierarchical basis functions and sparse grids.

2.3.2 Hierarchical Basis Functions and Sparse Grids

Before introducing sparse grids, we need to take a look at hierarchical basis functions, which are constructed as

$$\phi_{l,i}(x) = 1 - |2^l x - 2i + 1|. \tag{2.14}$$

So each function has a level $l \in [n]$ in addition to its index $i \in [2^{l-1}]$. The hierarchy comes from the fact that each level is constructed by taking the previous level, and

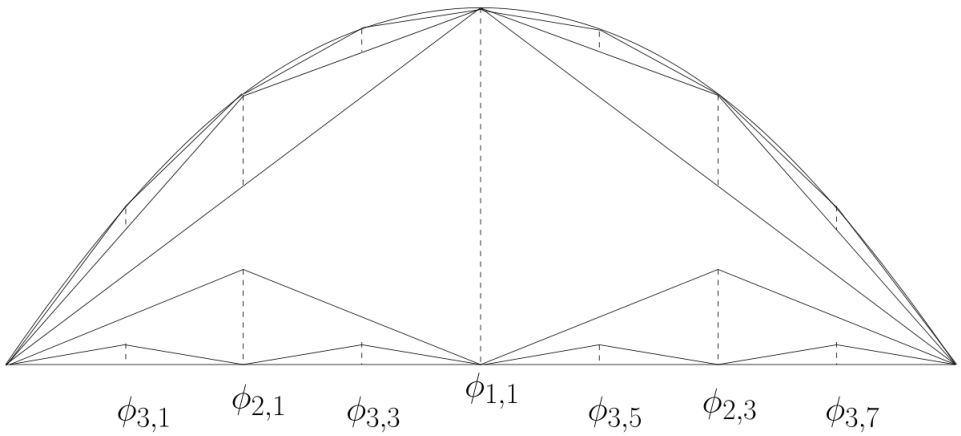


Figure 2.4: Interpolation of a parabola using hierarchical basis functions. [Gar13]

	Grid Points	Error
Full Grids	$\mathcal{O}(h^{-d})$	$\mathcal{O}(h^2)$
Sparse Grids	$\mathcal{O}(h^{-1}\log(h^{-1})^{d-1})$	$\mathcal{O}(h^2\log(h^{-1})^{d-1})$

Table 2.1: Comparison between full grids and sparse grids

dividing the nonzero domain of a basis function into half, where each subdomain gets its own basis function. Note that all possible function constructed from the basis functions (its span) is equal to the one for nodal basis functions [Pfl10]. Also note that the smaller the nonzero domain of a basis function, the less it contributes to the total sum.

In the d-dimensional case, during the construction of the d-dimensional basis functions, we are multiplying hierarchical basis functions of many different levels together. These products can be categorized into multiple subspaces. The idea behind sparse grids is that, since the high-level basis functions contribute less, and thus their products as well, we can ignore them in the final sum. This results in a sparse grid, which can also be represented as a diagonal cut of subspaces (see 2.5).

It can be shown that, while sparse grids have, in comparison with full grids, a logarithmically exponential number and not exponential number of points, the error rate does not change significantly (2.1) [Pfl10].

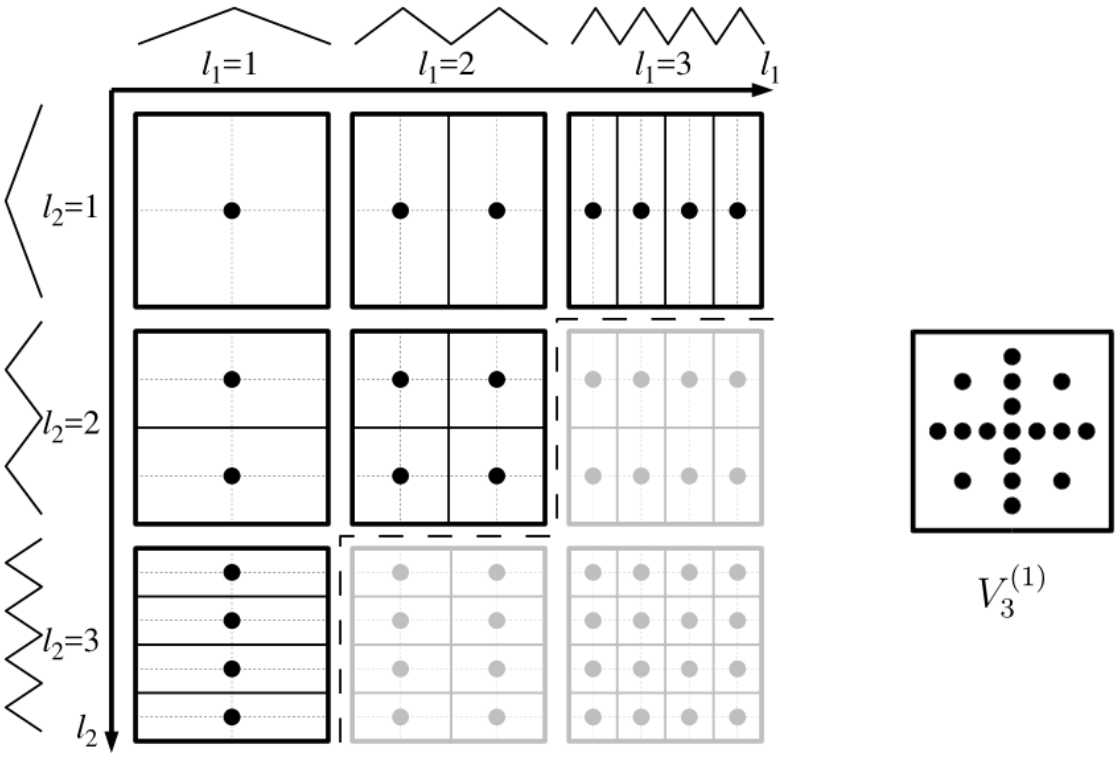


Figure 2.5: Sparse grids are obtained from full grids using a diagonal cut. [Pfl10]

