Computational Science and Engineering

Technische Universität München

Master's Thesis

Auto-Encoder Sparse Grids

Akhil Nasser

Computational Science and Engineering

Technische Universität München

Master's Thesis

Auto-Encoder Sparse Grids

Author:	Akhil Nasser
1st examiner:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Dr. Felix Dietrich
Submission Date:	May 9th, 2021

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

May 9th, 2021 Akhil Nasser

Acknowledgments

My work on this Thesis was greatly facilitated with the help of certain key individuals. Firstly, I would like to graciously thank my supervisor Dr. Felix Dietrich. As my Supervisor he was of great help to me as he guided me along my Thesis with his extensive subject knowledge and the practical aspects of doing a Thesis. I would like to thank my fellow student Zhen Zhang for our discussions that have helped me as I worked on the Thesis. I would also like to thank Paul Subaru and Michael obersteiner who have given valuable insights into the topics comprising this Thesis. Finally, I would like to thank my family who have been a constant source of support in my research endeavours for this Thesis.

"[T]he greatest benefit of machine learning may ultimately be not what the machines learn but what we learn by teaching them."

-Pedro Domingos

Abstract

Deep Learning-based approaches have become very popular across many fields due to their versatility and their ability to model complex phenomena. Deep Learning based models have far outperform traditional Machine Learning based models and techniques in many tests such as the ImageNet competition. A popular network configuration is the Autoencoder that essentially captures the latent space representation of higher dimensional functions. The Autoencoder consists of two neural networks that work in tandem to minimize an objective function.

Sparse Grids are a mathematical technique that help us to deal with the 'curse of dimensionality' that is associated with high dimensional functions. Traditional Cartesian coordinate based grid system i.e the full grid becomes too computationally expensive and memory intensive for representation/interpolation of high dimensional functions. Sparse Grids also perform a similar function as an Autoencoder but whereas in Neural Networks the underlying basis functions are non-linear and intractable, in sparse grids, this is not necessarily the case.

This thesis aims to demonstrate the feasibility of using Sparse Grids as components of 'neural networks'. This is accomplished in the case of the Sparse Grid Autoencoder(SGA) and the Sparse Grid Variational Autoencoder (SGVA) with the help of an example. A comparative analysis is performed with the traditional Autoencoder network architecture.

Contents

List of Codes

List of Figures

1. Introduction

The world today produces an astounding amount of data every second due to advances in technology like the Internet and the abundance of monitoring devices. In this situation of an abundance of data there is a growing interest from governments and companies to make sense of the data to extract valuable insights that could further help in improving a product or service. This has started a ”data race” to collect even more data and also to develop techniques to interpret this data. This is clearly evident in the requests for cookie/app permissions and the total penetration of advertisements throughout the Internet based on the browsing activities of potential customers. The rise in job postings looking for data scientists/analysts is also an indicator of this movement towards a data-driven world.

The vast amounts of data that are being collected are very often high-dimensional i.e having many attributes, many that are often redundant or not interesting. This makes the downstream processing of data very cumbersome and expensive. Thus, researchers have devoted a significant amount of time and effort to reduce the dimensionality of the collected data. The Mathematical models such as Clustering Algorithms, Support Vector Machines, Principal Component Analysis and others have emerged out of this research. These algorithms perform very well in dimensionality reduction and are still predominantly used in many applications but they have some limitations. Their greatest limitation is their inability to handle large datasets that are becoming prevalent now. This handicap was overcome with the advent of Deep Learning. Deep Learning performs very well on large data sets and also works well with many different types of data like audio, images, graphs, videos and others. Deep Learning models are able to do what used to take human interpretation and many different methods in a single end-to-end model. The incorporation of Robustness into Deep Learning models has also allowed for the models to learn from corrupted data as well as data with large natural variations.

In this race to develop better and better models, people across the world are trying different architectures of Networks incorporating many new feature maps. This is shown by the continuous improvements in Convolutional Neural network architectures to improve both accuracy on predefined datasets and reduce the number of parameters used to get there. Expansion into the areas of graphical data, image segmentation, Object detection, Machine translation, data generation have seen the rise of Networks such as LSTM, GAT, GCN, U-Net, GAN, DCGAN, WGAN, CycleGAN, Transformers, RBM, Autoencoders and many more. We too look at ways to improve existing architectures using a new set of ideas.

Motivation

One class of Deep Learning Networks is the Autoencoder. The Autoencoder is neither a Classifier nor a Generative Network. Classifier Networks are those that are, as the name suggests able to label/classify data after adequate training. Generative Networks are Networks that are able to generate new data that have common underlying functional relationships similar to the training data. The Autoencoder is a Network that learns the latent representations of the input data. Autoencoder Networks have achieved great results in the field of unsupervised learning. This has been successfully used in the application areas of feature extraction, image compression, image denoising and most importantly dimensionality reduction. A related Network to the Autoencoder is the Variational Autoencoder which is a generative Network. It can generate new samples similar to the input after learning the hidden patterns in the training data. Variational Autoencoders have been used in similar applications as the Autoencoder and in word interpolations, generate more training data or sequences. This data generation ability can greatly expand the collection of data for further processing. This is especially useful in Medical applications where there is a dearth of data.

The main motivation behind the Sparse Grid Autoencoder is to develop a novel architecture for an Autoencoder that is more Human-Interpretable while trying to be more efficient with respect to memory storage and computational complexity. A great challenge with Deep Learning Networks in general is their inscrutable manner of operation which leads to decreased confidence by industry professionals in the results produced by such mathematical tools. This is partially solved in the case of the Autoencoder due to the ability to sample the entire latent space thus learning the exact nature of features being embedded. The composition of non-linear functions in traditional neural network architecture make human-explainable AI a challenging task. The other advantage is that studies by [16] have shown that for dimensionality reduction of data with moderate dimensionality, Sparse Grids are far more efficient than Feed forward neural networks. This thesis hopes to introduce this novel idea of using Sparse Grids as the Layers of a Neural Network. This is a first-of-a-kind approach towards designing such a Network.

This Thesis is structured as follows. Section 2 is a broad overview of the fields of Sparse Grids and Deep Learning. Section 2.1 explains the basic concepts and history of Sparse Grids. Section 2.2 does the same for Deep Learning. This Section is meant to give the necessary background information for understanding the Thesis. Section 3 delves into the core ideas of the Thesis. The subsection 3.1 discusses the Methodology behind the Sparse Grid Autoencoder and the Sparse Grid Variational Autoencoder. Section 3.2 presents how the architecture models for these Networks was constructed. Section 3.3 details the implementation of these Networks. Section 3.4 describes the results that were obtained by these Networks on a problem. It also discusses the Hyper-Parameter tuning of these models. Finally, Section 4 presents the Conclusions of the Thesis and also gives some ideas for future work.

2. State of the Art

2.1. Sparse Grids

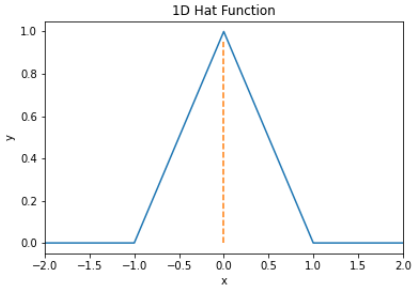
Sparse Grids arouse out of the need to address the ”curse of dimensionality” that was encountered when solving PDE in high dimensional spaces. The use of Finite Difference Methods to solve these PDE on uniform grids led to complexities of the order of $O(N^d)$ where N is uniform number of discretized points in a single dimension. N is set depending on the how accurate the predicted function should be to the true solution. As can be clearly seen the exponential rise of complexity with the number of dimensions meant that for cases beyond $d = 4$ the calculations become very expensive. It is in this context Sparse Grids were introduced.

The full grid or regular grid is the space discretized by points in regular intervals (See Fig 2.1b). Each grid point has a function defined with respect to it. This function is typically a linear hat function defined as follows in the one-dimensional example.

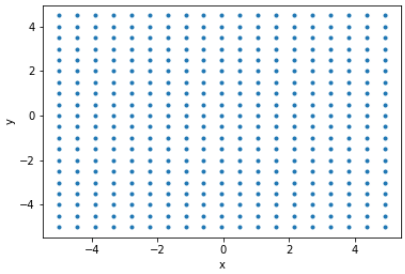
$$\phi_i(x) = \begin{cases} 1 - |x - x_i|, & \text{for } x_{i-1} \leq x \leq x_{i+1} \\ 0 & \text{else} \end{cases} \tag{2.1}$$

Here x is the co-ordinate of any given point and x_i,x_{i-1} and x_{i+1} are grid points.

Figure 2.1a shows a simple one-dimensional hat function defined with respect to the point $x = 0$ with a support length of 2. This hat function is now extended to all points on the full grid. The result of this is shown in 2.2. The Figure 2.2a shows the individual hat



(a) A Simple One-Dimensional hat function



(b) A Simple Full Grid on the x-y plane

Figure 2.1.: Basic Grid Setup

functions overlapping each other with the dotted lines showing the point around which each of these functions are defined. The effective function looks as shown in 2.2b. In higher dimensions we perform tensor products to obtain linear basis functions in the coordinate axes. An example of this case is shown in Figure 2.3a. It shows a simple 2-d hat function. The Tensor Product implies the following operation.

$$\phi_{X-Y}(x,y) = \phi_X(x) * \phi_Y(y) \tag{2.2}$$

Here $\phi_X(x)$ and $\phi_Y(y)$ are the hat function defined on the respective coordinate axes. This is extended to the multi-dimensional case. The nodal basis of functions is the set of functions defined on $[0,1] \rightarrow \mathbb{R}$ represented by the set $\omega = \phi_i | 0 <= i < N|$. N is the number of discrete points (See Fig 2.2c). The support of a nodal basis function is defined as the interval over which the function is non-zero.

Using this nodal basis of functions we can solve the relevant PDE's. This involves finding the integral of a complex function using approximations to the function. Though in this case we use the grid in the context of solving the PDE's this can be done wherever an approximation to a complex function using simple basis functions is required. The approximation for the integral calculation of this complex function is shown below.

$$\int_0^1 f(x)dx = \int_0^1 \sum_{i=0}^{N-1} \alpha_i \phi_i(x)dx = \sum_{i=0}^{N-1} \alpha_i \int_0^1 \phi_i(x)dx \tag{2.3}$$

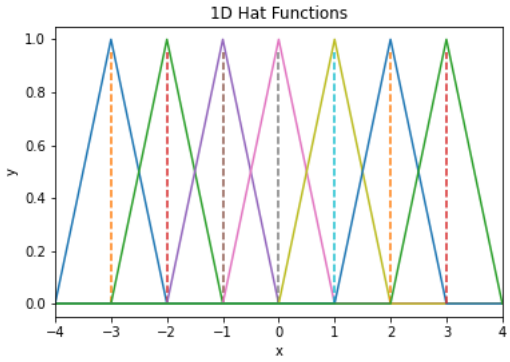
Here $\phi_i(x)$ is the nodal basis and the space of $[0,1]$ is discretized into N points. α_i is the coefficient associated with each nodal basis function. These α_i have to be solved for. Iterative methods are often used to obtain the values of α_i. We already know the definition of the basis functions allowing us to calculate the integral.

Sparse Grids take advantage of the fact that not all basis functions are equally important in the reconstruction of a target function. Sparse Grids use a hierarchical basis of functions. A hierarchical basis is so called because it consists of an hierarchy of discretized grids with each grid lower on the hierarchy being a finer grid than the one above it. In the hierarchical basis of functions not all nodal functions on each level is taken.

