Deep Generative Design

A Deep Learning Framework for Optimized Shell Structures

TU Delft
MSc Architecture, Urbanism & Building Sciences
Building Technology Track
Studio: Building Technology Sustainable Design Studio

Student
Stella Pavlidou
Student Number: 5385571

Mentors
Dr. Charalampos Andriotis, Structural Design & Mechanics
Dr. Michela Turrin, Design Informatics

Delegate of the Board of Examiners
Herman de Wolff

ABSTRACT

In an urban context that needs to be constantly adapted to global crises, population movements, climate change and economic crises, designers and engineers strive to configure solutions that respond to multiple criteria. Within this framework, the concept of generative design is gaining more and more ground in the construction field, allowing rapid design space exploration, optimization and decision making for complex design problems.

This thesis implements an experiment in a common design problem such as optimizing the topology of shell structures for structural performance, using an Artificial Intelligence Framework. To implement this experiment a novel dataset consisting of various mesh tessellations is created. The next step is to design a generative workflow that combines unsupervised and supervised learning along with a Gradient Descent Algorithm for pattern generation, structural performance estimation and optimization. A Variational Autoencoder is trained to generate new mesh tessellations and a Surrogate Model is used to predict the structural performance of the decoded designs. Finally, a Gradient Descent Algorithm searches the latent space of the Variational Autoencoder for optimum solutions.

The results show that the proposed Artificial Intelligence workflow is able to generate novel and structurally better performing solutions that those existing in the training dataset. The findings of this thesis indicate that Artificial Intelligence can be successfully integrated into the concept of Generative Design to optimize shell structures.

Keywords: Generative Design, Structural Optimization, Mesh Shells, Tessellations, Artificial Learning, Machine Learning, Deep Learning, Unsupervised Learning, Supervised Learning, Gradient Descent, Finite Element Method

ACKNOWLEDGMENT

Completing these two years of my studies, I feel extremely fortunate for everything I have learned, for every course and lecture that shaped my perspective, allowing me to discover new creative and innovative fields.

I can't think of a topic closer to my interests nor having better guidance than my mentoring team. I want to thank both my mentors for introducing Artificial Intelligence to the curriculum of Building Technology, broadening my horizons and allowing this thesis to exist. I am infinitely grateful for every minute of valuable consultation with Charalampos Andriotis, who patiently and insightfully guided me to the completion of my thesis. I am also thankful for the all the apt remarks and encouragement I received from Michela Turrin.

Throughout this process I had the unlimited support of my family and friends. I could not have attempted this step without my mother and sister motivating me to challenge myself, without the love of Ifigeneia and Anastasia and the comfort of knowing that Maria and Antonis were by my side when I needed them. I would also like to thank Sean Virk for his help and support during the last year as well my AI buddy, Namrata Baruah who reminded me that “we can do it”.

ABBREVIATIONS

AI:	Artificial Intelligence
ML:	Machine Learning
VAE:	Variational Autoencoder
GD:	Gradient Descent
NN:	Neural Networks
FEM:	Finite Element Method
KL:	Kullback-Leibler divergence
GAN:	Generative Adversarial Network

CONTENTS

1. INTRODUCTION

1.1 Background

The built environment industry is responsible for massive amounts of energy usage and employ millions. Sustainable design and construction is a necessity. Early stage decisions affect significantly the final outcome with respect to cost, structural performance, assembly time, etc. Generative tools that allow a deeper exploration of the design space are desirable.

Topology exploration and optimization is a critical field, which includes early-stage decisions, affecting significantly the performance of the final structure as well as its aesthetics. Using the power of computation, we can define goals and set constraints to generate designs that meet both our qualitative and quantitative criteria. That can be a time consuming and computationally heavy process. Therefore, technologies that allow a deep investigation of the design space can be powerful and decisive tools.

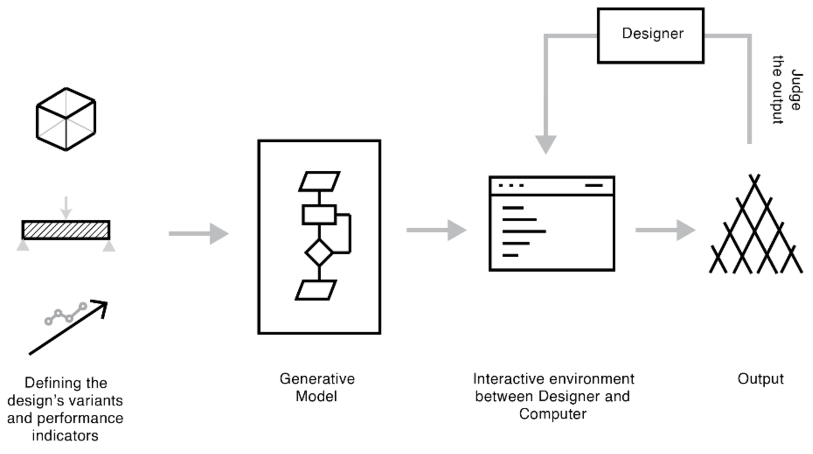
Artificial Intelligence (AI) is altering the way of learning and problem-solving in most scientific fields. As the world faces urgent and complex challenges, AI comes to the forefront of research to improve the decision-making process.

1.1.1. Generative Design

The concept of Generative Design allows for a more integrated workflow between designer/engineer and computer and a deeper exploration of the design space beyond the traditional design techniques. Inspired by nature's evolutionary approach it as the iterative process that assesses the design's variants to fit specific design criteria and finds the optimum solution. (McKnight, 2017)

Creating a Generative Workflow involves the following steps:

Step 1	In this step design's variants and the performance indicators are specified.
Step 2	A algorithmic generative model produces a large amount of design options based on the step 1.
Step 3	An interactive environment is created where the de sign/ engineer can update the design's variants.
Step 4	Upon receiving the results the parameters and goals are adjusted and the generative design system will then iterate until the most relevant solution is found.
Step 5	The final design is received as an outcome.


Designer
Judge the output
Defining the design's variants and performance indicators
Generative Model
Interactive environment between Designer and Computer
Output

1.1.2. Importance of topology exploration in Shell Structures

Shell structures have evolved over time from stone masonry domes to brick and concrete structures as well as timber and metal networks. They usually consist of beam networks that integrate cladding systems. Thanks to their curvature they are stiff structures that can enclose large spaces. Modern applications of shell structures include light weight skins with visible beam networks.

Examples of shell structures

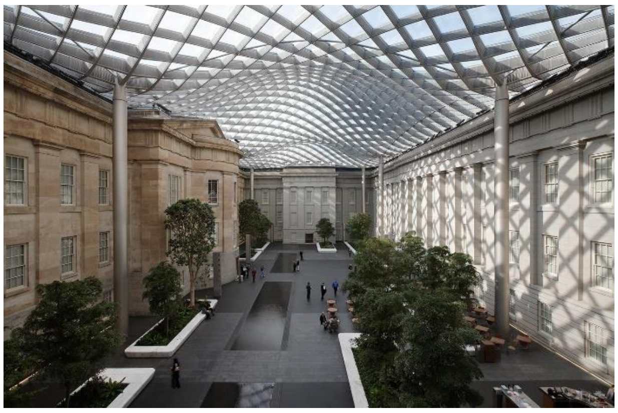


Figure 1.1. Robert and Arlene Kogod Courtyard (Young)

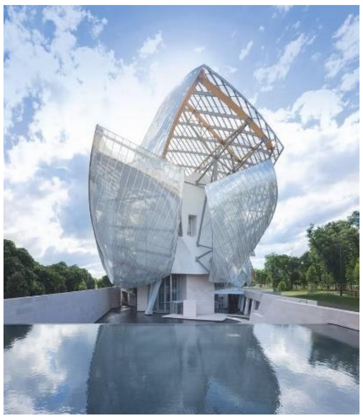
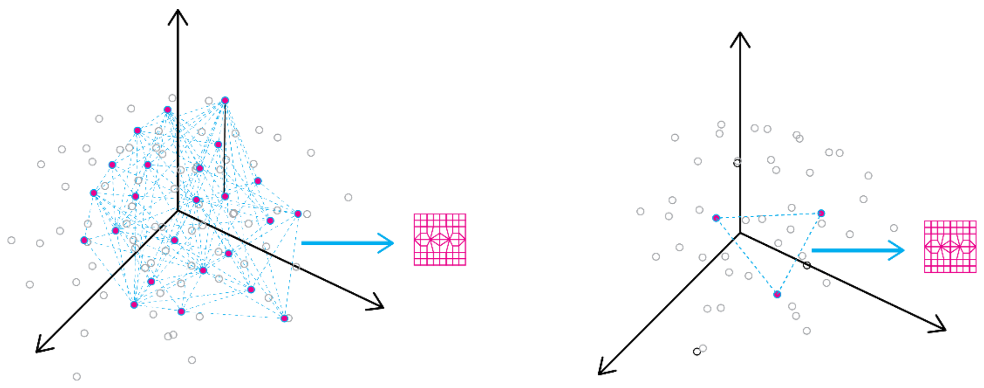


Figure 1.2. Robert and Arlene Kogod Courtyard (Gehry Partners, 2014)

The design process of these structures begins with the investigation of mesh tessellation options and then structural verifications using finite element method (FEM). Their topologies define architectural aesthetics, structural performance as well as cost, assembly complexity and time. The pattern exploration of their topologies is a time-consuming process yet matters. Therefore, tools that ease and improve this process can be immensely helpful. Computational tools can be integrated in the design process for topology finding patterns but heuristics is often needed (Oval et al., 2019).