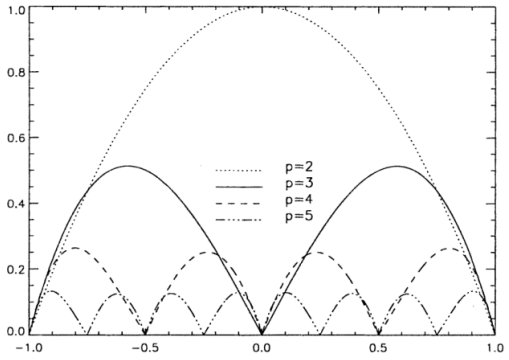


Figure 2.6: Polynomial functions p for basis functions, note that they are not scaled yet. [BD98]

2.3.3 PDE Solving with Sparse Grids

Sparse grids were originally developed to solve PDEs [Zen91]. While they are generally used to solve elliptic partial differential equations such as the Poisson equation [BD98], and not parabolic ones like the heat equation, the main techniques used are still similar, and we will use these in the main part of this thesis.

Since partial differential equations involve derivatives so heavily, in many cases, it is important for the hierarchical basis functions to have derivatives which are continuous and defined for higher than the first, both of which are not the case for the linear basis functions we have defined. To solve this problem, we define the polynomial hierarchical basis functions, which are defined as [Bun98]

$$\phi_{i,l}(x) = \frac{p_{i,j}(x)}{p_{i,j}((2 \cdot i - 1) \cdot 2^{-l})} \tag{2.15}$$

where the polynoms p can be defined recursively as

$$p_{1,1}(x) = x(x-1), \quad p_{i,l}(x) = p_{i',l'}(x) \cdot (x - ((2 \cdot i - 1) \cdot 2^{-l})). \tag{2.16}$$

Here, the terms i' and l' refer to the index and level of the parent basis function, and

$$l' = l - 1, \quad i' = \lfloor \frac{i}{2} \rfloor \tag{2.17}$$

holds.

3 Solving partial differential equations in high dimensional spaces using Deep Sparse Grids

The main idea goal of the thesis, solving a PDE using a Deep Sparse Grid, will be explained in this section. Section 3.1 will devise a way of solving the heat equation using sparse grids. The autoencoder will be added to the solver in Section 3.2, and its effects will be explained in Section 3.3. Due to these effects, a new equation for the sparse grids solver will be created in Section 3.4. Afterwards, Section 3.5 will briefly explain deep sparse grids and the code, and finally, in Section 3.6 the numerical results of an example problem will be discussed.

3.1 Solving the 1D Heat Equation

In the following, we will attempt to construct a method to solve the 1 dimensional heat equation (with Dirichlet boundary conditions) using sparse grids. While the sparsity of the structure will not be useful in the 1 dimensional case, the hierarchical basis functions will be, which can be extended into multiple dimensions using the Up-Down principle [Bun98], which will not be explained in detail here.

The solution to the heat equation must satisfy

$$u_t = \alpha \cdot u_{xx}, \tag{3.1}$$

where α is a constant. For simplicity, we will assume that the function u must be defined on the domain $[0,1]$, if this is not the case, we can easily transform the domain to $[0,1]$ using a linear transformation. We also have the initial condition u_0, and the boundaries are always 0. Note that the following technique is derived from [Glo11].
We start with constructing the weak form of the equation, which also incorporates a weakening function v:

$$\int_0^1 u_t v dx = \int_0^1 \alpha u_{xx} v dx. \tag{3.2}$$

The second step is to divide the interval $[0,1]$ into 2^n meshes of length 2^{-n}, and assume that an interpolant of u constructed using hierarchical basis functions exists, which we will call u':

$$u'(t,x) = \sum_{l=1}^{n} \sum_{i=1}^{2^l-1} c_{i,l}(t)\phi_{i,l}(x). \tag{3.3}$$

Note that the coefficients $c_{i,l}$ only depend on the time, as they are constant for a specific basis function, and as such do not depend on t. Similarly, the basis functions $\phi_{i,l}$ only depend on x. The linearity of the differential operator results in the following partial derivatives of u':

$$u'_t = \frac{\partial}{\partial t}(\sum_{l=1}^{n} \sum_{i=1}^{2^l-1} c_{i,l}(t)\phi_{i,l}(x)) = \sum_{l=1}^{n} \sum_{i=1}^{2^l-1} (\frac{\partial}{\partial t} c_{i,l}(t))\phi_{i,l}(x), \tag{3.4}$$

and

$$u'_{xx} = \frac{\partial^2}{\partial x^2}(\sum_{l=1}^{n} \sum_{i=1}^{2^l-1} c_{i,l}(t)\phi_{i,l}(x)) = \sum_{l=1}^{n} \sum_{i=1}^{2^l-1} c_{i,l}(t)(\frac{\partial^2}{\partial x^2}\phi_{i,l}(x)). \tag{3.5}$$

We can then use these definitions in the integrals of the weak equality, to get

$$\int_0^1 u_t v dx = \int_0^1 (\sum_{l=1}^{n} \sum_{i=1}^{2^l-1} (\frac{\partial}{\partial t} c_{i,l}(t))\phi_{i,l}(x))v dx = \sum_{l=1}^{n} \sum_{i=1}^{2^l-1} (\frac{\partial}{\partial t} c_{i,l}(t)) \int_0^1 \phi_{i,l}(x) v dx, \tag{3.6}$$

and

$$\int_0^1 u_{xx} v dx = \int_0^1 (\sum_{l=1}^{n} \sum_{i=1}^{2^l-1} c_{i,l}(t)(\frac{\partial^2}{\partial x^2}\phi_{i,l}(x)))v dx = \sum_{l=1}^{n} \sum_{i=1}^{2^l-1} c_{i,l}(t) \int_0^1 (\frac{\partial^2}{\partial x^2}\phi_{i,l}(x)) v dx. \tag{3.7}$$