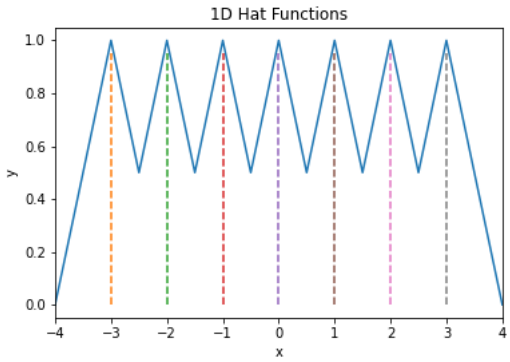
Without loss of generalization we take the function to be one-dimensional in the space of $\Omega = [0,1]$. To approximate a function on this interval we consider a family of grids Ω_l where l denotes the level of the grid. Each grid is discretized with a grid size of $h_l = 2^{-l}$ this leads to $2^l - 1$ points per grid level. These points are represented by $x_{l,i} = ih_l, 1 \leq i \leq 2^{l-1} - 1$. The hierarchical basis functions are constructed on each level of this grid as follows:

$$\phi_{l,i} = \phi(\frac{x - x_{l,i}}{h_l}), \quad 1 \leq i \leq 2^l - 1 \tag{2.4}$$

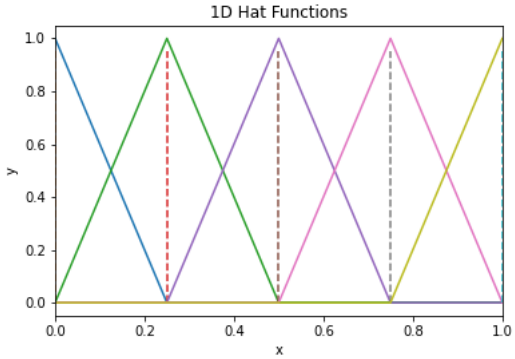
$$\phi_i(x) = \begin{cases} 1 - |x - x_i|, & \text{for } x_{i-1} \leq x \leq x_{i+1} \\ 0 & \text{else} \end{cases} \tag{2.5}$$



(a) A Series of One-Dimensional hat functions

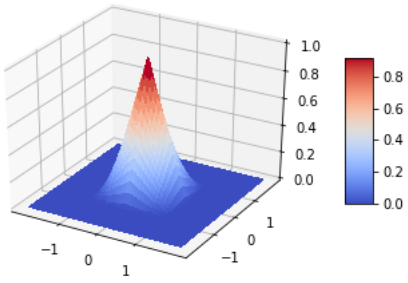


(b) Effective function basis on one-dimensional grid

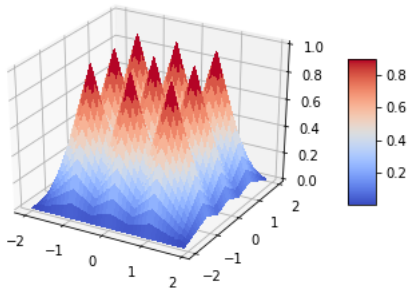


(c) Nodal basis on one-dimensional grid

Figure 2.2.: Hat Functions for 1D Grid



(a) A Simple Two-Dimensional hat function



(b) A Series of Two-Dimensional hat functions

Figure 2.3.: Hat Functions for 2D Grid

The hierarchical basis functions are taken to be linear hat-functions. These can be changed for including more expressivity in the model. The difference between the hierarchical basis and the nodal basis is shown in Fig 2.4. The figure shows the hierarchical basis (above the dotted line) for a level 3 grid. $W1, W2$ and $W3$ are the differently discretized grids. All the Φ's denote basis functions. The basis function for this $\mathbb{H}_3 = \phi_{3,1}, \phi_{2,1}, \phi_{3,3}, \phi_{1,1}, \phi_{3,5}, \phi_{2,3}, \phi_{3,7}$. They exist in different grids unlike the nodal basis that are in the same grid. The nodal basis is shown below in $V3$. The nodal basis is what is used in the case of the full grid. The different grids are called subspaces. They can be defined as W_l given by [2],

$$W_l = span\{\phi_{l,i}, i \in I_l\} \tag{2.6}$$

with,

$$I_l = \{i \in \mathbb{N}^d : 1 \leq i \leq 2^l - 1, i_j odd\ for\ all\ j\} \tag{2.7}$$

A Sparse Grid of level n consists of all subspaces with $\sum_i^D l_i \leq n$ where l_i is the discretization level of the grid in co-ordinate axis i. D is the dimensional size of the subspaces. A l_i value of 2 implies a discretization length of $1/2^2$ for a unit length. Thus a Sparse Grid of dimension 2 will contain the subspaces $(1,1), (1,2), (1,3), (2,1), (2,2), (3,1)$. This is shown in Figure 2.5. The Figure shows all the subspaces for the full grid. The subspaces above the dotted line shows the subspace for the Sparse Grid of level 3.

2.2. Deep Learning

Machine Learning is becoming a critical part of the modern world due to the explosion of data and the race to utilize this valuable resource. Machine learning algorithms had been achieving great success in the problems of regression and classification. Yann LeCun who is a pioneer in the field, first applied the standard backpropagation algorithm with deep neural networks back in 1989 [11]. His algorithm though successful was extremely time-consuming and required a lot of resources. Many other advances were made with the discoveries of CNN for image recognition tasks [13], use of LSTM's [5] and RNN [19] and attempts to deal with graph-based data [9] too. The Deep Learning revolution took off since 2010 due to the availability of vastly more powerful hardware specifically GPU's and significant improvements in Programming paradigms. This was reflected in 2012 when deep neural networks achieved superhuman performance in classification tasks [10]. Their performance has only improved since then. Discoveries in how the human brain functions have also helped in this revolution [20].

Deep Learning is so named because of the "deep nature" i.e the utilization of multiple layers of neural/convolutional layers to increase the expressivity of the model. The neural layer consists of "neurons" that are computational centres that operate on the input data. The convolutional layer operates on the entire input data and the use of the convolution

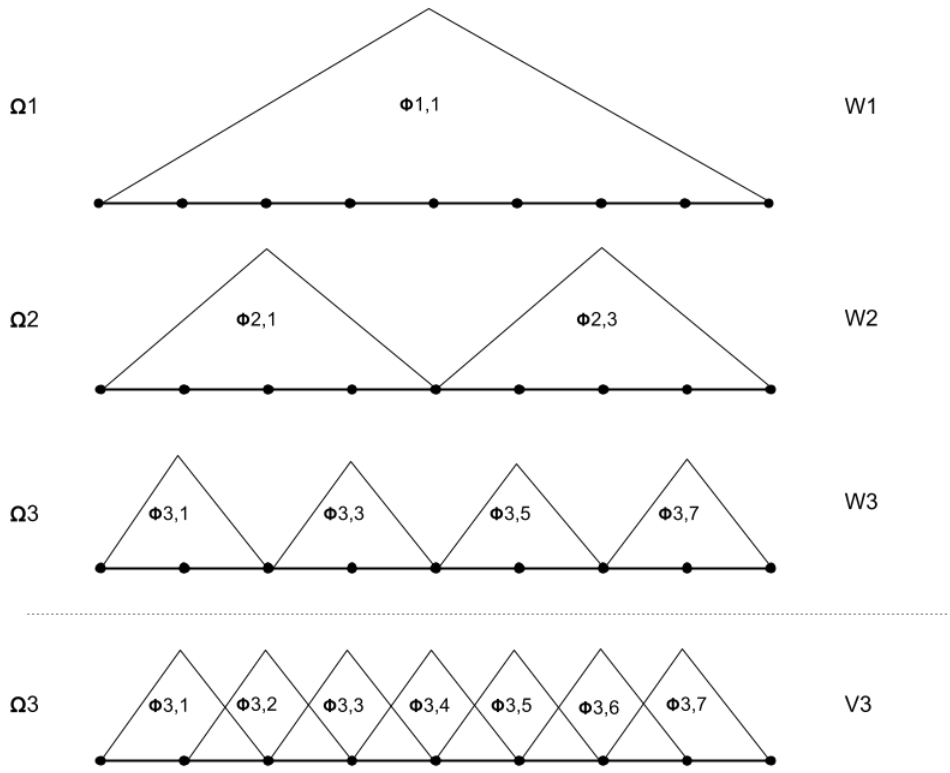


Figure 2.4.: A Hierarchical basis of functions (top) vs. A nodal basis of functions (bottom)

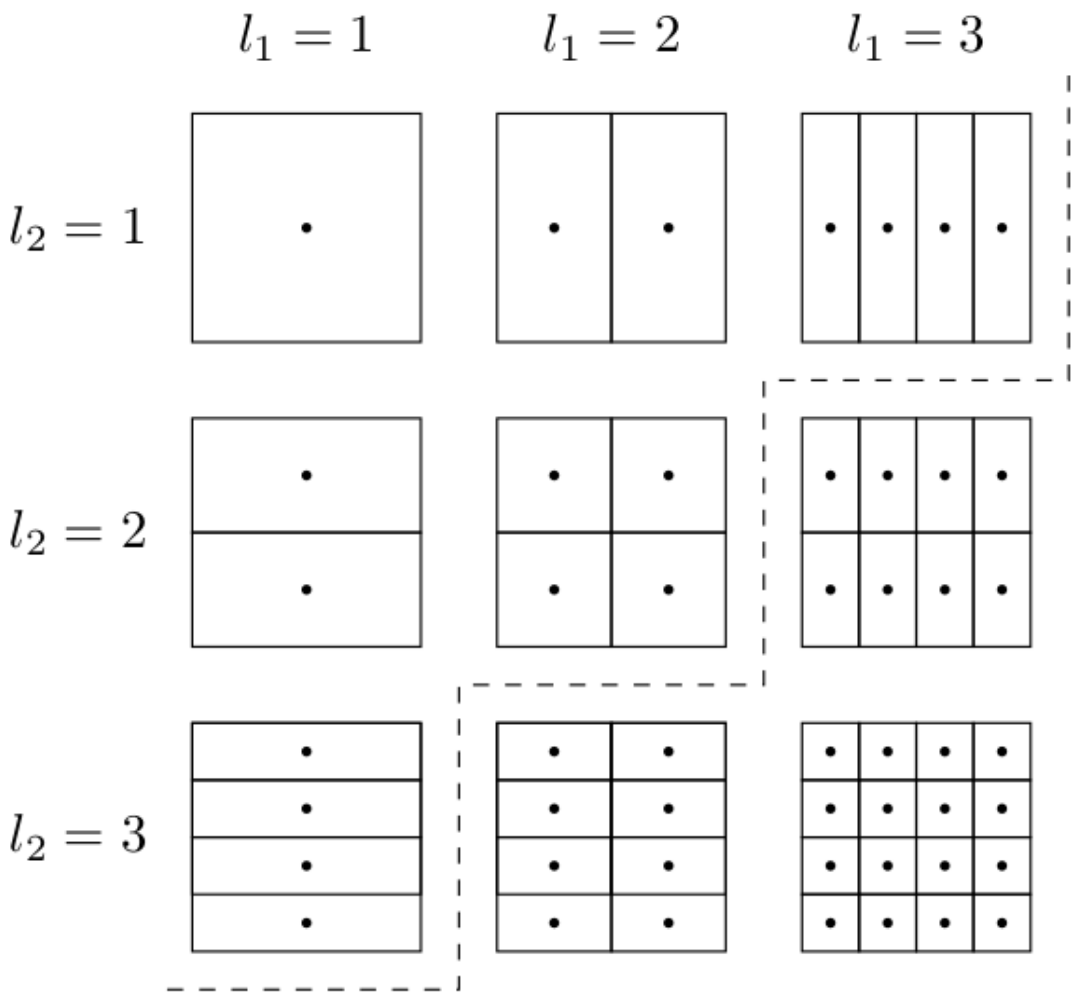


Figure 2.5.: Two Dimensional Subspaces (Source - [15])

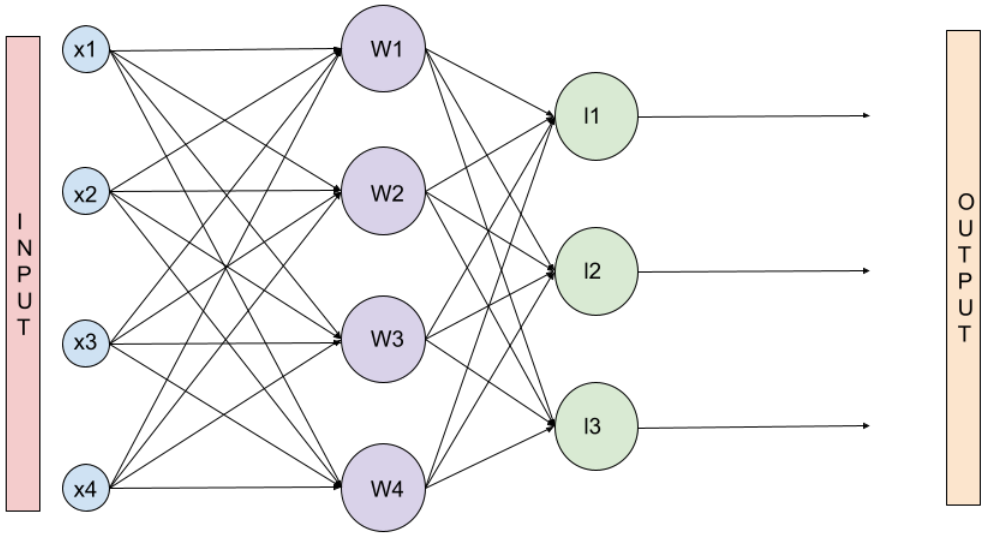


Figure 2.6.: A Simple Deep Neural Network

operation implies strong correlation of data to neighbouring points. A simple deep neural network of depth 1 is shown in Fig 2.6. The Input is x with the components being numbered from one to four. The hidden layer (in purple) of the Networks consists of the weights W1, W2, W3 and W4. This is called the hidden layer as it is "hidden" from us unlike the input and output layers. The Outputs are the labels output l1,l2,l3 which are the probabilities of each label. This is also called as a fully connected layer as each input elements influences the values of all weights. A more realistic example would have many hidden layers with many more individual weights per layer.