1.2 Problem Statement and Design Assignment

The topology of shell structures is critical and affects cost, assembly time, structural performance, and aesthetics. Designers and engineers need conceptual and practical tools to explore it. There are many variants that describe a design and exploring the design space for optimum solutions is a time-consuming process. Generative Models that integrate Artificial Intelligence can minimize the number of variants in a smaller sized space and could potentially be integrated in an optimization workflow.



1.3 Research Questions

Main Question

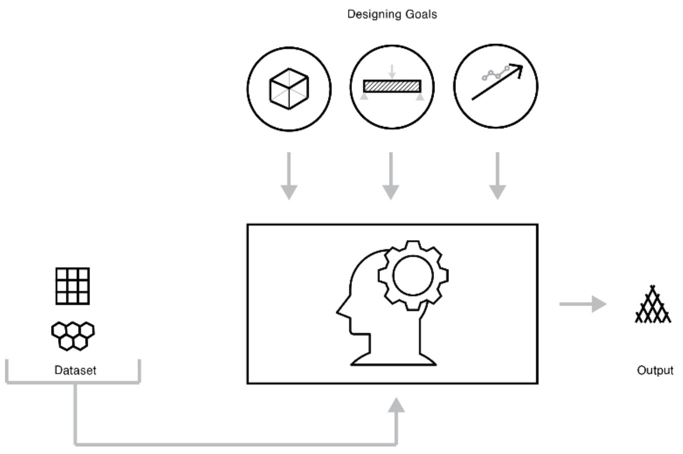
According to the problem statement, mentioned above, the main research question is if an AI based framework can generate new structurally effective solutions, in relation to the dataset that was used for training. This would prove that AI can be a powerful creative assistant for designers and engineers, and could potentially help expand the possibilities of generative design.

Sub-question

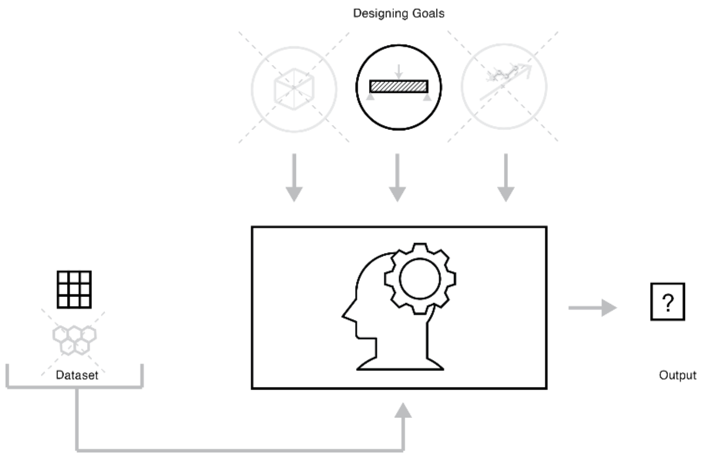
- Can a Variational Autoencoder be trained to generate mesh tessellations?
- What form of data can be used to train a Variational Autoencoder to generate mesh tessellations?
- Can a surrogate model learn to predict the structural performance of encoded data occurring from samples describing truss shell structures?
- Can a surrogate model learn to predict the structural performance of decoded data occurring from samples describing truss shell structures?
- Can a Gradient Descent Optimizer propagate back to encoded data to search for optimum solutions?

1.4 Objectives and Boundary Conditions

The purpose of this thesis is to prove that AI can be a powerful assisting tool for designers and engineers. A helpful AI workflow would request for certain boundary conditions as an input (such as shape, structural performance, etc) and produce effective solutions.



Due to time limitation this thesis is restricted in terms of input criteria. The criterion for the suggested workflow is structural performance for mesh shell patterns with specific boundaries.



This workflow includes a generative model able to produce mesh tessellations with a specific boundary and a surrogate model able to predict a design's structural performance. The architecture of the generative model is that of the Variational Autoencoder, an Artificial Neural Network architecture introduced by Diederik P. Kingma and Max Welling (Kingma & Welling, 2014).

A Gradient Descent Optimizer is integrated in the workflow to search the design space for effective solutions.

A training dataset is needed that according to bibliography should exceed 1000 samples. The samples need pre-processing to be used in training machine learning models. Appropriate options for the form of the training dataset will be explored.

Computational power is crucial in training machine learning models. For this reason, TU Delft's supercomputer "Delftblue" is used.

1.4 Methodology

Literature Review: The first step is to investigate existed research examples. The methods for pattern generation in shells structures must be defined. Then existed research for generative design and optimization using AI models will be reviewed. During this process, the computational tools that will be used have to be specified. To familiarize with these tools and models, an experimentation with existed tutorials is needed.

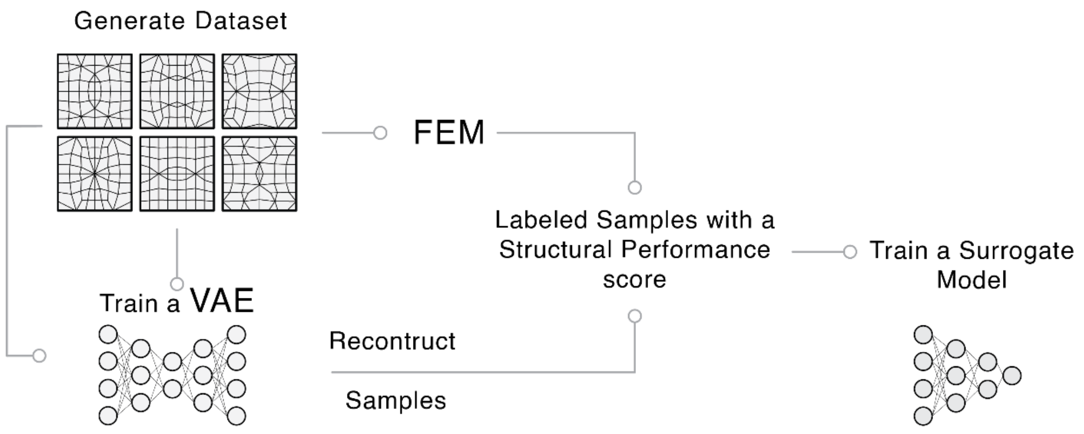
Design the Training and Optimization Framework: Based on the literature review a framework will be proposed that might be altered slightly during the training process

The proposed steps are:

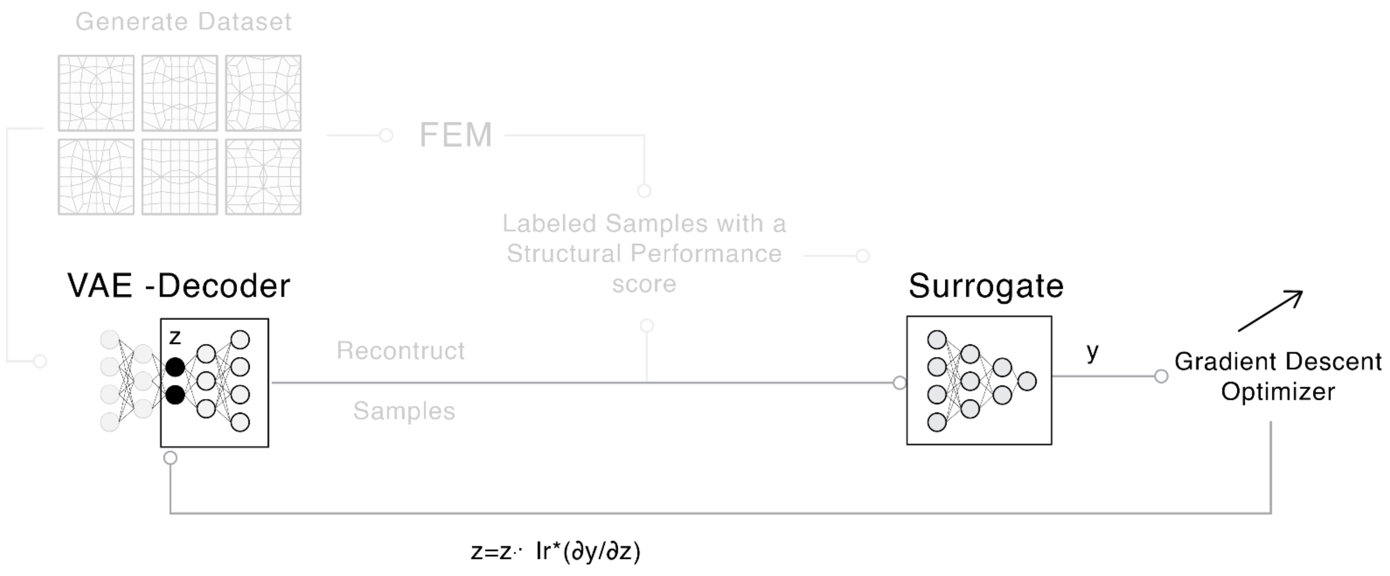
- Create the dataset and measure its structural performance (performance indicators will be displacement, utilization and the structure's mass), using Finite Element Method software (FEM).