Now, if we choose another hierarchical basis function $\phi_{i',l'}$ for v, and we use integration by parts on the second integral, we obtain the following:

$$\int_0^1 (\frac{\partial^2}{\partial x^2}\phi_{i,l}(x))\phi_{i',l'} dx = (\frac{\partial}{\partial x}\phi_{i,l}(1))\phi_{i',l'}(1) - (\frac{\partial}{\partial x}\phi_{i,l}(0))\phi_{i',l'}(1) - \int_0^1 (\frac{\partial}{\partial x}\phi_{i,l}(x))(\frac{\partial}{\partial x}\phi_{i,l}(x)) dx. \tag{3.8}$$

Since $\phi_{i',l'}$ was a basis function, its value is 0 on the boundaries, and so we only have the final integral remaining.

Finally, the end result of the equation is as follows:

$$\sum_{i,l} \frac{\partial c_{i,l}(t)}{\partial t} \int_0^1 \phi_{i,l}(x)\phi_{i',l'}(x) dx = -\alpha \sum_{i,l} c(t) \int_0^1 \frac{\partial \phi_{i,l}(x)}{\partial x} \frac{\partial \phi_{i',l'}(x)}{\partial x} dx. \tag{3.9}$$

With an ordering of hierarchical basis functions j, this equation can be represented in matrix form as

$$c_t \cdot I_{Stiff} = c \cdot I_{Mass}. \tag{3.10}$$

With the so-called stiffness and mass matrices defined as [BD98]

$$I_{Stiff}(j, j') = \int_{\Omega_{i,l} \cap \Omega_{i',l'}} \phi_{i,l}(x) \phi_{i',l'}(x), \tag{3.11}$$

and

$$I_{Mass}(j, j') = \int_{\Omega_{i,l} \cap \Omega_{i',l'}} \frac{\partial \phi_{i,l}(x)}{\partial x} \frac{\partial \phi_{i',l'}(x)}{\partial x} dx. \tag{3.12}$$

Note the integral boundaries, per definition of hierarchical basis functions, most functions will not have a common domain, and the integral will evaluate to zero.

Now, if we use the polynomial basis functions introduced earlier, then the derivatives and integrals will be trivial to compute, and the matrices describes above can be computed easily. This computation is implemented in the code using the Polynomial library from NumPy.

In the end, the equation describes a linear relationship between a n-dimensional function $c(t)$ and its derivative $c_t(t)$, and thus, can be as an ordinary differential equation, the details of which would be too large for the scope of this thesis. We simply solve this equation by using a least squares solver from NumPy, which calculates the correct coefficients at certain times, and uses the coefficients $c(0)$ of the initial condition while doing so.

3.2 Autoencoders and the 2D Case

We now consider a 1 dimensional domain embedded on a higher dimensional domain, for example, 2. Such an embedding is also called a manifold. Intuitively, this can be understood as a line on a plane, for example, a chalk drawing of a line on a blackboard.

Our plan would be, then, to solve the heat equation on this domain, as opposed to the simple 1 dimensional case from the previous chapter. While it is possible to solve the heat equation on the 2 dimensional plane, in fact, using a method similar to the previous one, by using the hierarchical structure of sparse grids; since the underlying structure is actually one dimensional, another, simpler method could be possible using

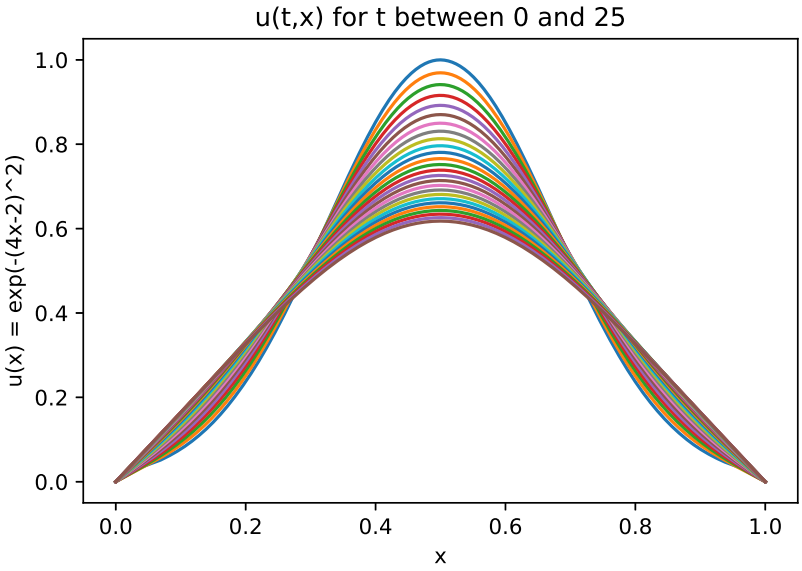


Figure 3.1: Solution of the heat equation used to predict some values, with the initial condition $e^{-(4x-2)^2}$

this one dimensional structure could be possible. Especially considering the case of potentially higher dimensional problems, this reduction becomes crucial.

First of all, since the object is actually one dimensional, there must be a mathematical function E, that can map the 2 dimensional points of the manifold to some 1 dimensional space, and a function D that must do the inverse. Specifically, let $X \subset \mathbb{R}^2$ be the original domain, and $Y \subset \mathbb{R}$ be some encoded domain, then we have, for all $(x_1, x_2) \in X$:

$$E(x_1, x_2) \in Y, \quad D(E(x_1, x_2)) = (x_1, x_2)$$

So the functions E and D must be inverses of each other, and E and D must both be bijective. Also, we require both functions be continuous.