The core idea that incorporated the learning aspect into these "deep" networks is Backpropagation. Backpropagation was first discussed in 1960 by Henry J. Kelley [6]. The idea was further improved on with the inclusion of the chain rule for derivatives. The earliest contributors to Backpropagation were Seppo Linnainmaa [14] and Werbos [22] .Backpropagation is the method by which the Network calculates the gradient of the objective function with respect to the weights of the Networks. The objective/loss function is something

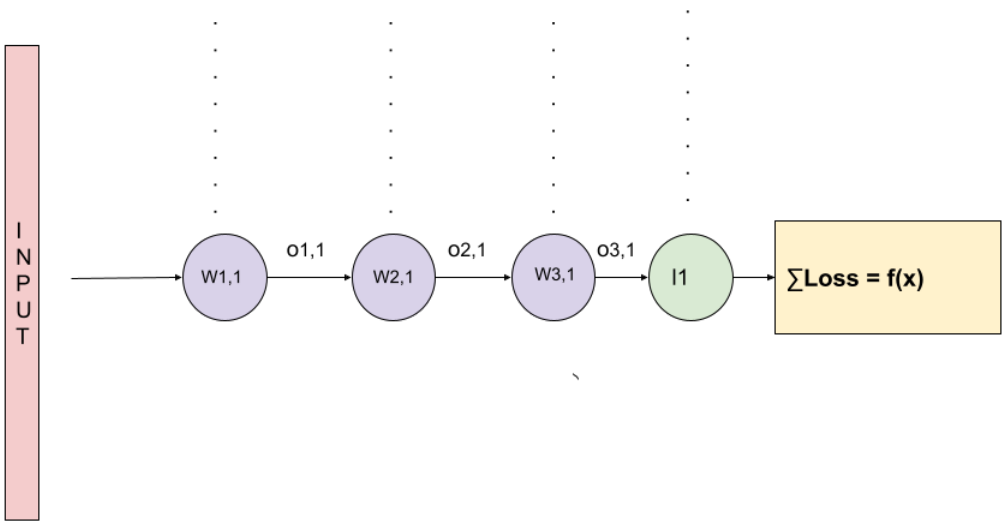


Figure 2.7.: A Backpropagation Example

that the Network is trying to minimize to solve the problem. This can take the form of Cross-Entropy loss for correct label classification, Mean square error loss for direct image generation, L1 loss for simpler data and many other cases. Using 2.7 we explain in simple terms the idea of backpropagation.

In Figure 2.7, a deep neural network is shown which calculates the loss value using a function $f(x)$ where x are the calculated label probabilities l. To optimize this loss function we calculate the gradient of the loss function with respect to the variables i.e the weights of the network. The gradient is being calculated as the optimization techniques involve a gradient based update rule to the variables. Here a very important mathematical idea is used i.e the Chain rule of Derivatives. The chain rule is shown in 2.8.

$$\frac{\delta u}{\delta t} = \frac{\delta u}{\delta x} * \frac{\delta x}{\delta t} \quad where \quad x = f(t) \tag{2.8}$$

We perform the calculations for one such weight in the Network(2.9).

$$\begin{aligned}\frac{\delta Loss}{\delta W_{1,1}} &= \frac{\delta Loss}{\delta l1} * \frac{\delta l1}{\delta W_{1,1}} \\ &= \frac{\delta Loss}{\delta l1} * \frac{\delta l1}{\delta o_{3,1}} * \frac{\delta o_{3,1}}{\delta W_{1,1}} \\ &= \frac{\delta Loss}{\delta l1} * \frac{\delta l1}{\delta o_{3,1}} * \frac{\delta o_{3,1}}{\delta o_{2,1}} * \frac{\delta o_{2,1}}{\delta o_{1,1}} * \frac{\delta o_{1,1}}{\delta W_{1,1}}\end{aligned} \tag{2.9}$$

Similarly the gradients are calculated with all other weights. This is automatically implemented in frameworks such as PyTorch and Tensorflow and the feature is called autograd. Deep Learning Networks have many different update rules. The use case of a certain type of optimizer depends on the situation and the problem being solved but often the best performing one can only be found after training. There are four main variations though many more niche optimizers do exist.

1. Standard Gradient Descent
2. Stochastic gradient descent
3. RMSProp
4. Adam

There exists a lot of literature regarding the advantages and what essentially each type of optimizer is good at capturing. The Adam optimizer [7] is at present the most widely used optimizer.

Depending on the nature of data and what is to be learned deep learning techniques techniques can be divided into three large groups. They are supervised learning, unsupervised learning and Reinforcement learning. Supervised learning is performed with fully labeled data and when the objective is to know the labels of unknown samples. In unsupervised learning the data is unlabelled and the objective is to learn hidden functional relations and patterns. Semi-supervised learning is a mix of both of these approaches. It has partially labeled data and the idea is to generate labels for the unlabeled data after training. Reinforcement learning is concerned with how an agent reacts to surroundings and encouraging/discouraging certain actions based on a rewards system. Unsupervised learning is a very important field among these as most of the data in real-world is often unlabelled and labeling data is a time-consuming and expensive process. It could also be that in certain cases we do not know the exact labels to be used.

This thesis goes in depth into unsupervised learning with Autoencoders and Variational Autoencoders.

Autoencoders learn latent representations of input data. The architecture is shown in Figure 3.2. The model consists of two main blocks - a Encoder and a Decoder. The Encoder

Figure 2.8.: A Interpolation of Latent Variables of an Autoencoder (Source - [1])

and Decoder are fully-connected layers as shown in 2.6. The Weights are of the appropriate dimensions to result in the predefined latent space size. Each value in the latent space representation characterized some feature of the original input data. These can be studied by varying these values i.e performing interpolations to better understand the features being learned (See Fig. 2.8). The Figure shows the effect on the output for interpolating the latent representations of the MNIST dataset.

3. Sparse Grid Autoencoder

This section details the development process of the Sparse Grid Autoencoder and the Sparse Grid Variational Autoencoder.

In Section 3.1 the Methodology behind the Sparse Grid Autoencoder (SGA) and the Sparse Grid Variational Autoencoder (SGVA) is discussed. Section 3.2 explains the Development of the SGA and SGVA models. In Section 3.3 the implementation details of the SGA and SGVA models in Python is described. Finally Section 3.4 details the Hyper-Parameter Tuning of these models.

3.1. Methodology

The main idea in the Thesis is to substitute the conventional Neural Network that is composed of linear layers and non-linear activation functions with Sparse Grids. The input to the Network undergoes multiple layers of functional composition with linear and non-linear functions. The output of such a Network can be succinctly represented by

$$h(x) = ReLU(f(ReLU..x)) \quad \text{where} \quad f(x) = \mathbf{W}x + \vec{b}$$

Here x represents the input data to the Network and ReLU is a non-linear activation function (Fig. 3.1). $\mathbf{W}$ is the Weight matrix and $\vec{b}$ is the bias that is added to the model. $f(x)$ is the mathematical operation that is performed when the input is passed through a layer of the network. The $ReLU$ function is defined as

$$ReLU(x) = max(0, x) \tag{3.1}$$

As explained previously we replace these layers with a single Sparse Grid Layer which can be represented as follows

$$h(x) = f(scaled\ x) * \mathbf{W} \quad \text{where} \quad f(x) = a * g(x - x_{sg})$$

Here $scaled\ x$ refers to the input that has been scaled to the hypercube $[-1, 1]^d$. $\mathbf{W}$ is the Weight matrix and $f(x)$ is the short-form representation of the main calculation of the Sparse grid. $g(x)$ is a clamping function that limits the values to the range $[0, 1]$, x_{sg} is the points of the sparse grid and a is a known co-efficient. This is done for the both the Encoder and Decoder sections of the Sparse Grid Autoencoder. The Sparse Grid Autoencoder utilizes the same general architecture as the Autoencoder architecture.

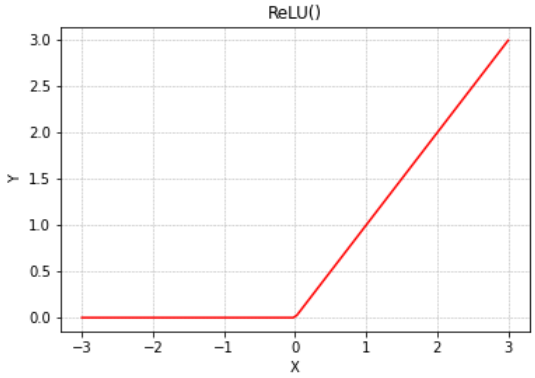


Figure 3.1.: Plot of the $ReLU(x)$ non-linearity function

The Sparse Grid Variational Autoencoder (SGVA) follows a similar architecture to the standard Variational Autoencoder. The limitation of the input to the Sparse Grid Variational Autoencoder in that it should be in the hypercube $[-1, 1]^d$ significantly changes the calculation of the assumed prior distribution.

3.2. Model Development

The Sparse-Grid Autoencoder architecture is very similar to the conventional Autoencoder in that it consists of a Encoder and a Decoder section(See Fig 3.2 & 3.3). The Encoder and Decoder are Sparse-Grid Layers of the appropriate dimensions of the input and the intended latent dimensions. Since these dimensions are pre-determined the architecture is built only for a pair of input dimensions and latent dimensions.

The big difference between the architectures is the inclusion of the addition data scaling layers. These are included as the Input to the Sparse Grid have to be in the range of [-1,1] due to the definition of the hierarchical basis functions for the Sparse Grid Layer. The basis functions are taken to be piecewise linear functions with a support length of twice the discretization length. These are defined for every sparse grid point.

The Sparse Grid Variational Autoencoder has more significant variations to the standard Variational Autoencoder. The limitation that the Sparse Grid Layer can only accept inputs in the range $[-1, 1]$ changes the assumption of the prior distribution from a standard Gaussian distribution. This is due to the fact that samples generated from a Gaussian distribution have a non-zero probability of being outside the specified range which would violate the requirements of the Sparse grid layer. The addition of data-scaling to the generated samples would twist the originally assumed prior thereby defeating the purpose. This work takes the assumed prior to be a Uniform distribution on the interval $[0, 1]$. This solves the aforementioned problem.