- Train a Variational Autoencoder with the produced data.

- Decoded Data is used to train a surrogate model that is able to predict a design's structural performance.



- Use a Gradient Based Optimizer that propagates back to the encoded data to search for best solutions.


Generate Dataset
FEM
Labeled Samples with a Structural Performance score
VAE -Decoder
z
Recontruct
Samples
Surrogate
y
Gradient Descent Optimizer
z=z- lr*(∂y/∂z)

- Produced results are assessed to see if more effective solutions are generated.

2. LITERATURE REVIEW

2.1 Artificial Intelligence, Machine Learning and Deep Learning

In the 21st century, the amount of digital information created is astonishing. Since 2020 we produced 90% of the world's data(Bradshaw, n.d.). Massive amounts of information need to be processed daily, making our lives increasingly dependent on learning algorithms.

Machine learning (ML) is a part of computer science and allows machines to learn from data without being explicitly programmed. It is often used as a synonym of Artificial Intelligence although it is actually its subfield. AI the general term used to classify systems that mimic human intelligence whereas ML is more about extracting knowledge from the data. It can be used to predict, automate, perfect tasks and generate new systems.

Deep learning is a subset of machine learning and it is based on artificial neural networks able mimic the learning process of the human brain.

(Manager, 2020)

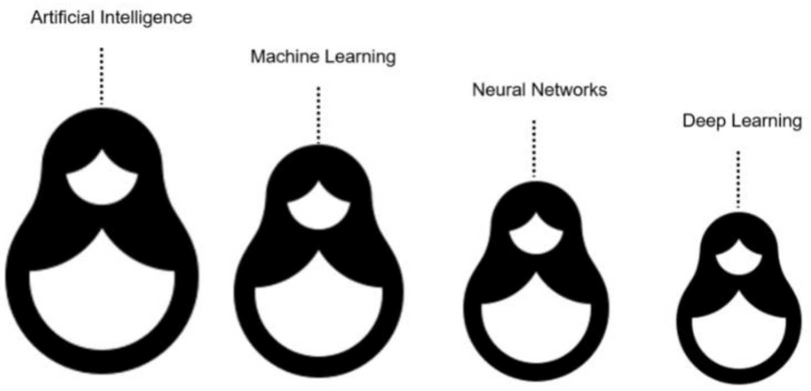


Figure 2.1 We can understand AI, Machine Learning and Neural Networks and Deep Learning like Russian nesting dolls (Manager, 2020)

2.1.1 Neural Networks

Neural networks (NNs) are the heart of deep learning algorithms. Their name and structure are inspired by the biological neural networks that brains have. They consist of or artificial neurons or node layers. They have an input layer, one or more hidden ones and an output layer. (What Are Neural Networks?, 2020)

The following image describes how an artificial neuron is constructed. Each one has an **input** and an **output**. The input consists of values for which the NN needs to predict the output value. The input data are assigned **weights** which describe their importance. The purpose of the **summation function** is to bind the inputs and their respective weights together and find their Sum. **Bias** is used to shift the Sum towards left or right. An **activation function** transforms the output of the node and decides whether the neuron can be activated. (Sharma, n.d.),(Ganesh, 2020)

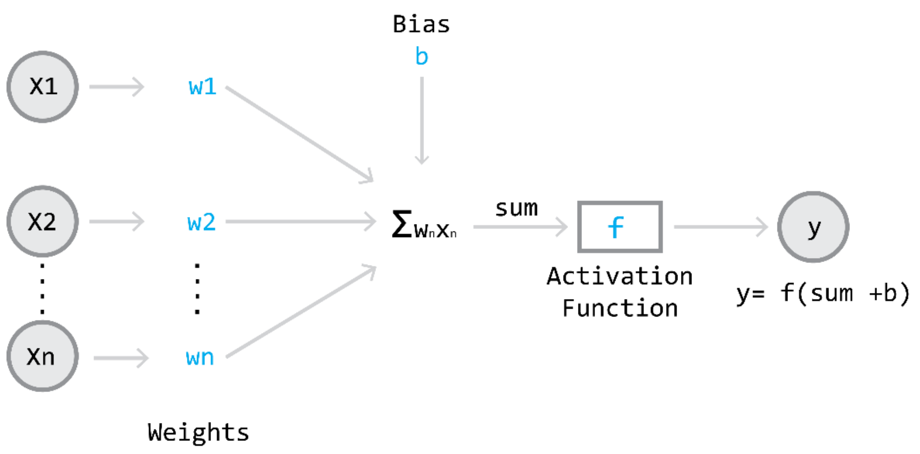
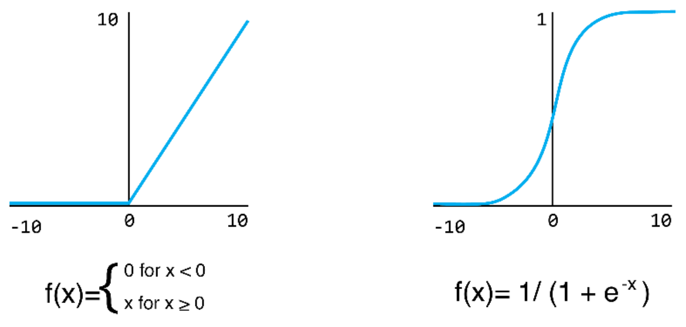


Figure 2.2 Inside an artificial neuron

Activation Functions

In artificial neuron the activation function decides whether the neuron can be activated or not and defines its output. The **ReLU** and **Sigmoid** are two non -Linear activation functions that are used in this thesis. The function of **ReLU** will output the input directly if it is positive, and zero if it is negative. It is probably the most popular activation function due to its computational efficiency and fast convergence. A problem with **ReLU** is that the outputs go far away from zero. That problem can be treated with **Sigmoid**, a function known to be very good for classification problems and whose output always ranges between 0 and 1. **Sigmoid** however is computationally heavy and slow converged. (Vinodhkumar, 2020a)



$$f(x)=\begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases} \qquad f(x)= 1/\left(1 + e^{-x}\right)$$

Figure 2.3 From left to the right: ReLu and Sigmoid activation functions.

Learning

When the training dataset is too big, we cannot process it all at once. The dataset is therefore divided into a number of **batches** or sets, that pass forward through the NN's nodes. This process is also called **propagation.** The NN calculates the error between the expected output and the NN's output, called **loss function**. The **backpropagation** algorithm computes the gradient of the loss function with respect to the weights and an **optimizer** adjusts them. The process of propagation and backpropagation through which the weights of the nodes are adjusted to fit the input to the expected output is called **learning.** When a whole dataset has entered the neural network we say that an **epoch** is completed. The goal of the NN is to decrease this loss function, as much as possible to reach its **convergence**. (Vinodhkumar, 2020b) (Daoud, 2020) (Stojiljković, n.d.), (Sharma, 2017)

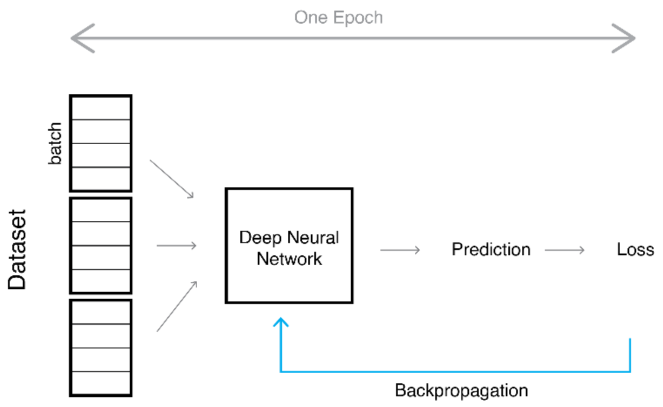


Figure 2.4 Learning process.

Deep Learning Algorithms

The depth, of neural networks is what defines if it is a simple or a deep learning NN. If the number of node layers is more than 3 then we talk about a deep learning algorithm. (Manager, 2020).(Sharma, n.d.) (What Are Neural Networks?, 2020).