After finding such functions E and D, we can map the initial conditions to the encoded domain using the function E, solve a modified problem on this domain, then transform them back using D, and we would be done. However, we must first find such functions.

For an example of one such function, consider the arc length of the curve, that is, the length of the curve between the point to encode, and a start point. Since each point

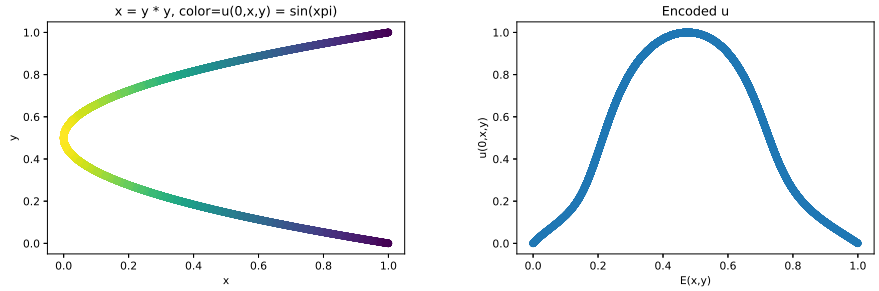


Figure 3.2: The initial conditions on a manifold on 2D, and the domain encoded on 1D using an autoencoder.

on the manifold can be uniquely identified by its arc length, we can choose for E the function that calculates the length, and for D its inverse. The problem, however, is that these functions are not easy to define. For example, the arc length of the curve of a function f between the x values a and b can be calculated as

$$s = \int_a^b \sqrt{1 + f'(x)^2} dx. \tag{3.13}$$

However, our curve must not be a function, nor should the derivative be easily calculable, so it would not be as easy to define a E. And even after coming up with a way to compute E efficiently, D must also be formed.

Instead, for the purposes of creating functions E and D, an autoencoder can be used. As explained in the State of the Art section, an autoencoder already consists of two neural networks, the encoder and the decoder, that reduce and increase the dimension of input data respectively. For this, we would need to set the dimensions of the input/output to our original domain, which is 2, and the encoded dimension to the dimension of the manifold, which is 1. The rest of the variables such as number of layers or neurons per layer can be chosen as one desires, but will have consequences, which will be explained in a little bit.

In order to produce the desired functions E and D, the autoencoder must be trained. To do this, the autoencoder must be fed a set of points on the manifold, which will produce outputs. Depending on the difference between these outputs and the inputs, the weights and biases of the artificial neurons of the autoencoders will be adjusted, and in the end of the training process, the encoder will serve as E, and the decoder as D.

Note that, since the functions are the result of training, and not analytical calcula-

tions, they will not be perfect. Specifically, D will not be the inverse of E, but only approximate it. The level to which this approximation is correct can be increased by many ways, but usually at a trade-off: For example, one way is to increase the number of points to train the autoencoder, but then the training will take more time, or it might not be that easy to generate many points in certain cases. Similarly, more layers or neurons per layers may also make the Encoding and Decoding functions more accurate, however, more training would then be needed to fine tune these parameters, thus again, increasing time or resources.

3.3 Domain Change and its Consequences

Now, as a reminder, we have our original 2D domain X, and the new, 1D code domain Y. Our goal was to solve the heat equation, formulated on X, instead on Y. The naive approach would be, then, to use the exact method introduced in Section 2.1 on Y, however, this is incorrect. The problem is that, while this does solve the heat equation on Y, there is no guarantee that it will also solve it on X.

The reason for this discrepancy lies within the functions E and D. While converting one domain to the other, these functions inevitably change the stretch and warp the points. For example, two sets of points that had different distances between them in the 2D domain, might have the same distance in the 1D domain. This results in the need of a different PDE to solve in the encoded domain [BD98].

Once this point is formulated, afterwards, we will no longer consider the 2D-1D autoencoder, but instead a 1D-1D autoencoder. While a 1D-1D autoencoder, that is, an autoencoder that does not actually reduce a dimension at all, might not be a useful idea at first sight, it is nonetheless a very useful concept to understand the underlying problems and combat them, as will be seen in the next section. Moreover, a 1D-1D autoencoder still causes the aforementioned changes between the domains, despite the fact that both domains have the same dimension. In fact, most research on this subject has been done on transformations between domains of the same dimensionality [BD98].

Note that, while the resulting PDE on the new domain will be different from the heat equation, it should still be similar to it, in fact, it must equal the heat equation for the specific case of $E(x) = x$, as that would mean that solving the equation on X is no

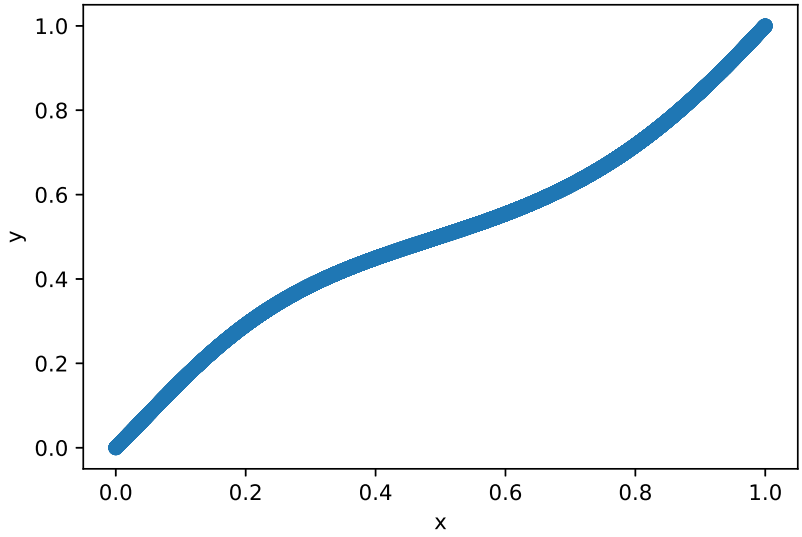


Figure 3.3: Example relationship between values in input domain X and coded domain Y, it is not linear.

different than solving it on Y.