The loss function of the Variational Autoencoder is given by Equation 3.2. This is the

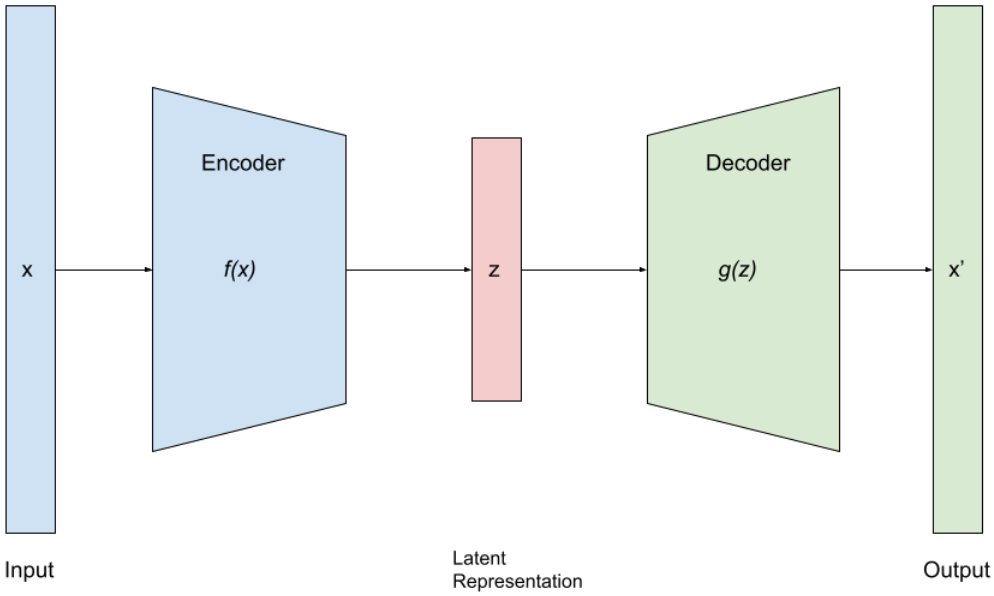


Figure 3.2.: Autoencoder architecture

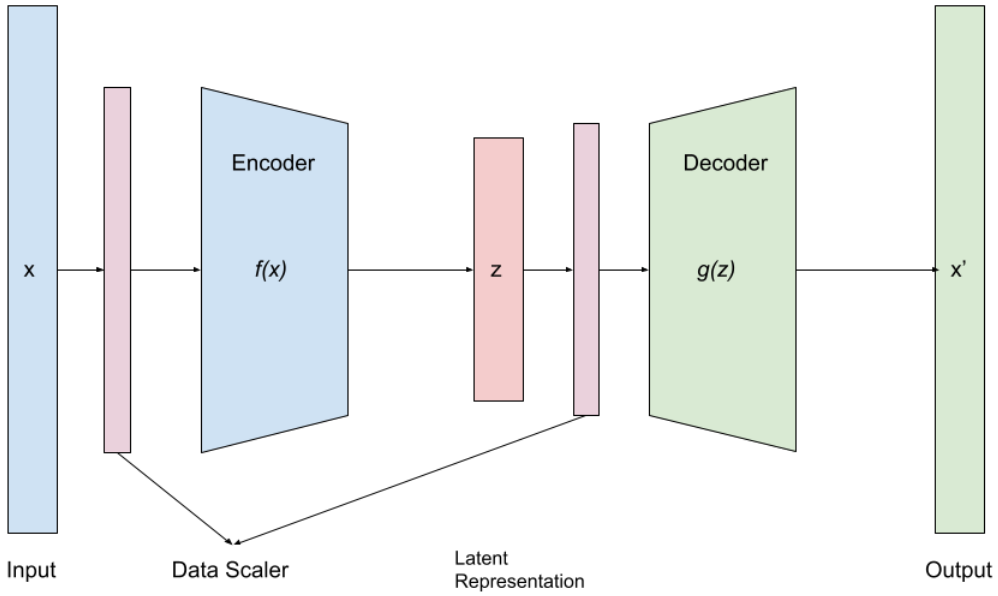


Figure 3.3.: Sparse Grid Autoencoder architecture

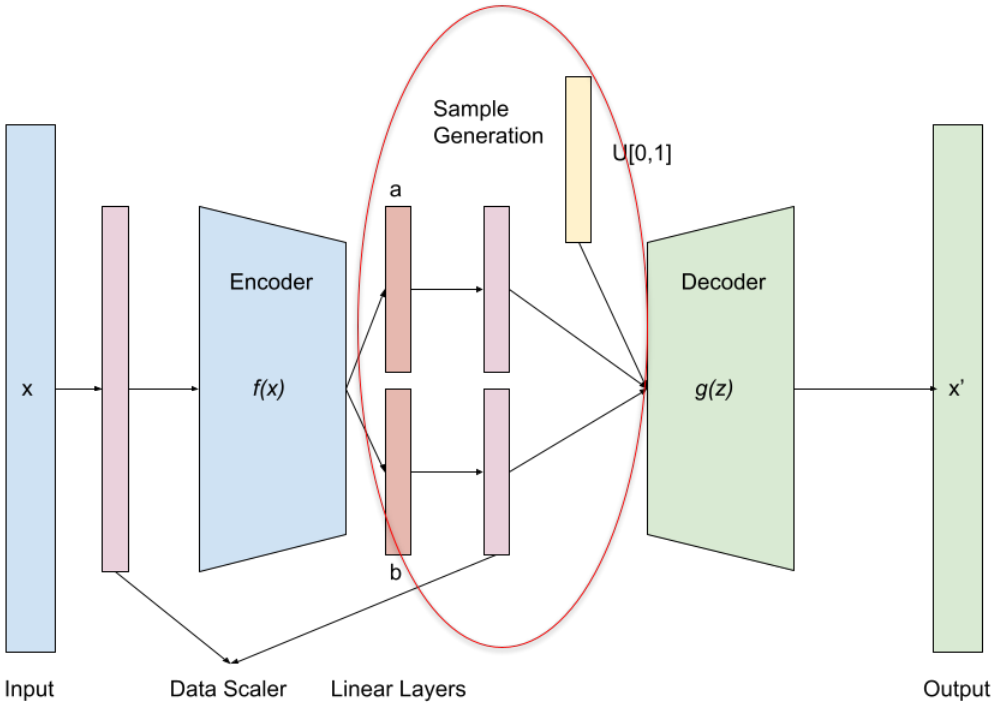


Figure 3.4.: Sparse Grid Variational Autoencoder architecture

generic definition as applied to a individual data point i. The first term is the reconstruction loss i.e it is a measure of how close the final output of the Variational Autoencoder to the Input. The second term is a called the Kullback-Leibler divergence. It is a measure of how close the distribution of the assumed prior $p(z)$ is to the distribution of the encoder output $q_\theta(z|x_i)$. If they are identical this term becomes zero which is the ideal case.

$$l_i(\theta, \phi) = -\mathbb{E}_{z\sim q_\theta(z|x_i)}(\log p_\phi(x_i|z)) + \mathbb{KL}(q_\theta(z|x_i)||p(z)) \tag{3.2}$$

The reconstruction loss in the Sparse Grid Variational Autoencoder is taken to be the Mean Square Error loss between the final output and the input. The Kullback-Leibler divergence has to be calculated for our assumed prior, the Uniform distribution $[0, 1]$. The Kullback-Leibler divergence is defined as in Equation 3.3.

$$\mathbb{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) \log(\frac{p(x)}{q(x)})dx \tag{3.3}$$

Using 3.3 to calculate the loss for the Sparse Grid Variational Autoencoder gives us:

$$\begin{aligned}
\mathbb{KL}(q_\theta(z|x_i)||p(z)) &= \int_{-\infty}^{\infty} q_\theta(z|x_i) \log(\frac{q_\theta(z|x_i)}{p(z)}dz) \\
&= \int_{-\infty}^{\infty} q_\theta(z|x_i) \log(q_\theta(z|x_i))dz - \int_{-\infty}^{\infty} q_\theta(z|x_i) log(p(z))dz \\
&= \int_{0}^{1} q_\theta(z|x_i) \log(q_\theta(z|x_i))dz - \int_{0}^{1} q_\theta(z|x_i) log(p(z))dz \\
&= \int_{a}^{b} \frac{1}{b-a} \log(\frac{1}{b-a})dz - 0 \quad ([a, b] \in [0, 1] \text{and} p(z) = 1 \forall z) \\
&= \frac{1}{b-a} \log(\frac{1}{b-a}) \int_{a}^{b} dz \\
&= \log(\frac{1}{b-a})
\end{aligned}$$

This loss is implemented in the code.

3.3. Implementation

This section will describe the Implementation of the Sparse Grid Autoencoder (SGA) and the Sparse Grid Variational Autoencoder (SGVA).

The Sparse Grid Autoencoder (SGA) and the Sparse Grid Variational Autoencoder (SGVA) are implemented in Python using the PyTorch framework. Python is used as the Programming language in this Thesis due to wide availability of libraries and tools designed for Machine Learning. Since there is no layer implementation of the Sparse Grid in PyTorch as this is a novel idea the layer architecture has to constructed from scratch.

3.3.1. Sparse Grid Layer

The Sparse Grid is implemented in the code in two stages. The first stage deals with the actual construction of the Sparse Grid points i.e their positions and subspace coefficients. The second stage constructs the Sparse Grid class with all of the associated features namely the hierarchical functions, functional coefficients calculation and many necessary parameter retrieval functions for further use down the pipeline.

The first stage is the construction of the Sparse Grid points. This stage requires two main inputs the number of dimensions of the Sparse Grid and the discretization level of the Sparse Grid. These are hyper-parameters that has to be decided based on the data and problem. The idea is explained in Algorithmic terms in Algorithm 1. We first need to find the valid subspaces for the Sparse Grid of specified input parameters. This is calculated in two parts. Firstly, all subspaces i.e level vectors are calculated that are of the dimension of the Sparse Grid. Then the validity of the subspace to belong to the the Sparse Grid of that level is checked. After this process we have the list of all valid subspaces. The second part is calculating the co-ordinates of the sparse grid points, the subspace coefficient and the scaling factors for each subspace. The scaling factor is the factor by which you add to the co-ordinates of a grid point to get another grid point. The importance of this measure is to calculate at each subspace level the support of the sparse grid functions.

The Code snippet 3.1 is how the first stage is worked out. The `combi_scheme` variable is responsible for generating the subspace levels associated with the Sparse Grid of the specified dimension and discretization level. Each of these subspaces have the associated grid points and they all have a common coefficient associated with their subspace. This is calculated in the variable `coefficients`. The variable `positions` determines the Cartesian co-ordinate in the $[0, 1]$ hypercube for the Sparse Grid points. This is calculated with the help of `combi_scheme` as once the subspace is known, the discretization level in each dimension is also known. `scales` calculates the support values of the functions for each subspace level.

The second stage is assembling the Sparse Grid and all of the associated functions together. This is done inside the class `CombinedSparseGridTest` in the code (See 3.2). The `_construct_grid` functions as the name implies constructs the grid using the given positions and scale_factors. These are set to not requiring the gradient as these are not the Parameters to be learned. Setting the gradient is not required saves a lot of valuable computation time as otherwise the program would be trying to calculate the backward pass of the network with respect to these Parameters. In PyTorch "Parameter" is a special Tensor that becomes associated with the parameter iterator. This allows to check for all of the inputs/variables required for a particular class. A Tensor is the basic data structure in PyTorch that comes with the benefit of being able to run on either the CPU or the GPU when compared to the standard numpy implementation for a array or matrix. The `compute_coefficients` function does the computation of the exact coefficients if the output of the Sparse layer is known. This is useful if the latent representation is already known. The problem statement of the Thesis does not specify the knowledge of latent

```
    combi_scheme = [level_vector for level_vector in product
(*[list(range(1,level+1)) for d in range(n_latent_dimensions)])
if level <= sum(level_vector) <= level + n_latent_dimensions - 1]
    combi_coefficients = []
    dim = n_latent_dimensions
    for level_vector in combi_scheme:
        q = (level + n_latent_dimensions - 1 - sum(level_vector))
        coefficient = (-1)**q * math.factorial(dim-1)/
(math.factorial(q)*math.factorial(dim-1-q))
        combi_coefficients.append(int(coefficient))

    positions = np.array([]).reshape(0,dim)
    scales = np.array([]).reshape(0,dim)
    coefficients = np.array([], dtype=int)
    for grid, coefficient in zip(combi_scheme, combi_coefficients):
        level_vec_combi_grid = 2**(np.asarray(grid)) - 1
        num_points_grid = np.prod(level_vec_combi_grid)
        positions = np.concatenate((positions,np.asarray
(list(product(*[np.linspace(0,1,level_vec_combi_grid[d] + 1,
endpoint=None)[1:] for d in range(dim) ])))))
        scales = np.concatenate((scales,np.ones((num_points_grid,dim)) *
1/(level_vec_combi_grid+1)))
        coefficients = np.concatenate((coefficients,
np.ones(num_points_grid) * coefficient))
```

Code 3.1.: Sparse Grid Points Setup

Algorithm 1 Sparse Grid Points Setup

for $iteration = 1, 2, \ldots level$ **do**
    Generate all possible combinations of grids i.e level vectors ($iteration, \ldots$) of latent dimension size.
    Check for validity of Subspace to be part of Sparse Grid.
    if $level \leq \sum_i levelvector_i \leq level + latent\_dimension\_size - 1$ **then** Accept level vector
    else Reject level vector
    end if
end for
for $level\_vector \in Total\_set\_of\_level\_vectors$ **do**
    Calculate the Coefficient of Subspace.
    Calculate the Positions of Sparse Grid points in Subspace.
    Calculate the scaling factor between points for Subspace.
end for

space representation. Thus, this is going to be a parameter to be learned by the network. `size_return` and `parameters_return` are functions defined to retrieve specific values required for future computations.