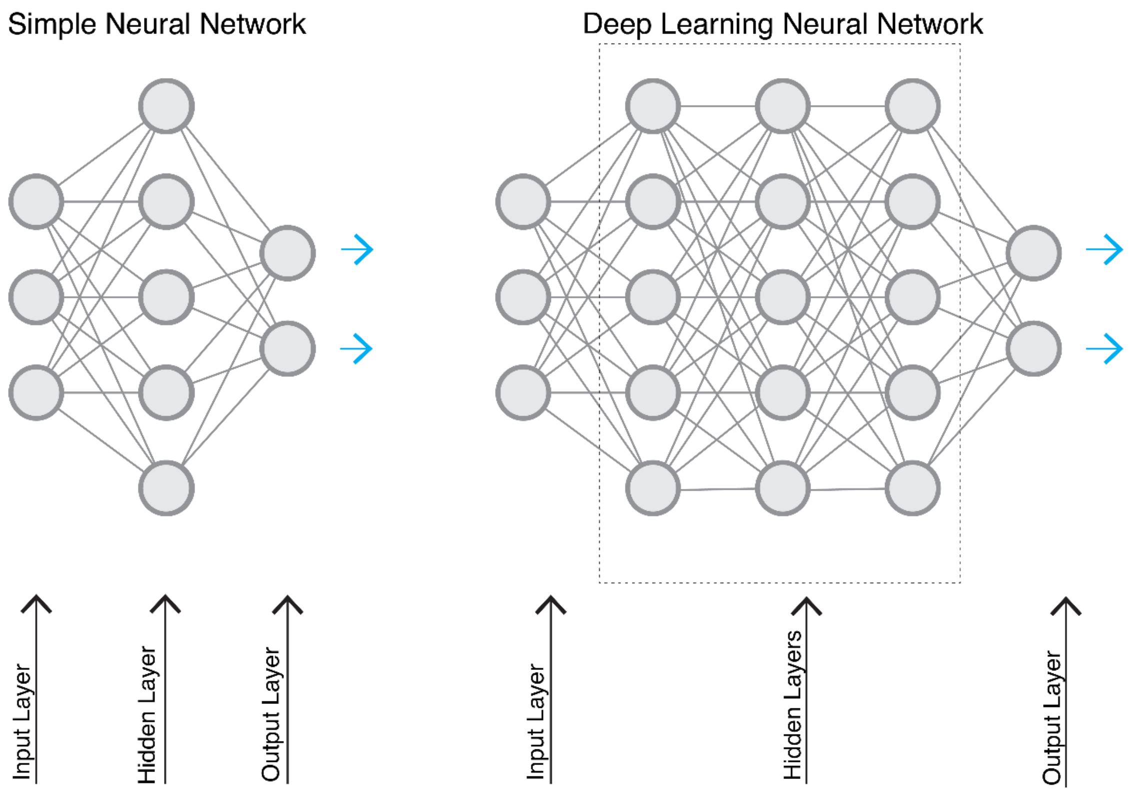


Figure 2.5 A Simple NN versus a Deep Learning NN.

2.1.2 Types of Neural Networks

Feed Forward Neural Network
A Feed Forward NN is the first and the simplest type of an NN in which the connections of the nodes move to a single direction and don't form a circle or loops.(Feed Forward Neural Network, 2019)

Recurrent Neural Networks
The opposite of a Feed Forward NN is the Recurrent NN which has loops. The advantage of this type of NNs is that it remembers previous inputs, therefore they can be used when training data occur from a series of observations over time (time-series data). (What Are Neural Networks?, 2020)

Fully Connected Neural Network:
A Fully Connected NN (FCNN) consists of a series of fully connected layers or dense layers. These layers connect every neuron in one layer to every neuron in the next one. The advantage of fully connected networks is that they are "structure agnostic.", meaning that no special assumptions need to be made about the input (for example, that the input consists of images or videos). This allows them to be used broadly, for more general purposes. However they tend to have weaker performance than special-purpose networks.(Ramsundar & Zadeh, 2022)

Convolutional Neural Networks
Convolutional Neural Networks (CNNs) are mainly used for image recognition, pattern recognition and computer vision. These networks apply principles from linear algebra, to identify patterns within an image. (What Are Neural Networks?, 2020)

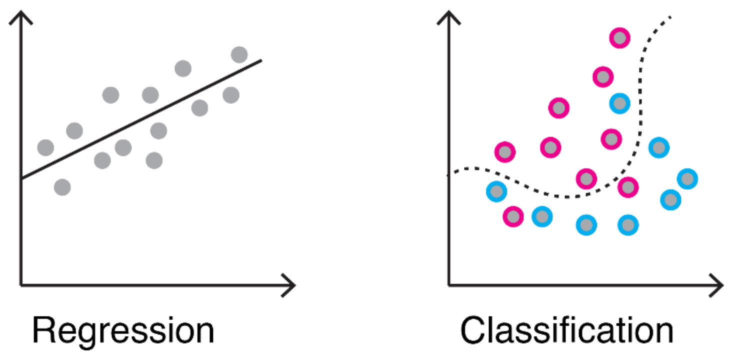
2.1.3 Supervised and Unsupervised learning

When it comes to machine learning there are two types:

- Supervised learning, used for prediction tasks
- Unsupervised learning, used for generation

a. Supervised Learning

In supervised learning the training is being operating with a prior knowledge of the outcome target. Supervised learning can be done for classification or prediction problems. Classification NNs are used to fit training data to discrete values like a true or false value, or gender, etc while regression NNs try to fit them to continues values like price, time, etc. (Soni, 2022)



Surrogate Models

Performance is critical aspect of design. Computer simulations can calculate a system's final behaviour. Engineers rely on those simulations to perform sensitivity analysis for multiple parameters, optimization and risk analysis. Simulations can be heavy computationally. Heavy simulations can be replaced by a statistical model, called a surrogate model. It is a case of supervised learning, meaning that we are using labelled data. Through this model we can predict the final output. (Guo, 2020) (Guo, 2020)

b. Unsupervised Learning

In contrast to supervised learning where data are labelled, unsupervised learning exhibits a self-organization that aims to understand the pattern of the given samples and build a compact internal representation of the data

Variational Autoencoders

An autoencoder is a case of unsupervised learning, used for dimension reduction. Its neural network consists of two pairs of neural networks: an **encoder** and a **decoder**. The encoder has input of nodes and consists of hidden layers that are able to reduce the number of nodes describing the input data.

Encoded data are directed to a hidden layer called bottleneck or **latent space**. This layer has a lower dimensionality than the original input. The decoder performs the inverse process. It takes a vector point from the compressed representation in the latent space and reconstructs a corresponding output.

.

This architecture allows keeping key information of a large database and storing it using less memory.

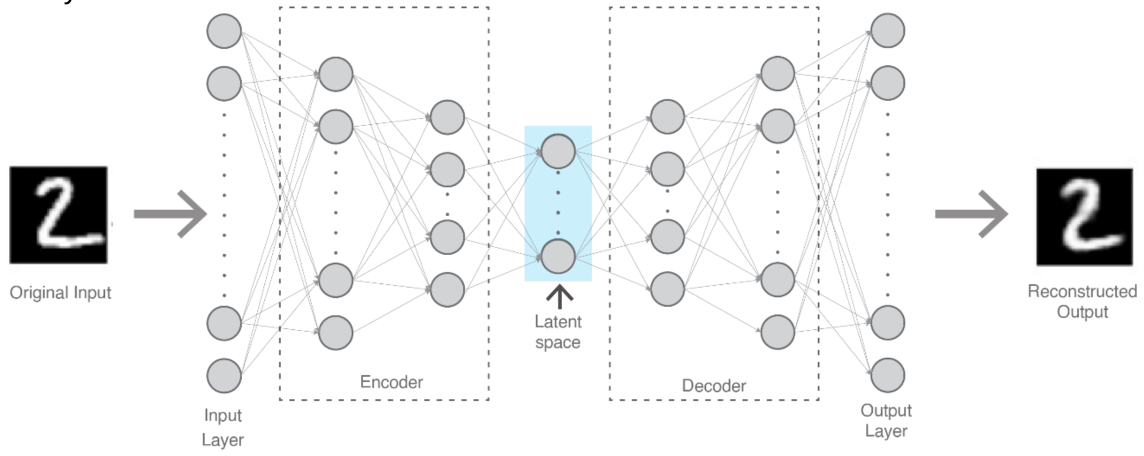


Figure 2.6 An autoencoder.

Autoencoders can be converted to deep learning generative models called **Variational Autoencoders** (VAEs). These models sample data from the latent space and reconstruct them through the decoder producing realistic systems such as images, texts, sounds, etc. But they use a slightly different encoding-decoding process: instead of encoding an input as a single point, it is encoded as a distribution over the latent space.

A problem that comes with the dimensionality reduction is that the trained latent space may not be regular, having gaps between clusters. The lack of regularity means an autoencoders cannot generate new content. If the sample point is for instance from a gap the output can be unrealistic. The VAE is an autoencoder that ensures that the encoding distribution is regularized during the training process, allowing us to generate new data.