3.4 Solving the Modified 1D Heat Equation

In the following, we will attempt to construct and solve a PDE on a domain changed by encoding and decoding functions, where the solution of the PDE decoded on the input domain will solve the heat equation.

Let E be the encoding function, and D the decoding function. Furthermore, let u be our original function as it was defined on the input domain X, we have the function v defined on the encoded domain Y as

$$v(t, y) = u(t, D(y)). \tag{3.14}$$

Our original function must fulfill the heat equation, so we have

$$\frac{\partial u(t, D(y))}{\partial t} = \alpha \frac{\partial^2 u(t, D(y))}{D(y)^2}. \tag{3.15}$$

Since we are changing the domain of x/y and not the time t, the left side is simply equal to v_t. On the right side, we can use the chain rule:

$$\frac{\partial^2 u(t, D(y))}{D(y)^2} = \frac{\partial}{\partial D(y)}(\frac{\partial u(t, D(y))}{\partial D(y)}) = \frac{\partial}{\partial D(y)}(\frac{u_y(t, D(y))}{D_y}) = \frac{D_y v_{yy} - D_{yy} v_y}{(D_y)^3}. \quad (3.16)$$

The PDE can then be redescribed by using

$$D_y v_t = \alpha \frac{\partial}{\partial y}(\frac{v_y}{D_y}). \quad (3.17)$$

The term D_y, to explain in words, is the partial derivative of the decoding function with respect to time t. It can be calculated using a GradientTape from Tensorflow, which is implemented in the code.

The techniques used in solving the 1D heat equation in the first subsection can then be used to get the following weak formulation equation:

$$\sum_{i,l} \frac{\partial c_{i,l}(t)}{\partial t} \int_0^1 D_y(y) \phi_{i,l}(y) \phi_{i',l'}(y) dy = -\alpha \sum_{i,l} c_i(t) \int_0^1 \frac{1}{D_y(y)} \frac{\partial \phi_{i,l}(y)}{\partial y} \frac{\partial \phi_{i',l'}(y)}{\partial y} dy. \quad (3.18)$$

Which results in the same equation as the one for the simplified case, but with the stiffness/mass matrices:

$$I_{Stiff}(j, j') = \int_{\Omega_{i,l} \cap \Omega_{i',l'}} D_y(y) \phi_{i,l}(y) \phi_{i',l'}(y) dy, \quad (3.19)$$

and

$$I_{Mass}(j, j') = \int_{\Omega_{i,l} \cap \Omega_{i',l'}} D_y(y) \frac{\partial \phi_{i,l}(x)}{\partial y} \frac{\partial \phi_{i',l'}(x)}{\partial y} dy. \quad (3.20)$$

While it is possible to explicitly calculate these integrals, for example by using methods from NumPy or TensorFlow, since we are already discretizing the domain using meshes, we can also discretize the domain for the function D_y and redefine the matrices as

$$I'_{Stiff}(j, j') = \sum_{k<2^l} \frac{D_y(kh+1) - D_y(kh)}{2} \int_{kh}^{kh+1} \phi_j(y) \phi_{j'}(y) dy, \quad (3.21)$$

and

$$I'_{Mass}(j, j') = \sum_{k<2^l} \frac{2}{D_y(kh+1) - D_y(kh)} \int_{kh}^{kh+1} \frac{\partial \phi_j(y)}{\partial y} \frac{\partial \phi_{y'}(y)}{\partial y} dy. \quad (3.22)$$

Now the integrals are again of polynoms, and can be computed easily. Afterwards, solving the PDE follows exactly as it was in the 1D normal case, with the exception of

having to encode the initial conditions.

As mentioned in the previous section, the resulting PDE is indeed similar to our original heat equation, with the only difference being the terms D_y. Also, in the case of the identity encoder $E(x) = 1$, the derivative of the decoder equals 1, so we do obtain the classical heat equation.

Note that, in the case of an autoencoder with multidimensional inputs and outputs, the equation would look different. Specifically, the partial derivative D_y would be a multidimensional vector as well, a Jacobian to be specific. But actually constructing and solving the resulting PDE is not within the scope of this thesis.

3.5 Deep Sparse Grids and the Code

Deep / Neural Sparse Grids, or DSGs / NSGs for short, are a type of combination of neural networks and sparse grids [Zha21]. The exact nature of this structure can differ, for example, the sparse grid can be embedded within the Neural Network, or appended to the end of it, however, the core principle of the structure remains the same in each use case: To use both methods in unison, in order to be able to use each methods strengths, and remedy their weaknesses. Our use case is also similar, where we are using the autoencoder's strength of dimensional reduction, and the sparse grid's strength in solving PDEs reliably. As such, our structure would also qualify as a Deep Sparse Grid.

Deep Sparse Grids have been previously used to tackle problems such as regression, classification, and image recognition, with varying success. In contrast to these problems, which are classical problems in Neural Network research, the problem we are dealing has to do more with sparse grids, as such, the implementation of a Sparse Grid PDE Solver is more prominent in the codebase, where the implementation of the autoencoder is more by-the-books and not particularly innovative.