The Sparse Grid setup for the Encoder and the Decoder are identical despite the difference in dimensionality.

3.3.2. Sparse Grid Autoencoder

Once the Sparse Grid is set up for the Encoder and the Decoder, they are combined together in the Autoencoder module. As mentioned previously we need to scale the data that is fed into the Sparse Grid Layer. This is achieved with a `data_scaler` function that scales the data to $[-1, 1]$ (See Code 3.3). The section of code takes into account the maximum and minimum values of input in each dimension and then scales as per the following formula.

$$scaling\_function(x) = \left(2 * \frac{x - x_{min}}{x_{max} - x_{min}}\right) - 1 \tag{3.4}$$

In the intermediate stage when going from the latent space representation to the Decoder it was observed that the Tanh non-linearity function gave superior results to this scaling formula. The Tanh non-linearity is very similar to the definition defined above as in it too maps the input data to the range $[-1, 1]$. Tanh is defined as

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.5}$$

The Tanh non-linearity is graphed in Fig 3.5.

```
class CombinedSparseGridTest():
"""
Generic Class for a Sparse Grid
"""
def __init__(self, positions, scales, combi_coefficients,
l2_regularization=1e-5):
    self.size = (positions.shape[0])
    self.l2_regularization = l2_regularization
    self._construct_grid(positions, scales)
    self._combi_coefficients = combi_coefficients

def _construct_grid(self, positions, scales):
    """
    Constructing Sparse Grid with given position of points and
    respective scaling level
    """
    self.positions = Parameter(torch.Tensor(positions),
requires_grad = False)
    self.scales = Parameter(torch.Tensor(scales), requires_grad = False)

def compute_coefficients(self, _inputs, _outputs):
    """
    Exact Calculation of Coefficients of Sparse Grid (After Training)
    """
    _kernel = self.kernel(_inputs)

    self._sg_coeff = torch.lstsq( _outputs,_kernel)

def size_return(self):
    """
    Compute the grid with given coefficients
    """
    return self.size

def parameters_return(self):
    """
    Returns the Parameters of the Sparse Grid for further calculation
    """
    return self.positions,self.scales,self._combi_coef
```

Code 3.2.: Sparse Grid Setup

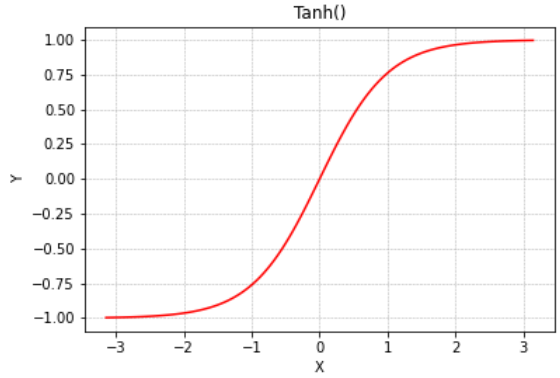


Figure 3.5.: Plot of the $tanh(x)$ non-linearity function

```
def compute_data_scaler(self, data_in):
    d_min = data_in.min(0, keepdims=True)[0]
    d_max = data_in.max(0, keepdims=True)[0]
    scaler = d_max - d_min
    scaled_data = ((2 * (data_in - d_min))/scaler) - 1
    return scaled_data
```

Code 3.3.: Data Scaler

The actual computation with the Sparse Grid is performed in the `hat_function`. This is done in the Code Section 3.4. In the `hat_function`, x is the input data to the Sparse grid, loc is the position information of the Sparse grid points and $scale$ refers to the support of the corresponding hat function defined by the sparse grid point. $combi\_coefficients$ is the coefficient associated with the subspaces of the sparse grid. The pairwise distance i.e distances between the coordinates in each dimension is calculated between each input data point and all the points of the Sparse grid points. This distance is then divided by the support/extant of the hat function associated with the set sparse grid point. This value that is obtained is clamped to $0, 1$ if the value is outside the range $[0, 1]$, otherwise it is set as is. This reflects the actual hat function being performed on the value. The functional value is then multiplied with the associated coefficient of that level and the result is returned to the calling function.

All the experiments were conducted on a sparse grid of discretization level of 4. The Sparse grid looks as shown in Figure 3.6.

The forward pass of the Autoencoder module is shown in Code 3.5. The forward pass is the path that the input takes along the network architecture to obtain the output that is constrained to match some predefined criteria. This is the same as the Architecture

```
def hat_function(x, loc=0, scale=1, combi_coefficients=1):
    # squared pairwise distances
    pairwise_diff = torch.abs(torch.unsqueeze(x, 0) -
torch.unsqueeze(loc, 1))
    pairwise_diff /= torch.unsqueeze(scale, 1)
    hat_evaluation = torch.clamp(torch.abs(1-pairwise_diff), 0, 1)
    output = torch.transpose(torch.prod(hat_evaluation,
axis =-1), 0, 1) * torch.Tensor(combi_coefficients)
    return output
```

Code 3.4.: Sparse Grid Linear Calculation Layer

```
def forward(self,inputs):
    inputs_scaled = self.compute_data_scaler(inputs)
    print(inputs_scaled)
    encoder_output = self.kernel(self.sparse_grids[0], inputs)
@ self._sg_coeff_encoder
    scaled_latent_space = self.tanh_layer(encoder_output)
    decoder_output = self.kernel(self.sparse_grids[1],
scaled_latent_space) @ self._sg_coeff_decoder

    return decoder_output
```

Code 3.5.: Forward Pass of the Sparse Grid Autoencoder

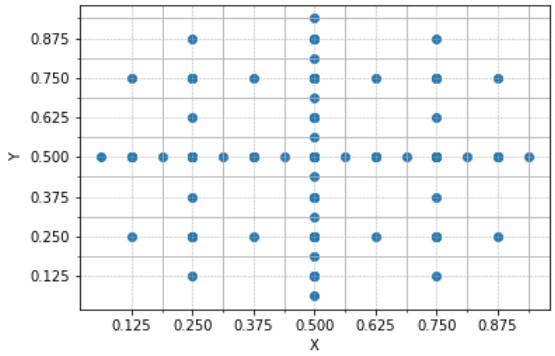


Figure 3.6.: Plot of the Sparse Grid of level 4

```
self._sg_coeff_encoder = Parameter(torch.randn(self.encoder_size,
    self.latent_size, dtype=torch.double), requires_grad = True)
self._sg_coeff_decoder = Parameter(torch.randn(self.decoder_size,
    self.input_size, dtype=torch.double), requires_grad = True)
```

Code 3.6.: Parameter Definition for the Sparse Grid Autoencoder

Diagrams presented in Section 3.2. The input is scaled then passes through the first Sparse Grid Layer which gives the latent representations. This is once again scaled and fed into the Decoder section which finally gives the output that is trained to be as close as possible to the input.

The parameters to be learned are the individual coefficients associated with the Sparse Grid points/functions that are multiplied with the outputs of the Sparse grid functions applied to the inputs. The encoder and decoder coefficients, both have to be learned while training.

The training parameters for the Sparse Grid Autoencoder is shown in Table 3.1. The Loss function used is defined in 3.7.

Learning Rate	0.005
Epochs	100
Batch Size	16
Input Dimensionality	8
Latent Dimensionality	2
Sparse Grid Level	4

Table 3.1.: Training Parameters of the Sparse Grid Autoencoder

```
1  def forward(self,inputs):
2      """
3      Forward Pass of the VAE Module (For Training)
4      """
5      # Scaling Input Data to [0,1]
6      inputs_scaled = self.compute_data_scaler(inputs)
7      # Passing through Encoder Sparse Grid
8      encoder_output = self.kernel(self.sparse_grids[0], inputs_scaled)
9  @ self._sg_coeff_encoder
10     # Latent Space Sample Generation
11     generated_samples, dist_parameters1, dist_parameters2  =
12 self.sample_generation(encoder_output)
13     # Rescaling Latent Space Generated Example to [0,1]
14     scaled_latent_space = self.sigmoid(generated_samples)
15     # Passing through Decoder Sparse Grid
16     decoder_output = self.kernel(self.sparse_grids[1],
17 scaled_latent_space) @ self._sg_coeff_decoder
18     # Scaling Decoder Output to [0,1]
19     decoder_output = self.sigmoid(decoder_output)
20
21     return decoder_output, dist_parameters1, dist_parameters2
```

Code 3.7.: Forward Pass of the Sparse Grid Variational Autoencoder

3.3.3. Sparse Grid Variational Autoencoder

The Sparse Grid Variation Autoencoder has a similar architecture to the Sparse Grid Autoencoder. The main difference with the as explained in Section 3.2 is that the instead of taking the latent representations to be directly the lower-dimensional representation of the input they are instead taken as realizations from a latent distribution whose parameters have to be learned. This leads to a change in the forward pass of the Variation Autoencoder as shown in Code 3.7.

The main difference can be observed in line 11 which generates samples from the prior distribution using the output of the encoder section of the SGVA.

The Code Section 3.8 details the implementation of sampling procedure for the Sparse Grid Variational Autoencoder. As explained previously in Section 3.2 the Variational Autoencoder assumes a certain prior that is approximated using a learned distribution $q(z)$. In the stand Variational Autoencoder the prior is assumed to be Gaussian but this cannot be taken as the case in this architecture as the Sparse Grid cannot input data that exceed the $[-1, 1]$ hypercube. The Gaussian curve extends from $[-\infty, \infty]$ thus samples drawn

```
def sample_generation(self, encoder_output):
    """
    Computes the Parameters of the Assumed Prior Distribution
    (Uniform - [a,b]) and generates samples.
    """
    # a and b parameter computation
    dist_parameters_a = self.fc1(encoder_output)
    dist_parameters_a = self.sigmoid(dist_parameters_a)
    dist_parameters_b = self.fc2(encoder_output)
    dist_parameters_b = self.sigmoid(dist_parameters_b)

    # Generating Samples from Uniform Distribution using
learned parameters
    z = self.reparameterize(dist_parameters_a, dist_parameters_b)

    return z, dist_parameters_a, dist_parameters_b
```

Code 3.8.: Sample Generation of the Sparse Grid Variation Autoencoder

from this distribution will have to be scaled to $[-1, 1]$ for the decoder sparse grid. This restriction will obscure many of the drawn samples and will result in effectively creating an entirely different distribution. This resulting distribution is going to be something extremely complex which would impact the calculation of the loss function. An example of this situation is shown in 3.7.

For the purpose of sample generation, the output generated by the Encoder section is passed through two different linear layers to generate the parameters of the assumed prior distribution i.e the Uniform Distribution. The Uniform distribution is taken to be defined on the interval $[0, 1]$. This is achieved with the help of the non-linearity Sigmoid function that maps the output of the linear layers to the required interval. The Sigmoid function is shown in Fig 3.8.