The loss function used when training a VAE has two terms: a “reconstruction term” (on the final layer), that tends to make the encoding-decoding scheme as performant as possible, and a “regularization term” (on the latent layer), that tends to regularize the organisation of the latent space. This regularization term is called the **Kullback-Leibler divergence** (kl divergence).
(Rocca, 2019).

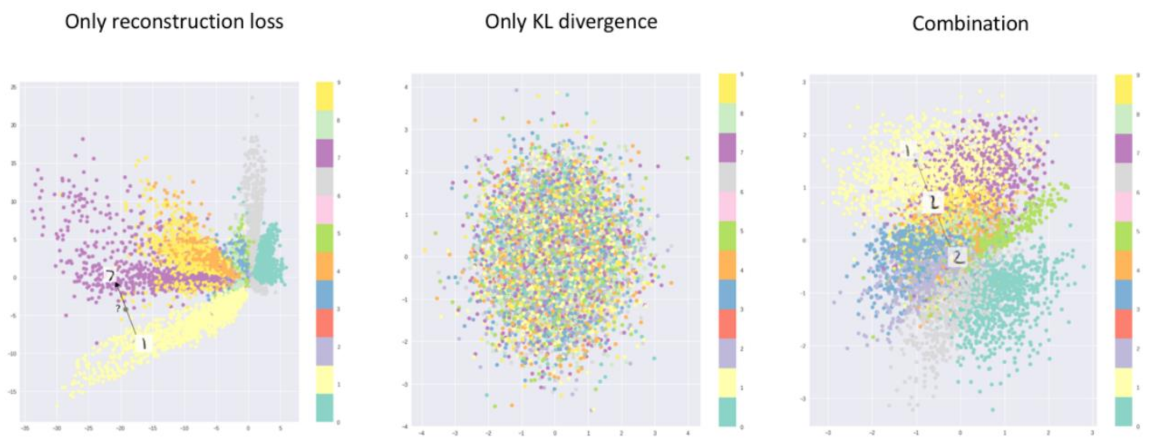


Figure 2.7 From left to the right: Only reconstruction loss, Only KL divergence and both (Shafkat, 2018)

2.1.4 The Learning Curves

The learning curve can help us access how well is the model learning over the time of training. The state of the model’s performance is evaluated through how well it has learned to minimize the loss function. The **Train Learning Curve** represents the learning curve how well it has learned to predict the dataset’s output. We can also exclude a part of the dataset that will give as the **Validation Learning Curve**, that will demonstrate how well has the model learned to generalize.

There are three states that can describe a model’s training :

- Overfit.
- Underfit.
- Good Fit

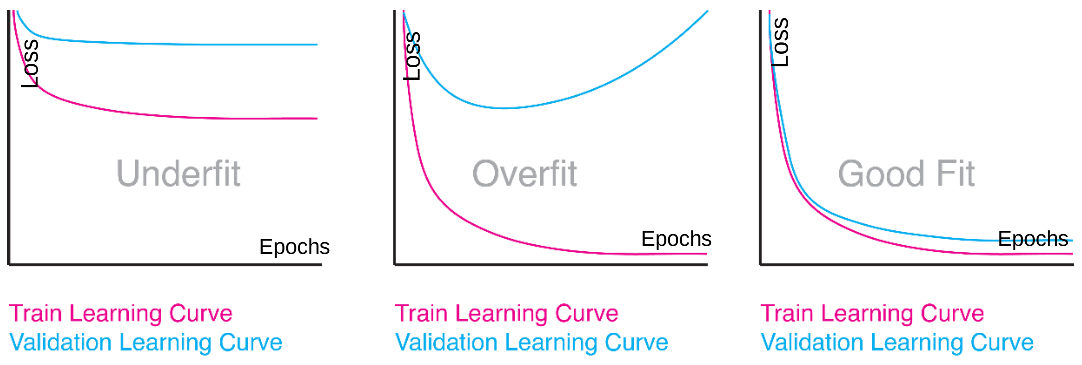


Figure 2.8 Learning Curves.

Overfit happens the model has learned to predict the outcome of the dataset (the model fits exactly to its training data) but is not able to generalize to new data. The loss function in this case drops for the training data and increases for the validation dataset.

Underfit occurs when the model fails to learn. In this case both the train and the validation learning curve don't drop enough

When no overfitting nor underfitting occur and the model fits well to the dataset and the validation dataset then we have accomplished a **Good Fit**.
(Brownlee, 2019)

2.1.5 Gradient Descent Optimizers

The Gradient Descent is an iterative algorithm, used for optimization to find the best result of a function (minima of a curve).

There are different instances of Gradient Descent Based Optimizers:

- Batch Gradient Descent or Vanilla Gradient Descent or Gradient Descent (BGD)
- Stochastic Gradient Descent (SGD)
- Mini batch Gradient Descent (MB-GD)

Batch Gradient Descent

The BGD is the most basic and used optimizer. It starts from an arbitrarily chosen position of the point or vector $\Theta = (\Theta_1, \ldots, \Theta n)$ and moves it iteratively in the direction of the fastest decrease of the loss function.

A GD is described by the following formula:
$\Theta j = \Theta_j - a\frac{\partial}{\partial \Theta j} J(\Theta_0, \Theta_{n,})$

α : Learning rate that determines how large the update or moving step is.
J : Loss function
Θ: Parameter to be updated

The BGD is easy to implement but requires a lot of memory as the entire dataset is loaded at a time to compute the derivative of the loss function.

Stochastic Gradient Descent

To overcome the problem of high memory usage that the BGD needs, the SGD comes with a modification. BGD considers the entire dataset to compute the gradient. SGD however picks a "random" instance of training data at each step and then computes the gradient. Therefore, the model's parameters are updated after the loss computation on each training set. Consequently, the steps taken towards the minima can be very noisy.

Mini batch Gradient Descent (MB-GD)

The MB-GD is an extension of the SGD algorithm and is considered the best among all the variations of gradient descent algorithms. This algorithm divides the dataset into various batches and after every batch, it updates the parameters. MB-SGD remains noisier compared to GD and takes a longer time to converge but requires less memory. (GOYAL, n.d.),(Stojiljković, n.d.), (Sharma, 2017)

2.2 Meshes

2.2.1 Mesh Data Structures

A Mesh network is a topology that describes the connections between nodes or vertices to edges and faces. A mesh is also a manifold if every edge is adjacent to one boundary or two faces. There are two common data structures that describe meshes:

- Face-Vertex Lists
- Half-Edge Data Structure

Face – Vertex Lists

This is the simplest representation of a mesh. It consists of an ordered list of vertices with their coordinates and a list of faces with each face being a list of the indices of the vertices that it occurs from. (Löffler & Vaxman, 2016),(L. Chen, 2014)

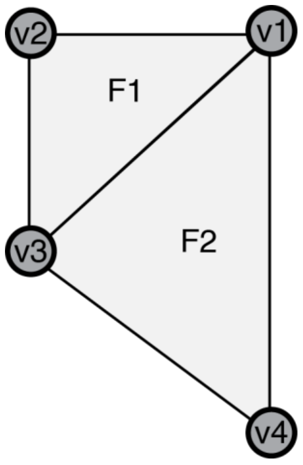


Figure 2.9 A simple mesh.

Vertex	Coordinate
v1	(x_1,y_1,z_1)
v2	(x_2,y_2,z_2)
v3	(x_3,y_3,z_3)
V4	(x_4,y_4,z_4)

Table 2.1 List of Vertices

Vertex	Coordinate
F1	v1, v2, v3
F2	v1, v3, v4

Table 2.2 List of faces

Half-Edge Data Structure

This is a data structure that encodes more information about the structure of a mesh. First we must explain the meaning of half-edge. A half-edge describes the connection of two vertices in a single direction meaning that an edge consists of two edges. Each half-edge references to one starting point assuming a counter-clockwise order.

Half-Edge Data Structure consists of:

- A list of vertices that includes its coordinates and one outgoing half-edge.
- A list of half edges that stores the origin vertex, the twin half-edge, the next half-edge and the previous.

(Löffler & Vaxman, 2016)

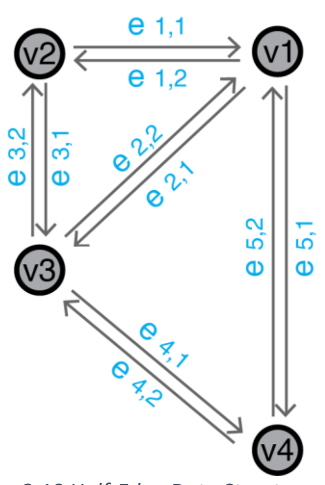


Figure 2.10 Half-Edge Data Structure.