Due to the difference in nature with other Deep Sparse Grid projects, their codebase was not used, instead, most sparse grids code were written anew, see the code on the next page for an example. And, since the problems that were dealt with were 1 dimensional in nature, most algorithmic classics such as the Up-Down principle or the sparse structure of the grids themselves were not relevant, however, the code itself was written in a scalable and open-ended way, to allow to implement these features in the future.

Listing 3.1: Python example

```
#array of polynomial basis functions
polys = [poly.Polynomial([0, -1, 1])] * (nx - 2)
for i in range(spl - 1):
    for j in range(nx - 2):
        if (j + 1) % (2 ** (spl - 1 - i)) != 0:
            polys[j] *= poly.Polynomial([-2 ** (-i - 1) *
            (2 * int((j + 1) / (2 ** (spl - i))) + 1), 1])
for i in range(nx - 2):
    polys[i] *= poly.Polynomial([1 / polys[i](2 ** -spl * (i + 1))])

#function for calculating interpolant
def sparse(x, l):
    if x >= 1 or x < 0:
        return 0
    ret = 0
    for i in range(l):
        ind = int(2 ** i * x) * 2 ** (l - i) + 2 ** (l - i - 1) - 1
        ret += polys[ind](x) * ct[ind]
    return ret

#adjusting weigths
ct = ux0[1:(nx - 1)]
for i in range(spl - 1):
    for j in range(2 ** (i + 1)):
        ind = j * 2 ** (spl - i - 1) + 2 ** (spl - i - 2) - 1
        ind2 = 2 ** (spl - 1) - 1
        for k in range(i + 1):
            ct[ind] -= polys[ind2]((ind + 1) * dx) * ct[ind2]
            if ind2 > ind:
                ind2 -= 2 ** (spl - k - 2)
            else:
                ind2 += 2 ** (spl - k - 2)

sg_dx = 0.001
sg_nx = int(1 / sg_dx) + 1
sgx0 = np.asarray([sparse(0 + i * sg_dx, spl) for i in range(sg_nx)])
```

3.6 Experiments and Results

In this section, we will how the Deep Sparse Grid PDE Solver behaves based on a specific example, the accuracy of solutions will be discussed based on error rates.

3.6.1 Example Problem

As an example for solving the heat equation, we consider the following initial condition:

$$u(0, x) = sin(\pi \cdot x). \tag{3.23}$$

on the domain $[0, 1]$. Also, let $\alpha = 0.001$.

The reasoning behind choosing this example is that an analytical solution exists:

$$u(t, x) = sin(\pi \cdot x) \cdot e^{\alpha \cdot \pi^2 \cdot x}. \tag{3.24}$$

We want to calculate $u(t', x')$ at $t' \in \{10, 20, 30, 40, 50\}$ and $x' = 2^{-3} \cdot i$, $i \in \mathbb{N}$, $x' \in [0, 1]$. In other words, we will be using hierarchical basis functions with level up to 3.

3.6.2 Fake Encoder

First, we consider a fake encoder, which was not trained, yet still warps the input domain. The encoding and decoding functions are defined as

$$E(x) = \sqrt{2x + 0.25} - 0.5, \tag{3.25}$$

and

$$D(y) = \frac{y(y+1)}{2}. \tag{3.26}$$

Note that D_y is never zero, so the necessary matrices are always calculable.

The results can be seen in 3.5 and 3.6. The solution that was calculated seems to match the analytical solution almost perfectly, despite the relative large size of t. Also, it can be seen that, as t increases, so does the relative error. Similarly, a higher level of basis functions has an inverse correlation with the error rate.

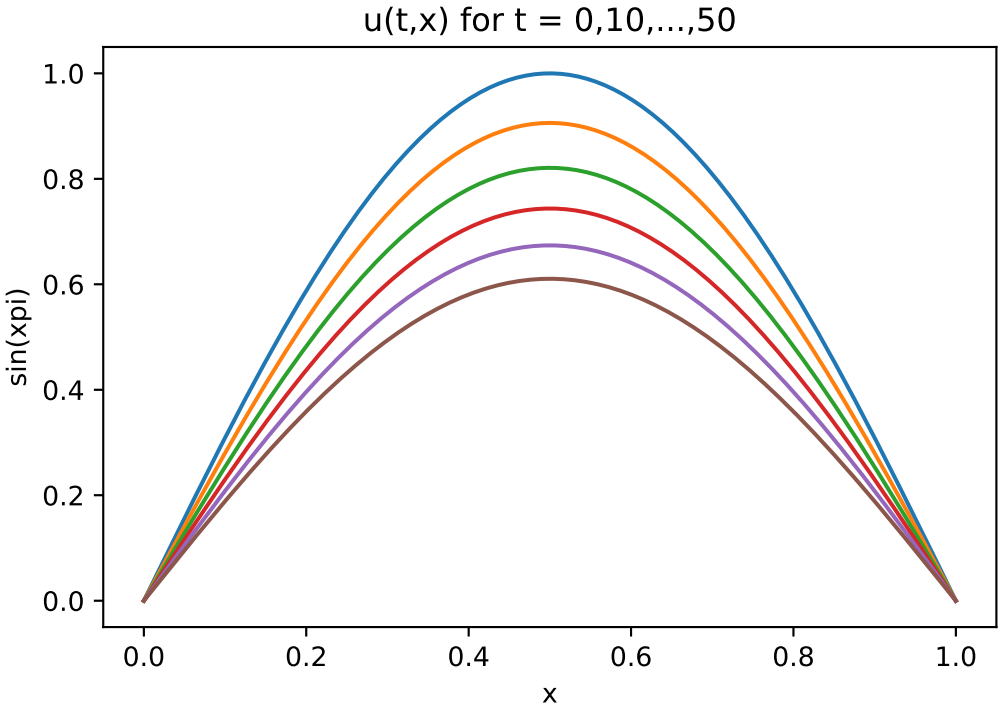


Figure 3.4: Analytical solution of the heat equation for the given example.