Code Section 3.9 is the implementation of the reparametrization trick in the Sparse Grid Variational Autoencoder. The learned parameters $\vec{a}$ and $\vec{b}$ of the Uniform distribution $[\vec{a}, \vec{b}]$ is the distribution from which the data is sampled. The parameters/weights associated with the linear functions are also learnable parameters. They are learned in conjunction with the sparse grid coefficients of the encoder and the decoder. Using the distribution parameters are then used in the reparametrization trick used in the Variational Autoencoder. Firstly samples are generated from the uniform distribution $[0, 1]$. These generated samples undergo a transformation with the learned parameters $\vec{a}$ and $\vec{b}$ to the range $[\vec{a}, \vec{b}]$

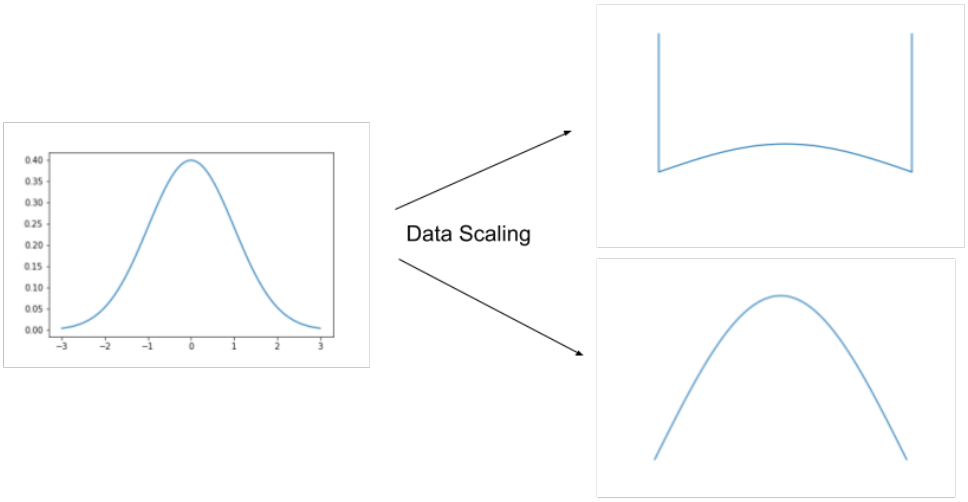


Figure 3.7.: Possible resulting distributions for a Gaussian prior

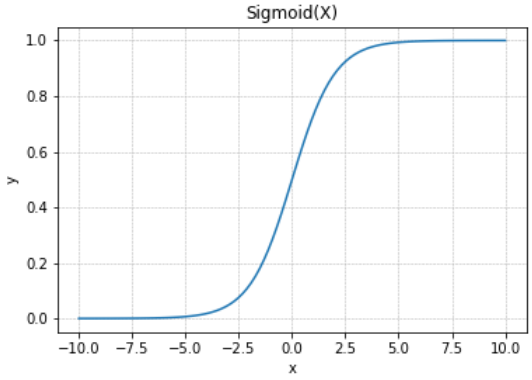


Figure 3.8.: Plot of the Sigmoid function

```
def reparameterize(self, dist_parameters_a, dist_parameters_b):
    """
    Generates Samples using the Computed Parameters
    (Uniform Distribution)
    """
    # Generate Random Samples from [0,1]
    esp1 = torch.Tensor(np.random.random
((dist_parameters_a.shape[0], 1)) )
    esp2 = torch.Tensor(np.random.random
((dist_parameters_a.shape[0], 1)) )

    # Reparametrize of the 2-d latent space
    z1 = dist_parameters_a[:,:1] + (dist_parameters_b[:,:1]
- dist_parameters_a[:,:1]) * esp1
    z2 = dist_parameters_a[:,1:] + (dist_parameters_b[:,1:]
- dist_parameters_a[:,1:]) * esp2

    # Combining the Generated Dimensional Samples
    z = torch.cat([z1,z2], dim = 1)

    return z
```

Code 3.9.: Reparametrization in the Sparse Grid Variation Autoencoder

Learning Rate	0.005
Epochs	200
Batch Size	16
Input Dimensionality	8
Latent Dimensionality	2
Sparse Grid Level	4

Table 3.2.: Training Parameters of the Sparse Grid Variational Autoencoder

```
def loss_fn(recon_x, x, dist_parameters_a, dist_parameters_b):
BCE = F.mse_loss(recon_x, x)
tmp1 = dist_parameters_b[:,:1] - dist_parameters_a[:,:1]
tmp2 = dist_parameters_b[:,1:] - dist_parameters_a[:,1:]
tmp1 = torch.reciprocal(tmp1)
tmp2 = torch.reciprocal(tmp2)
KLD =  torch.sum(torch.log(torch.abs(tmp1 * tmp2)))
return torch.sum(BCE + KLD)
```

Code 3.10.: Loss Function of the Sparse Grid Variation Autoencoder

using the fairly standard formula.

$$U(\vec{a}, \vec{b}) = \vec{a} + U(0,1) * (\vec{b} - \vec{a}) \tag{3.6}$$

The reason for two samples being drawn is due to the fact that the latent dimensionality of the data has been taken as two. This is also the reason for the vector notation for the parameters. So each sample drawn is for each dimension of the latent space. These individual samples are then concatenated to finally be the generated 2-D samples drawn from the Uniform distribution $[\vec{a}, \vec{b}]$.

The Parameters of Training the Sparse Grid Variational Autoencoder is shown in Table 3.2. The Optimizer used in Training is the Adam Optimizer.

The loss function is implemented as described in Section 3.2. First part of the loss function involves the calculation of the Mean Squared Error between the original data and the reconstructed data. This loss is calculated as per the formula 3.7. n is the number of samples and Y and Y' are the original and reconstructed data points respectively.

$$MSE\ Loss = \frac{1}{n}\sum_{i=1}^{n}(Y_i - Y_i')^2 \tag{3.7}$$

The second part of the loss function involves the calculation of the Kulback-Leibler divergence between the returned distribution and a standard Uniform distribution. This acts

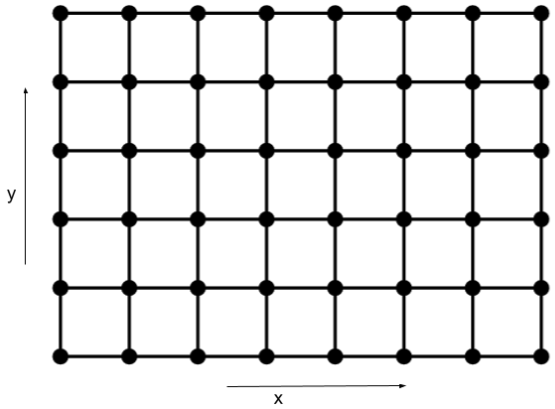


Figure 3.9.: A two-dimensional grid

as a form of regularization that penalized learned distributions depending on how much they deviate from the standard Uniform distribution.

3.4. Results

This section presents the results that have been achieved with the novel Sparse Grid Autoencoder(SGA) and Sparse Grid Variational Autoencoder(SGVA) architectures.

Model Problem 1

To test out the efficacy of the architectures we first simulate a controlled problem which is deterministic. This will help gauge the correctness of the proposed models. The problem is defined as follows for the Sparse Gird Autoencoder.

In this model problem the latent dimensionality of the data is taken to be two-dimensional. This is achieved by taking a two-dimensional grid(See Fig 3.9) and all the points on the grid are the latent dimensional representation of the input data. The low dimensional data is artificially transformed to a high-dimensional data using helper functions. These are transformations on the original co-ordinate data preferably non-linear to increase dimensionality. This problem uses the trigonometric family of functions of $sin(x), sin(2x)$ and so on as well as $cos(y), cos(2y)$ and so on. The sine functions are applied on the x-coordinate of the grid points while the cosine functions are applied on the y-coordinate of the grid. This results in a generated high-dimensional input. The experiment uses the input dimensionality of 8 for the data. Thus the functions used are $sin(x), sin(2x), sin(3x)$ and $sin(4x)$ and similarly for the cosine functions. The grid is taken to be 25x25. Since we have taken the latent dimensional points beforehand we can visualize how accurately the model recreates this grid after training.

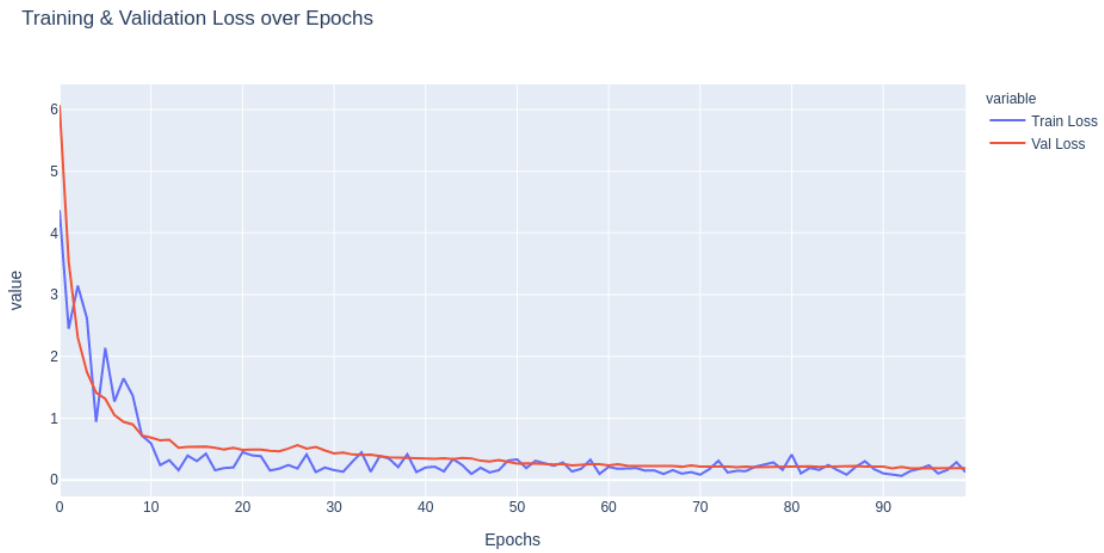


Figure 3.10.: Training & Validation Loss of the Sparse Grid Autoencoder

The training loss of the Sparse Grid Autoencoder is shown in Figure 3.10. The training loss decreases as the training progresses through the epochs and after 60 epochs the decrease in the loss value becomes very small .The validation loss follows the training loss pattern. The reason for the similarity is due to the homogeneity of the initial data. This implies that as the model learns the training data it also learns the validation dataset.

Figure 3.11 shows the latent space representation of the inputs after training the model. The Original 25x25 grid is shown as green crosses and the learned latent space representation are the blue circles. The high-dimensional input was obtained form this original 25x25 grid. The model learns the latent representation of the input but these are far from the actual latent representation. The learned latent representation are unable to capture the full space of the original grid and are very confined along the $x = 0$ and $y = 1$ axes.

To gauge the correctness of the Sparse Grid Variational Autoencoder we construct a model problem that is similar to the one constructed for the Sparse Grid Autoencoder. The Variational Autoencoder model assumes a prior of a certain distribution. This is the Gaussian distribution in the standard case. For the reason explained in Section 3.2 the prior is take to be a Uniform Distribution $[0, 1]$. The example assumes a two-dimensional latent space. This implies a combination of the Uniform Distribution in each axes as the generalized distribution. This is achieved with the help of Numpy helper function `np.random.random` that samples points in from $U[0, 1]$. So the data is generated from this distribution and the model tries to learn the parameters of the Uniform distribution. The type of distribution has to incorporated into the model as otherwise the model(VAE)