Vertex	Coordinate	Outgoing Edge
v1	(x_1,y_1,z_1)	e 1,2
v2	(x_2,y_2,z_2)	e 1,1
v3	(x_3,y_3,z_3)	e 3,2
V4	(x_4,y_4,z_4)	e 4,2

Table 2.3 List of Vertices

Half-edge	Origin	Twin	Next	Previous
e 1,1	v2	e 1,2	e 2,1	e 3,2
e 1,2	v1	e 1,1	e 3,1	e 2,2
e 2,1	v1	e 2,2	e 4,1	e 1,1
e 2,2	v3	e 2,1	e 1,2	e 3,1
e 3,1	v2	e 3,2	e 2,2	e 1,2
e 3,2	v3	e 3,1	e 1,1	e 4,2
e 4,1	v3	e 4,2	e 4,2	e 2,1
e 4,2	V4	e 4,1	e 3,2	e 1,1
e 5,1	v1	e 5,2	e 1,2	e 4,1
e 5,2	V4	e 5,1	e 4,2	e 1,1

`

Table 2.4 List of half-edges

(Faces can also be stored in the Half-Edge Data Structure but are optional)

2.2.2 Graph Data Structures: Adjacency Matrices

The connections of non-linear data can be described through Graphs. Graphs consist of a set of nodes or vertices and a set of edges that connect them.

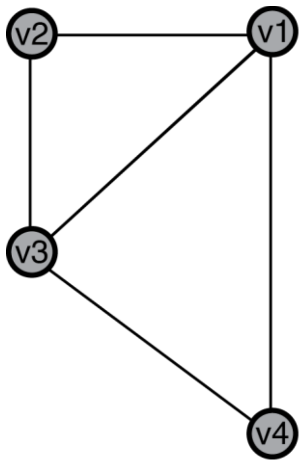


Figure 2.11 Graph Data structure.

```
Vertices=  [v1, v2, v3, v4, v5]

Edges= [[v1, v2,], [v2, v3,], [v3, v1,],  [v1, v4,],  [v3, v4,]]
```

These networks can be described in the form of adjacency matrices. These matrices have rows and arrays that respond to the vertices. An undirected connection (an connection with no direction) between v1 and v2 means that the cells (v1,v2) and (v2,v1) are assigned a value of 1. No connection as that between v2 and v4 means that the cells (v2,v4) and (v4,v2) have 0 value.(Millan & Ochoa, 2020),(Sauras-Altuzarra, 2022)

	V1	V2	V3	V4
V1	0	1	1	1
V2	1	0	1	0
V3	1	1	0	1
V4	1	0	1	0

Table 2.5 Adjacency Matrix

2.3 Mesh Tessellations

The tessellation process includes two essential steps:

Coarse control mesh:
The designer starts by defining a coarse control mesh

Method of subdivision:
The designer can then apply multiple algorithms for subdividing the faces of the coarse control mesh

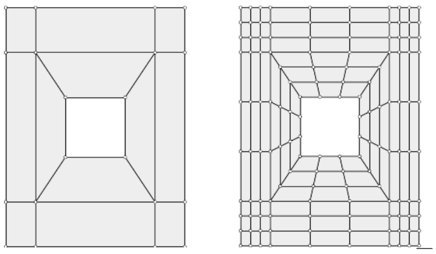


Figure 2.11 A coarse control mesh and the application of the quad subdivision (Oval et al., 2019)

2.3.1 Methods of subdivision

Loop subdivisions

There are various mesh-subdivision loop schemes. Catmull-Clark subdivision produces quadrilaterals faces. The Doo-Sabin subdivision creates four faces and four edges (valence 4) around every new vertex in the refined mesh. The Butterfly/Loop subdivision introduces a smooth subdivision scheme for triangle meshes (Bennett et al., 2003).

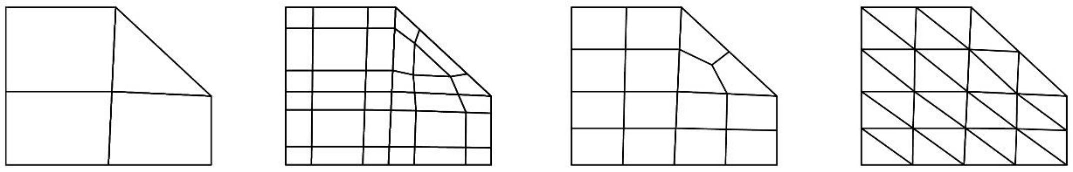


Figure 2.12. From left to the right: coarse control mesh, Catmull-Clark subdivision , Doo-Sabin subdivision, Butterfly/Loop subdivision

Convey operators

Convey operators can be used to transform the subdivision pattern of the surface by keeping the same symmetry. Operators like Dual, Ambo, Kis and Truncate replace the vertices, edges and faces of an original mesh with a combination of new vertices, edges and faces. They can also be combined to form more complex ones(Shepherd & Pearson, 2013).

In the images below the blue pattern is the original subdivision and the white one the new after the application of the convey operator.

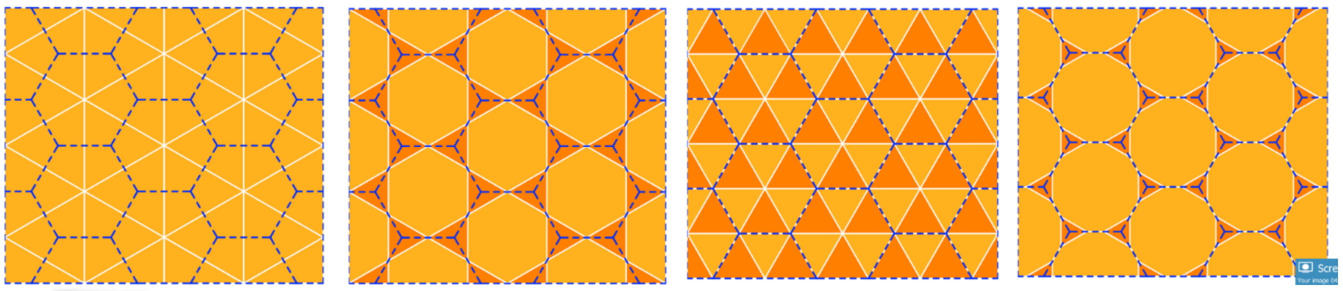


Figure 2.13 For left to right: Dual, Ambo, Kis, dual operation, followed by a Kis, followed by another Dual (Shepherd & Pearson, 2013).

Meshes with a Quad-Based Topology

This thesis is focused on meshes with Quad-Based Topology. We can distinct three main categories for these meshes

Quad meshes: They consist of quad faces, meaning that every face list four vertexes

Coarse quad meshes: A mesh strip can be parental for another quad mesh, creating a varied densification of the structure

Pseudo-quad meshes: Most of the faces are quads with some being pseudo-quad meaning that are geometrically like triangles but topologically like quads. (Oval et al., 2019)

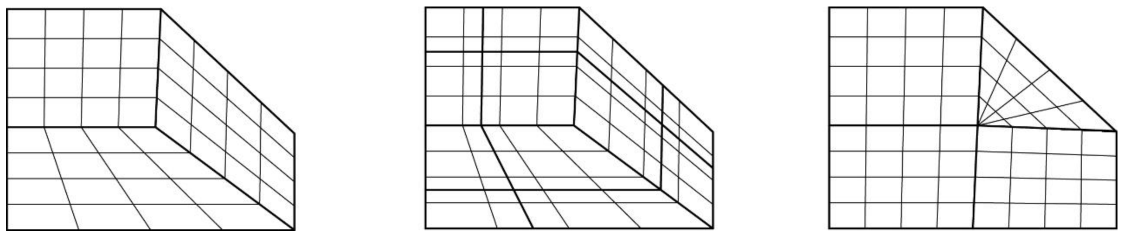


Figure 2.14. From left to the right: Quad meshes, Coarse quad meshes, Pseudo-quad meshes

2.3.2 Singularity

Singularities in meshes are vertices that have irregular valency vertices. We can specify the mesh singularities on quadrilateral meshes by defining a coarse control mesh(Fogg et al., 2018). From each singularity with valence n we can trace out n curves, consisting of sets of edges that end at other singularities or the boundary of the mesh. We refer to these curves as the separatrices(Xu et al., 2020).

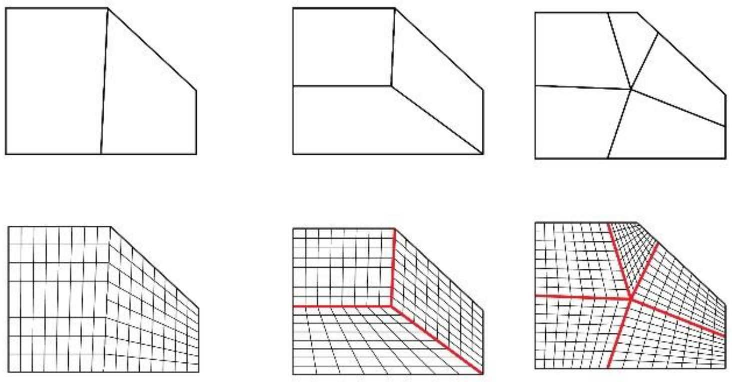


Figure 2.15. Different coarse meshes that result into meshes with different separatrices and singularities

According to the behaviour of these separatrices and the number of singularities we can classify the quad meshes in four categories:

1. Unstructured quad-mesh, where a large part of its vertices are singularities

2. Valence semi-regular quad-mesh. Here the number of singularities is few, but the separatrices have a complicated behaviour.