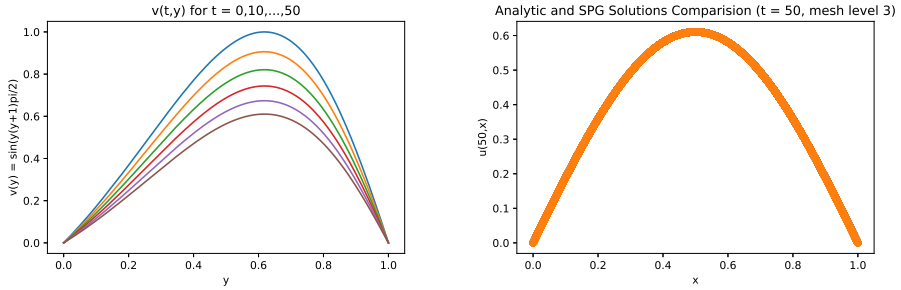


Figure 3.5: Solution of the first case on the encoded domain, and comparison of final solution with actual solution for $t = 50$.

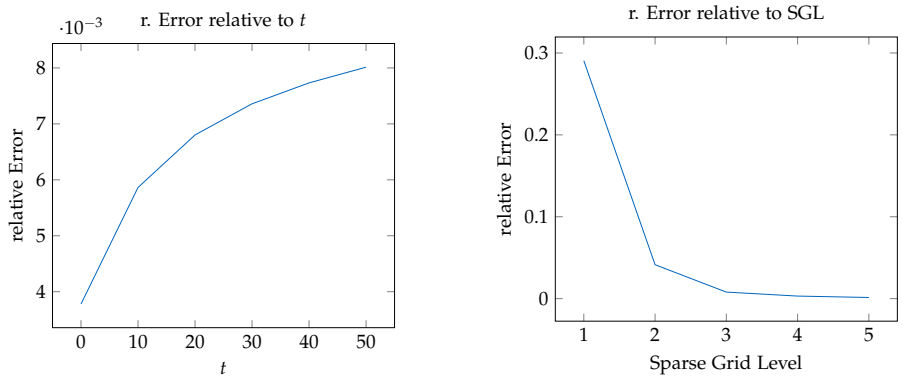


Figure 3.6: Relative error rates in relation to t with a sparse grid level of 3, and relative to sparse grid level with $t = 50$.

3.6.3 Real Encoder

Now, we train an actual autoencoder. For this, 10000 random points were generated as initial conditions. 7000 were used to train the autoencoder, and the rest was used for testing. The autoencoder had 3 hidden layers between the input and the encoded layer, as well as between the encoded layer and the output layer, and these layers consisted of 16 neurons each. As an activation function, the Sigmoid function was used at the end of each layer. The training rate was 0.01, the optimizer was Adam, and the loss function was Mean Squared Error.

After 8 epochs of 216 examples each, the autoencoder reached a loss rate $1.96 \cdot 10^{-4}$. In contrast to the previous example, this autoencoder was not perfect, and the decoding function was not the exact inverse of the encoding function, which was the cause of this loss rate. Note that the encoder and decoder functions were also normalized to fully contain the domain $[0, 1]$ with the following formula:

$$E'(x) = \frac{E(x) - E_{min}}{E_{max} - E_{min}}. \tag{3.27}$$

The formula for the decoder is the analogous. Here, E_{max} refers to the highest value the encoding function has given to a point, and E_{min} the lowest. This normalization ensures that E'_{max} becomes 1, and E'_{min} becomes 0.

Also, in order to calculate D_y, the GradientTape method mentioned earlier was used, whereas in the previous case, D_y was exactly known and calculated analytically.
The results can be seen in 3.7 and 3.8 in contrast to the previous case, the error rate

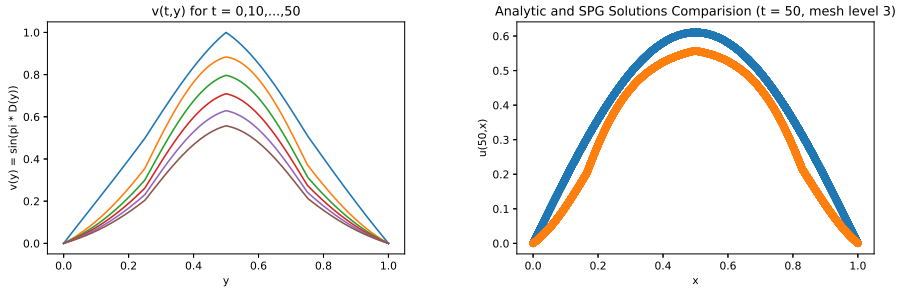


Figure 3.7: Solution of the second case on the encoded domain, and comparison of final solution with actual solution for $t = 50$.

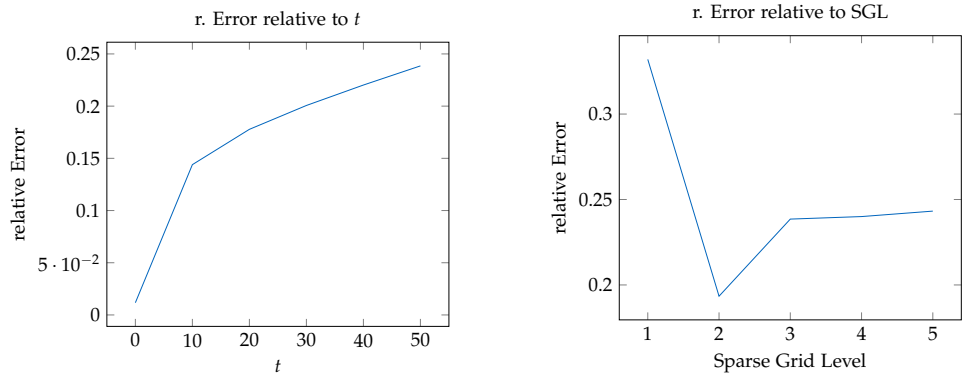


Figure 3.8: Relative error rates in relation to t with a sparse grid level of 3, and relative to sparse grid level with $t = 50$.

seems to be much higher. Furthermore, while the relative error rate does increase with t like the previous case, the relation the sparse grid level seems to be all over the place, with decreasing first, and afterwards increasing. This instability can maybe be attributed to the imperfection of the encoding and the decoding functions, or maybe an error in the normalization process. Further study and analysis is required to understand the exact reason of this discrepancy.