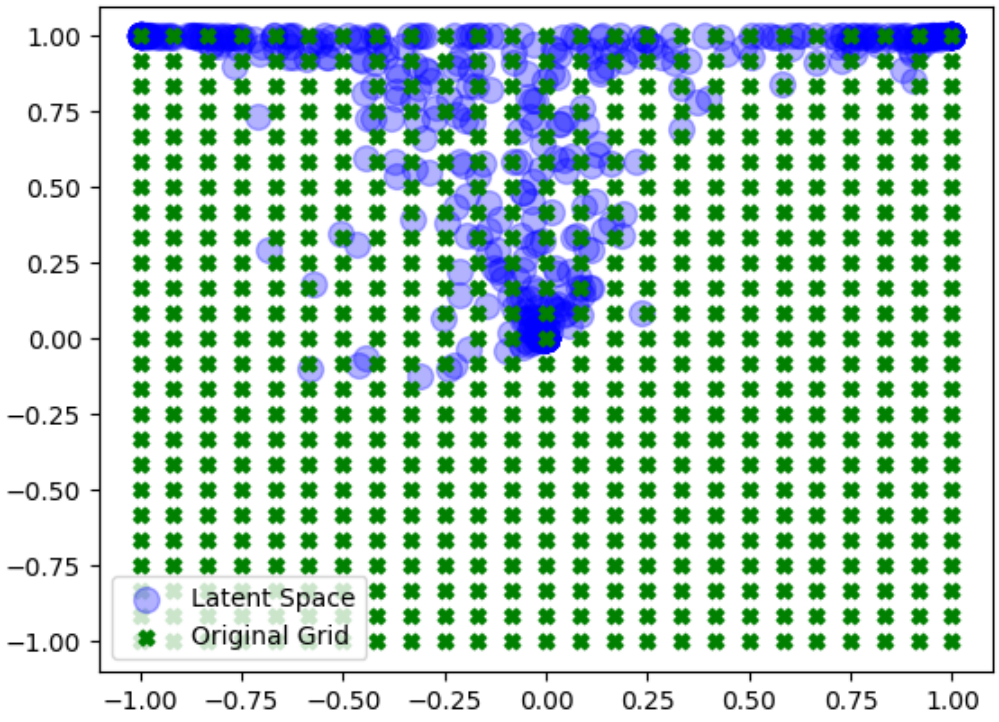


Figure 3.11.: Latent Space Representation of the Sparse Grid Autoencoder

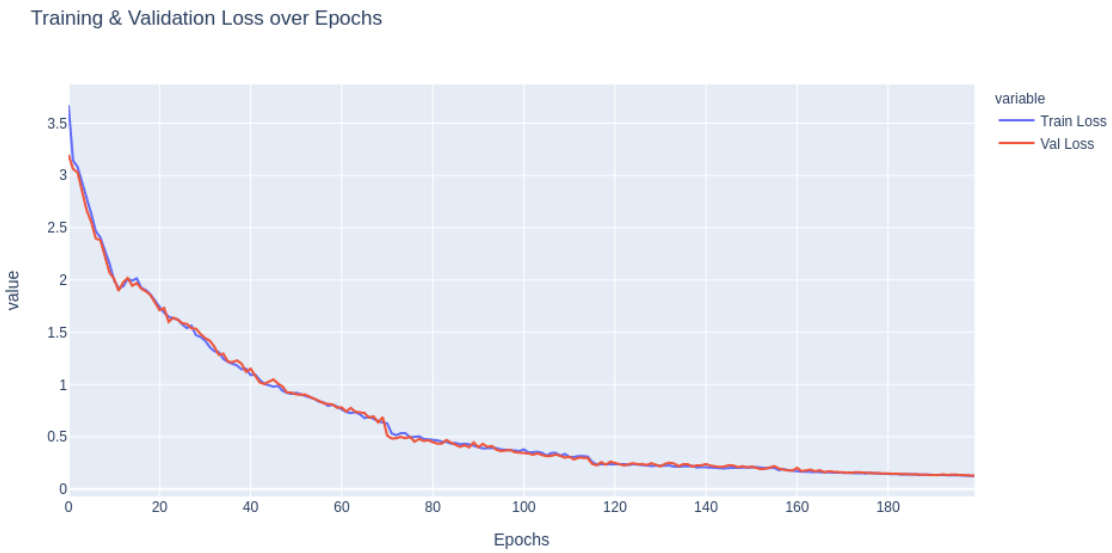


Figure 3.12.: Training & Validation Loss of the Sparse Grid Variational Autoencoder

cannot predict the distribution type. Once the points are sampled we follow the same procedure as in the case of the Sparse Grid Autoencoder to increase dimensionality. The trigonometric functions are used to generate the input data to be fed into the SGVA.

The training loss of the Sparse Grid Variational Autoencoder is shown in Figure 3.12. The training loss decreases as the training progresses through the epochs and after 180 epochs the decrease in the loss value becomes very small .The validation loss follows the training loss pattern. The reason is the same as the SGA the distributions of the training and validation data is identical.

Figure 3.13 shows the latent space representation of the inputs after training the model. The originally sampled points from the Uniform Distribution is shown in green. The latent space samples that are generated from the learned distribution are shown in blue. It is observed that in the case of the Sparse Grid Variational Autoencoder the model is able to learn the full space of the original latent distribution. Hence this model works well.

Hyper-Parameter Tuning

Hyper-Parameters are the parameters that are used in training the data but are not learned from training. They are fixed for each training cycle. Hyper-Parameter Tuning refers to the process by which the hyper-parameters are optimized for obtaining the best-possible result which in most cases implies lowest loss value. The Hyper-Parameters in the Sparse Grid Autoencoder and the Sparse Grid Variational Autoencoder are:

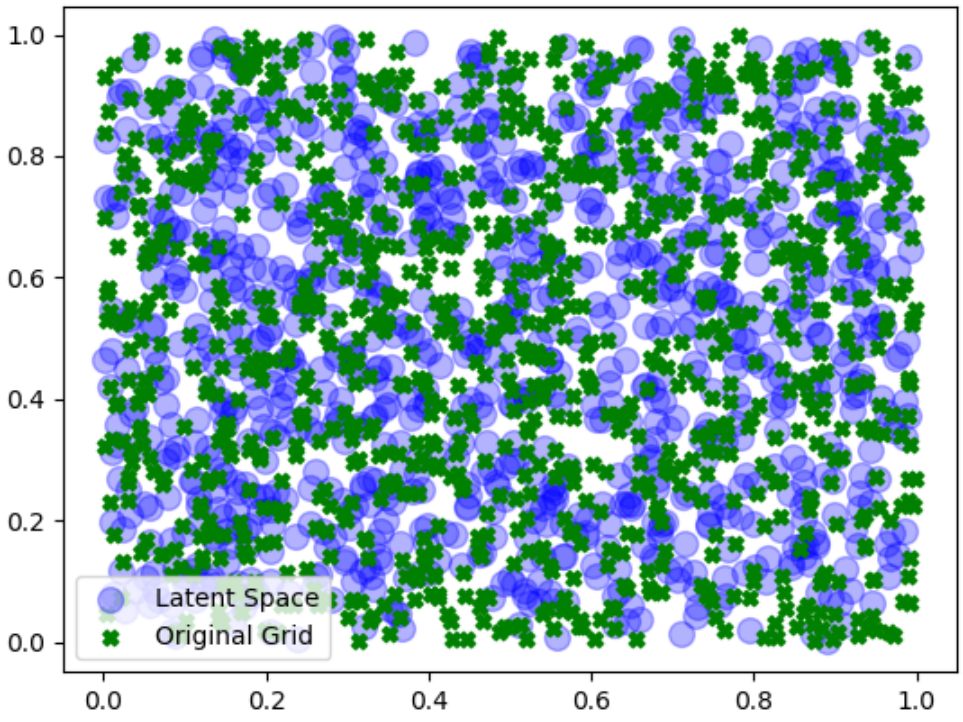


Figure 3.13.: Latent Space Representation of the Sparse Grid Variational Autoencoder

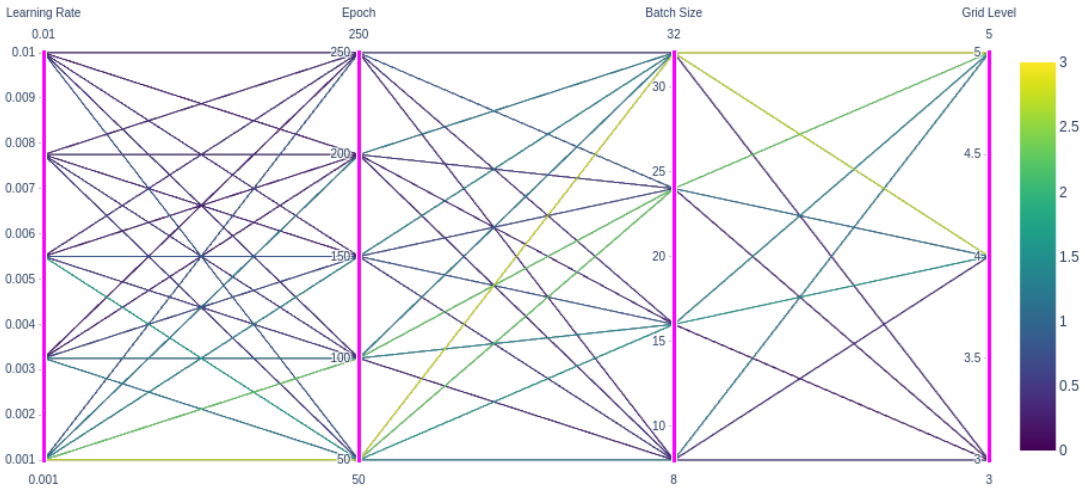


Figure 3.14.: Hyper-parameter Search of the Sparse Grid Autoencoder

1. Level of the Sparse Grid
2. Epochs
3. Batch Size
4. Learning Rate

These are optimized based on a grid search algorithm. The lower and upper bounds of each of these parameters is specified and the training is done for each with the values of parameters for reach point on this grid. The range of values of parameters over which the grid search algorithm is performed in shown in Table 3.3.

Hyper-Parameter	Range	Data Points
Learning Rate	[0.001,0.01]	5
Epoch	[50,250]	5
Batch Size	[8,32]	4
Grid Level	[3,5]	3

Table 3.3.: Hyper-Parameter Table of Values for Sparse Grid Autoencoder

The results of the Hyper-parameter grid search is sown in Figure 3.14. The Colour bar represents the training loss $[0, 3]$. The lines in the plot are coloured based on the associated

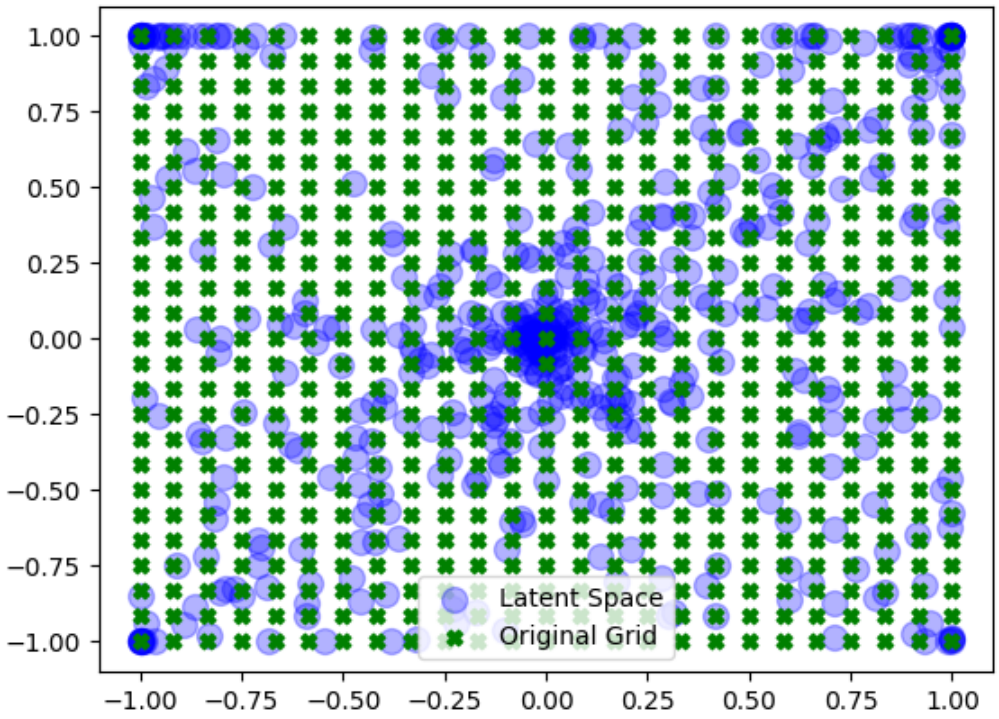


Figure 3.15.: Latent Representation of SGA with best set of hyper-parameters

training loss with that set of hyper-parameters. The darker the colour of the line the better is that set of parameters. With this heuristic, the plot informs us that the worst results are obtained with grid level of 5, batch size of 32, 50 epochs and a 0.001 learning rate. The best training set cannot be taken based on this data alone as a lot of parameter sets have very close training loss values. To further narrow down on optimal sets of hyper-parameters, we look into the learned latent representations. The combination in which the model best captures the original latent space is considered to be good. The "best" in this case means a good distribution of data points throughout the latent space. This criteria gives the best parameter set as $(learning\_rate, epoch, batch\_size, level) \rightarrow (0.001, 40, 32, 4)$. The Figure 3.15 shows the latent representation learned by the model under these hyper-parameters. Even though clumps of learned representations remain, this is a significant improvement over the standard set of hyper-parameters.