3. Semi-regular quad-mesh: The separatrices divide the quad-mesh into several topological rectangles, the interior of each topological rectangle is regular grids.

4. Regular quad-mesh: There are no singularities.

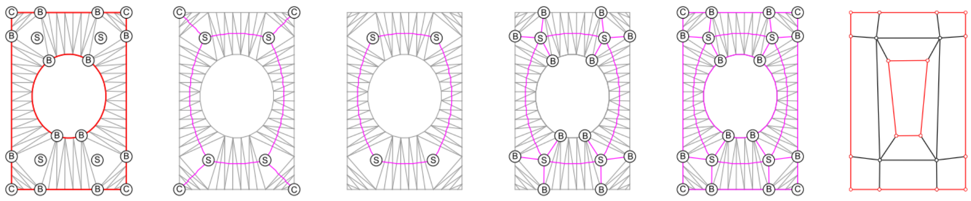
(W. Chen et al., 2018)

2.4 Examples

2.4.1 Feature-based topology finding of patterns for shell structures

R. Oval[a,b,*], M. Rippmann[b], R. Mesnil[a], T. Van Mele[b], O. Baverel[a], P. Block[b]

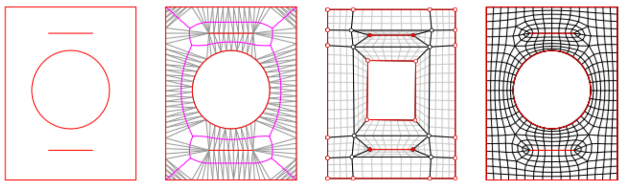
In his research R. Oval proposes a computational method based on singularity meshes to design quad-based mesh tessellations. The workflow's input can be curves or points and the mesh's curve boundary. This input results to a medial axis and a coarse control mesh. A quad subdivision follows. The author provides a python library integrated in the COMPAS python library for geometry processing.

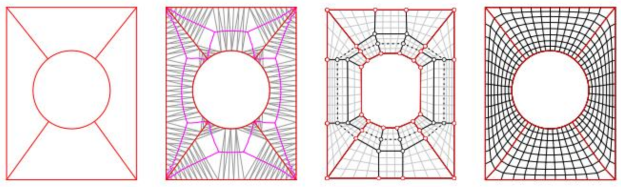

(a) Key points (b) Medial axis (c) Pruning (d) Grafting (e) Closing (f) Coarse mesh

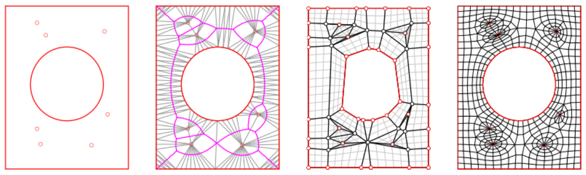
Steps:

1. **Input:** Curves or Points
2. **Medial Axis**
3. **Coarse Mesh**
4. **Quad-based subdivision:** Application of a quad-based subdivision algorithm

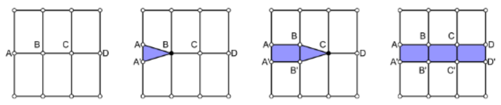
The different inputs result in various patterns

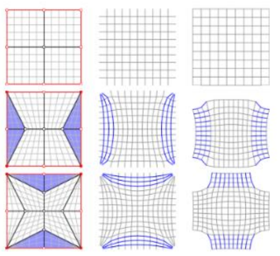

(a) Input (b) Medial axis (c) Singularity mesh (d) Pattern

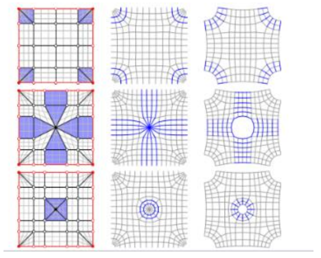

(a) Input (b) Medial axis (c) Singularity mesh (d) Pattern


(a) Input (b) Medial axis (c) Singularity mesh (d) Pattern

Pole points are points with a high valency. These points attract forces and are harder to materialise. They can be controlled during the skeleton-based generation or by adding and deleting strips at the coarse mesh.







Studying the case of the roof of the British Museum, the author assesses the structural performance of the shells by exploring mesh singularity scenarios for quad-based meshes. The different singularities or their absence seem to affect the structural performance of the shell roof.

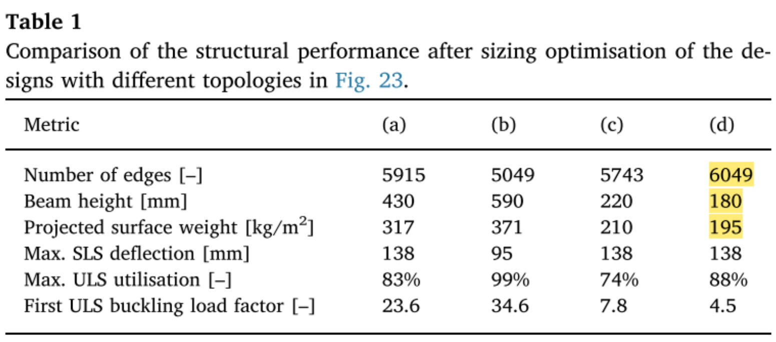
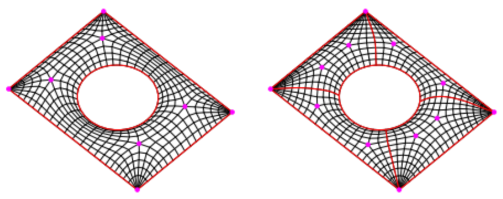
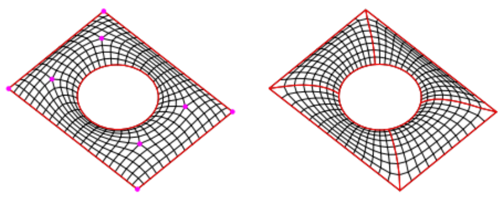


Table 1
Comparison of the structural performance after sizing optimisation of the designs with different topologies in Fig. 23.

Metric	(a)	(b)	(c)	(d)
Number of edges [–]	5915	5049	5743	6049
Beam height [mm]	430	590	220	180
Projected surface weight [kg/m^2]	317	371	210	195
Max. SLS deflection [mm]	138	95	138	138
Max. ULS utilisation [–]	83%	99%	74%	88%
First ULS buckling load factor [–]	23.6	34.6	7.8	4.5



(c) With point features (d) With both features



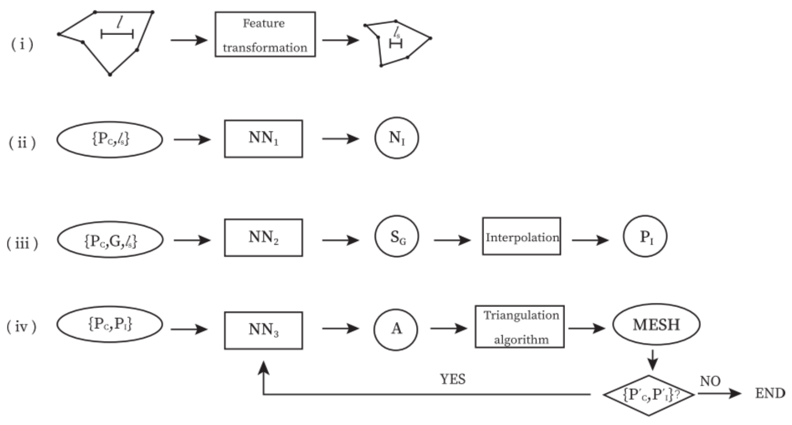
(a) Standard (b) With curve features

2.4.2 How to teach neural networks to mesh: Application on 2-D simplicial contours

Alexis Papagiannopoulos [a, *], Pascal Clausen [b], François Avellan [a]

Mesh generation need to be robust, adaptive to geometry complexities and satisfy the shape requirements (Owen, 1998). The analysis procedure to generate compatible meshes that respect geometric features can take up to 80% of the whole meshing procedure on account of automation absence (Hughes et al., 2005). Therefore there is a high need for efficient computational that require as less explicit treatment as possible. (Papagiannopoulos et al., 2021). Machine learning algorithms rely on data observation and pattern recognition and can solve complex problems.