4 Conclusion

4.1 Summary

In this section, we will summarize the previous sections, discuss the results obtained in Section 3.6, and list some possible additions to the project in the future.

4.1.1 Summary

This Bachelor's thesis introduces a method with which an autoencoder and a sparse grid could be combined as a Deep Sparse Grid for the purposes of solving partial differential equations, specifically the heat equation, all of which were introduced in section 2. Section 2.1 defined and explained partial differential equations, Section 2.2 dealt with neural networks and autoencoders, finally, Section 2.3 described the underlying principles of hierarchical basis functions and sparse grids. In Section 3.1, we devised a way of solving a specific PDE, the heat equation, using Sparse Grids. In Section 3.2, we introduced the Autoencoder to the system, and explained the resulting changes in Section 3.3, with which we have developed a different PDE in Section 3.4. Section 3.5 included a brief detour to Deep Sparse Grids and our codebase, and lastly, Section 3.6 described the results of these techniques with an example.

4.1.2 Discussion

The topic of solving the heat equation using sparse grids was not particularly popular among sparse grid research compared to other equations, and the involvement of neural networks such as autoencoders was even less prominent. With this thesis, an attempt was made to combine these topics, and a technique was devised. In the end, the codebase is capable of solving the one dimensional heat equation, even despite interference of domains by an autoencoder, albeit admittedly, with high error rates in the case of trained autoencoders.

4.1.3 Outlook

There are many options to further develop the codebase for solving partial differential equations using deep sparse grids. One possibility is to expand the partial differential

equation solver to solve other well-known equations, especially elliptic equations such as the Poisson equation, as sparse grid research involving them are already quite common. Another possibility would be to increase the dimensions of the input and encoded domains. Autoencoders and sparse grids both shine when working on domains higher than one dimensional, so it would make sense to adapt the code in this direction. Finally, sparse grids also have interesting techniques developed for them, such as the combination technique, or adaptivity, which may be integrated into the codebase.

List of Figures

2.1 A neural network with a 2D input, 1D output, and 2 hidden layers. . . 5
2.2 An autoencoder with 2D inputs/outputs, and a 1D encoding domain. . 6
2.3 Interpolation of the function $sin(\pi x)$ by a nodal basis 8
2.4 Interpolation of a parabola using hierarchical basis functions. [Gar13] . 9
2.5 Sparse grids are obtained from full grids using a diagonal cut. [Pfl10] . 10
2.6 Polynomial functions p for basis functions, note that they are not scaled yet. [BD98] 11

3.1 Solution of the heat equation used to predict some values, with the initial condition $e^{-(4x-2)^2}$ 15
3.2 The initial conditions on a manifold on 2D, and the domain encoded on 1D using an autoencoder. 16
3.3 Example relationship between values in input domain X and coded domain Y, it is not linear. 18
3.4 Analytical solution of the heat equation for the given example. 23
3.5 Solution of the first case on the encoded domain, and comparison of final solution with actual solution for $t = 50$. 23
3.6 Relative error rates in relation to t with a sparse grid level of 3, and relative to sparse grid level with $t = 50$. 24
3.7 Solution of the second case on the encoded domain, and comparison of final solution with actual solution for $t = 50$. 25
3.8 Relative error rates in relation to t with a sparse grid level of 3, and relative to sparse grid level with $t = 50$. 25

List of Tables

2.1 Comparison between full grids and sparse grids 9

Bibliography

[BD98] H.-J. Bungartz and T. Dornseifer. "Sparse Grids: Recent Developments for Elliptic Partial Differential Equations." In: *Multigrid Methods V* 3 (1998), pp. 45–70.

[Bor16] D. Borthwick. *Introduction to Partial Differential Equations*. 2016.

[Bun98] H.-J. Bungartz. *Finite elements of higher order on sparse grids*. 1998.

[Gar13] J. Garcke. "Spatially Adaptive Sparse Grids for High-Dimensional Problems." In: *Lecture Notes in Computer Science and Engineering* 8 (2013), pp. 57–80.

[Glo11] M. S. Glockenbach. *Partial Differential Equations. Analytical and Numerical Methods*. 2011.

[IC16] Y. B. Ian Goodfellow and A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[Lan15] H. P. Langtangen. *The 1D Diffusion Equation*. 2015. URL: `https://hplgit.github.io/num-methods-for-PDEs/doc/pub/diffu/sphinx/._main_diffu001.html`.

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

[Pen05] B. Pentenrieder. "Finite Element Solutions of Heat Conduction Problems in Complicated 3D Geometries Using the Multigrid Method." In: (2005).

[Pfl10] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. 2010.

[VU96] R. D. de Veaux and L. H. Ungar. "A Brief Intro duction to Neural Networks." In: (1996).

[Zen91] C. Zenger. "Parallel Algorithms for Partial Differential Equations." In: *Notes on Numerical Fluid Mechanics* 31 (1991), pp. 241–251.

[Zha21] Z. Zhang. "Neural Sparse Grids for High-Dimensional Data." In: (2021).