The Sparse Grid Variational Autoencoder also undergoes a similar treatment searching for the best set of hyper-parameters. The range of values of parameters over which the

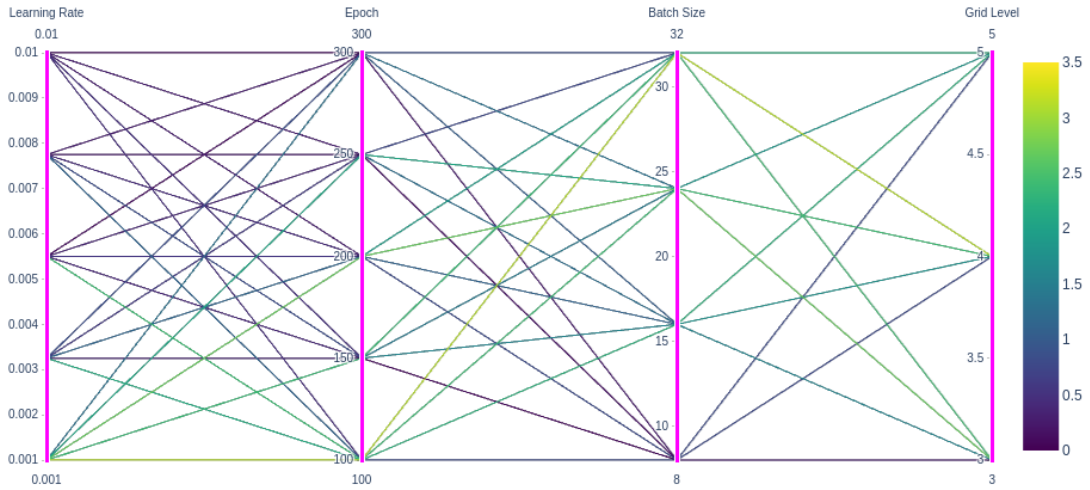


Figure 3.16.: Hyper-parameter Search of the Sparse Grid Variational Autoencoder

grid search algorithm is performed in shown in Table 3.4.

Hyper-Parameter	Range	Data Points
Learning Rate	[0.001,0.01]	5
Epoch	[100,300]	5
Batch Size	[8,32]	4
Grid Level	[3,5]	3

Table 3.4.: Hyper-Parameter Table of Values for Sparse Grid Variational Autoencoder

The results of the Hyper-parameter grid search is shown in Figure 3.16. The Colour bar represents the training loss $[0, 3.5]$. The Parallel graph shows that many combinations of hyper-parameters work well. This is the same case as in the Sparse Grid Autoencoder. Unlike the situation with the SGA the SGVA manages to capture the latent space of the input data very well with a wide range of hyper-parameters. This means that selecting the best combination is impossible and instead a wide set of hyper-parameters is good. The only indication is to avoid hyper-parameters with combinations of small learning rate and epochs with higher batch size and discretization level. Figure 3.17 shows the latent samples generated using the learned parameters of the distribution by the model with the hyper-parameters $(learning\_rate, epoch, batch\_size, level) \rightarrow (0.0055, 250, 32, 5)$.

The main takeaways from this exercise are that decreasing the learning rate will increase

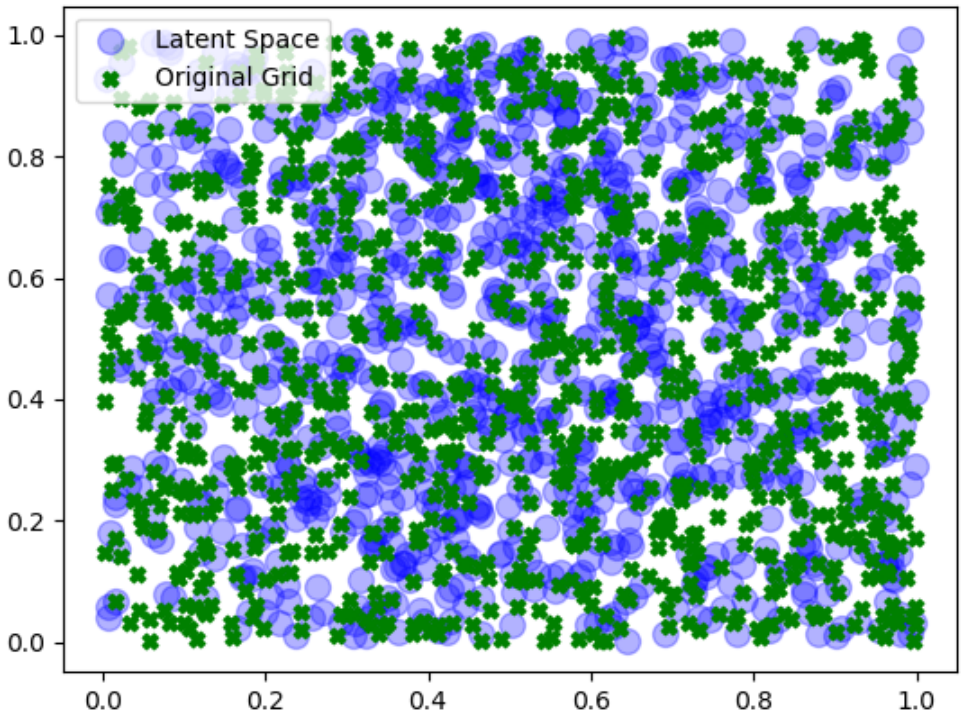


Figure 3.17.: Latent Representation of SGVA with an optimal set of hyper-parameters

the needed number of epochs to maintain the same error as previously. Larger Batch sizes need longer training periods. Finer Discretization methods also require longer training periods. These are broad generalizations that are applicable to a wide variety of deep learning networks and are not exclusive to the Sparse Grid Networks

4. Conclusion

This section explains the conclusions drawn based on the results (3.4) and also describes some further ideas that can implemented to improve the applicability of the Sparse Grid Layer to other problems.

This Thesis establishes the proof-of-concept for the implementation of the Sparse grid in the Deep learning pipeline. We first went through the basics of Sparse Grids, the reason for their existence, their advantages over conventional methods, the theory behind them and finally how they are implemented. Then Deep Learning was covered- the need for its existence, its history, the theoretical background and further implementation in PyTorch. We further delved into the different types of Deep Learning approaches and the Networks that are specialized for these purposes. Then the main idea of the Thesis is presented i.e the use of Sparse Grids in Deep Learning Networks. The Deep Learning Networks taken into consideration are the Autoencoder and Variational Autoencoder Networks. The models are explained with respect to their purpose , architecture and loss functions. Their architectures are then used as the inspiration behind the Sparse Grid Autoencoder (SGA) and the Sparse Grid Variational Autoencoder (SGVA). The resulting process behind these models are explained and their implementation details in PyTorch are also penned out in detail. The Results obtained by these networks on the model problem is presented and Hyper-parameter tuning is performed to get the best possible result.

The Results achieved validate the models. The Sparse Grid Autoencoder was able to achieve a very low error rate and have a general good understanding of the original latent space as it was approximately able to cover the entire latent space. This was achieved after performing Hyper-Parameter Tuning. Many combinations of hyper-parameters although able to bring down the error rate to a satisfactory level, they were unable to capture the original latent space. They instead clump up in certain areas and fail to capture the extant of the space. The Sparse Grid Variational Autoencoder was also able to achieve a very low error rate and able to fully capture the extant of the original latent space. The main difference here with the Sparse Grid Autoencoder is that the the SGVA was able to perform well with many different combinations of Hyper-Parameters. In most of the cases the model had learned the full extant of the latent space and was able to draw samples from most locations in the latent space. The Sparse Grid Variational Autoencoder is a more complex model than the Sparse Grid Autoencoder and thereby took longer to train.With both models performing well on these problems, we feel confident that these models can be extended to more real-world example cases too. The advantage in number of parameters saved would be of great benefit to many scientific problems.

We feel confident that this idea can be further improved upon in different ways. This

thesis only used the Sparse Grid in the Autoencoder and the Variational Autoencoder Networks. There is no such limitation though. They can be generalized across many Deep Learning Networks. From standard Multi-layer Perceptron to the current in-demand Transformers, Sparse Grids can be used. They can also be used in combination with standard Neural Network architectures in order to maximize performance. Since this thesis established proof-of-concept of these Networks, these Networks can be used to tackle more real-world applications in Flow simulations like fluid dynamics and crowd modelling.

Another idea to be explored is that of experiment design. With the implementation of the Sparse Grid in the Autoencoder Network for calculating the low-dimensional representation of inputs, a more efficient method of sampling can be implemented. The input space to the Sparse Grid Autoencoder is vast owing to its high-dimensionality, this can lead to regions of input space which are under representation or no representation in the input data. Using the Sparse Grid Autoencoder this problem can be mitigated. The Sparse Grid in the latent space can inform us of areas of under-representation suing an error function. This error function will involve the Sparse grid points and the latent representations of the inputs. We can then solve an inverse problem using the Sparse grid points to get the high-dimensional representation and thus give a complete picture of the input space. This gives us a methodology of designing future experiments with efficiency. This would be a huge advantage over the conventional Deep learning architectures.

An area of obvious improvement that applies not only to Sparse Grid Networks but in general is the use of Sparse Grids in Hyper-parameter tuning. The method used in this Thesis is a grid based searching algorithm that takes into account the full-grid with all of the hyper-parameters. For a sufficiently large network with many hyper-parameters, this can easily explode the number of training simulations required. The same idea of the "curse of dimensionality" applies here. This time the issue is even more serious as each training requires a significant amount of time leading to a cumulative training time of days and even more. This can be improved by changing the full-grid to a Sparse grid thereby dropping the amount of discrete points used for training simulations which would directly lead to savings in both time and energy.

This thesis hopefully is a starting point for widening the avenue of research into combining Sparse Grids and Deep learning Networks.

Appendix

A. Abbreviations

SGA : Sparse Grid Autoencoder

SGVA : Sparse Grid Variational Autoencoder

PDE : Partial Differential Equations

KL-divergence : Kullback-Leibler divergence.

LSTM : Long short-term memory

CNN : Convolutional Neural Network

GAT : Graph Attention Netowrk

GCN : Graph Convolutional Network

U-Net : U ”shaped” Convolutional Network

GAN : Generative Adversarial Network

DCGAN : Deep Convolutional Generative Adversarial Network

RNN : Recurrent Neural Networks

WGAN : Wasserstein Generative Adversarial Network

CycleGAN :Cycle - Generative Adversarial Network

RBM : Restricted Boltzmann Machine

VAE : Variational Autoencoder

MNIST : Dataset of handwritten integers from 0-9

Bibliography

[1] David Berthelot, Colin Raffel, Aurko Roy, and Ian Goodfellow. Understanding and improving interpolation in autoencoders via an adversarial regularizer, 2018.

[2] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. *Acta Numerica*, 13:147–269, 2004.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[6] HENRY J. KELLEY. Gradient theory of optimal flight paths. *ARS Journal*, 30(10):947–954, 1960.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[8] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

[9] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[11] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

[12] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[13] Yann Lecun, Koray Kavukcuoglu, and Clement Farabet. Convolutional networks and applications in vision. pages 253–256, 05 2010.

[14] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, Jun 1976.

[15] Hadrien Montanelli and Qiang Du. New error bounds for deep networks using sparse grids, 2018.

[16] Dirk Pflu¨ger. Spatially adaptive sparse grids for high-dimensional problems, 2010.

[17] Dirk Pflüger, Benjamin Peherstorfer, and Hans-Joachim Bungartz. Spatially adaptive sparse grids for high-dimensional data-driven problems. *Journal of Complexity*, 26(5):508–522, 2010. SI: HDA 2009.

[18] Joseph Rocca. Understanding variational autoencoders (vaes). *Medium*, 09 2019.

[19] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, Mar 2020.

[20] Concetto Spampinato, Simone Palazzo, Isaak Kavasidis, Daniela Giordano, Nasim Souly, and Mubarak Shah. Deep learning human mind for automated visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[21] Michael Tschannen, Olivier Bachem, and Mario Lucic. Recent advances in autoencoder-based representation learning, 2018.

[22] Paul Werbos and Paul John. Beyond regression : new tools for prediction and analysis in the behavioral sciences /. 01 1974.