In his research Papagiannopoulos proposes a framework based on machine learning for generation of 2D meshes. This framework is divided in 4 steps:

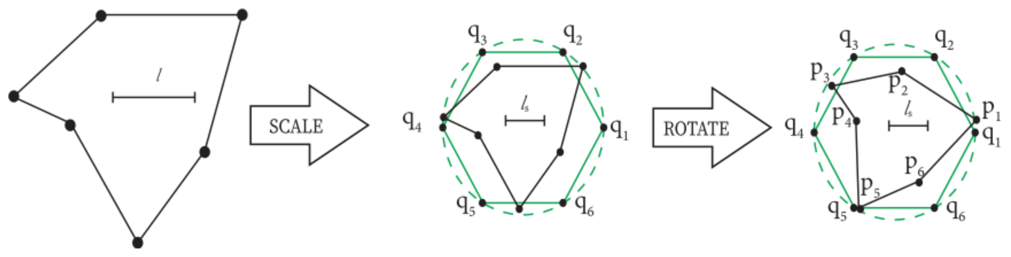


Step 1: Preparation
The contour is scaled and rotated with respect to a regular polygon. The target edge length is also scaled

Step 2: NN1
Input: Coordinates of Contour Vertex & target edge length
Output: Number of vertices that should be inserted inside the cavity of the contour

Step 3: NN2
Input: Coordinates of Contour Vertex, target edge length, a square grid over the Contour
Output: Coordinates of the inner vertices

Step 3: NN3
Input: Coordinates of Contour Vertex, Coordinates of the inner Vertex
Output: Connection table



The research of Papagianopoulos states the importance of mesh modification before the training process. Point coordinates are part of the training dataset. Unprocessed point coordinates, however, do not result in a robust and accurate pattern recognition from the NNs. This is because machine learning methods are usually used for grid-underlying structures, like images. Therefore, Papagianopoulos pre-processes the input data. That includes a step that applies a feature transformation, with scaling and rotating the mesh, to best fit a reference contour circumscribed in a unitary circle. (Papagiannopoulos et al., 2021)

The required scaling and rotation are achieved by applying the Procrustes superimposition[1] on a reference contour. For a contour with NC edges and P*C contour coordinates, a regular polygon is used with NC edges inscribed in a unit circle as a reference. (Papagiannopoulos et al., 2021)

Conclusions: The proposed meshing framework is approximately four times slower than the reference mesher. However, this framework is coded in Python while the reference mesher is written in C++. Taking into account that the speed factor between Python and C++ is that of 5 to 20 and that the current implementation of the algorithm is not optimized for performance Papagianopoulos concluded that the proposed framework attains reasonably good performance.

[1] Procrustes superimposition consists of three steps: translation, scaling, and rotation. As an example, take five configurations of four landmarks each. The contour mesh is translated so that it has the same centroid as the reference polygon. The centered configuration then is scaled to the same centroid size and iteratively rotated until the summed squared distances between the landmarks and their corresponding sample average position is a minimum. (Mitteroecker et al., 2013)

2.4.3 Robust Topology Optimization Using Variational Autoencoders

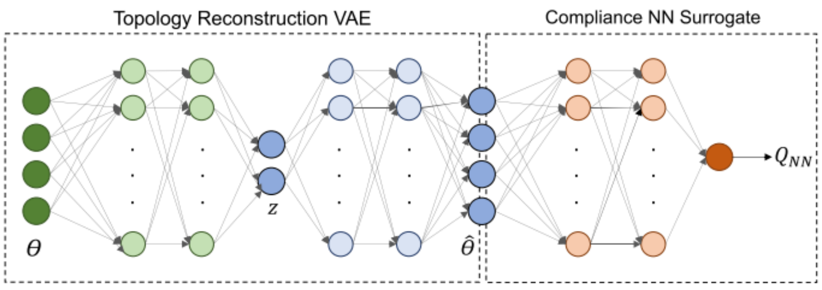
Rini Jasmine Gladstone[1], Mohammad Amin Nabian[1,2], Vahid Keshavarzzadeh[3], and Hadi Meidani*[1]

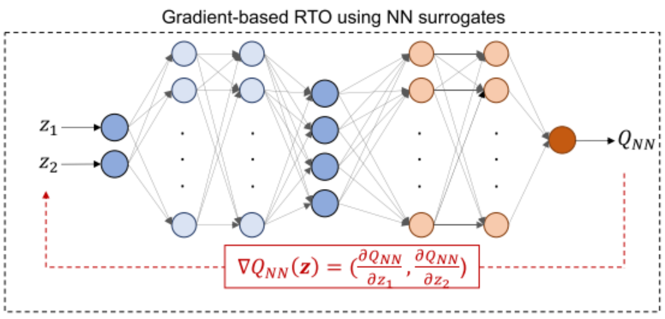
In order to improve the computational time for a compliance minimization problem, Gladstone (Gladstone et al., 2021) used a Variational Autoencoder (VAE) to transform the high dimensional design space into a low dimensional one, thus making the design space exploration more efficient. In her research finite element solvers were replaced by a neural network surrogate that predicts the probabilistic objective function.

Step1:
Parameterization of the high dimensional geometry of the design candidates using a low dimensional representation obtained by VAEs.

Step2:
Replacement of the finite element solver with feed forward fully connected compliance neural network surrogate to accelerate the cost (robust compliance) evaluation. The input layer has number of nodes equal to the dimension of the training image. Output layer is a single node which gives predicted robust compliance, QNN(θ).

Step3:
A gradient descent algorithm is used to find the optimal design on the low dimensional representation, minimizing the robust compliance.



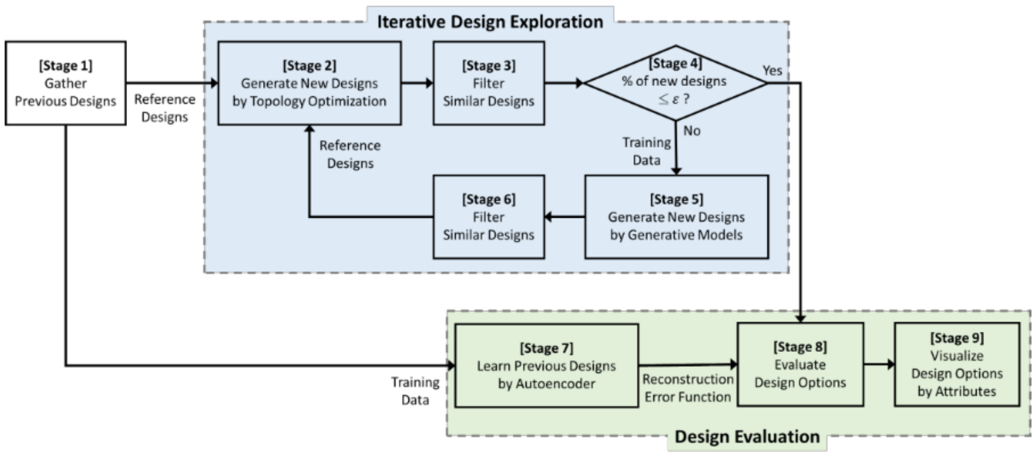

Gradient-based RTO using NN surrogates
z_1
z_2
Q_{NN}
$\nabla Q_{NN}(\mathbf{z}) = (\frac{\partial Q_{NN}}{\partial z_1}, \frac{\partial Q_{NN}}{\partial z_2})$

Conclusions: In this paper VAEs are used successfully to turn the high dimensional optimization problem into a low dimensional one. During the production of the training dataset a topology optimization algorithm based on finite element method was used. The proposed framework produced robust optimal designs better than the finite element method.

2.4.4 Deep Generative Design: Integration of Topology Optimization and Generative Models

Sangeun Oh[1,†], Yongsu Jung[2,†], Seongsin Kim[1], Ikjin Lee[2,*], Namwoo Kang[1,*]

In his study Oh (Oh et al., 2019) presented an artificial intelligent (AI)-based deep generative design framework that is capable of generating numerous design options which are not only aesthetic but also optimized for engineering performance.


Iterative Design Exploration
[Stage 1] Gather Previous Designs
Reference Designs
[Stage 2] Generate New Designs by Topology Optimization
[Stage 3] Filter Similar Designs
[Stage 4] % of new designs ≤ ε ?
Yes
No
Training Data
Reference Designs
[Stage 6] Filter Similar Designs
[Stage 5] Generate New Designs by Generative Models
[Stage 7] Learn Previous Designs by Autoencoder
Training Data
Reconstruction Error Function
[Stage 8] Evaluate Design Options
[Stage 9] Visualize Design Options by Attributes
Design Evaluation

The workflow consisted of the following steps:

Stage 1: The earlier designs in the market and the industry are collected as reference designs

Stage 2. The designs are topologically optimized. The optimization process is multi-objective and the performance indicators are (1) compliance minimization and (2) difference (i.e., pixel-wise L1 distance) minimization from the reference design.

Stage 3: Similar designs gathered from topology optimization are filtered out by a similarity criterion (also stage 6)

Stage 4: The ratio of the number of new designs in the current iteration to the number of total designs in the previous iteration is calculated. If it is smaller than the user-specified threshold, then exit the iterative design exploration and jump to Stage 8. Otherwise. proceed to Stage 5.

Stage 5: New designs are created by generative models after learning aggregated designs in the current iteration, and they are used as reference designs in Stage 2 after filtering out similar designs in Stage 6.

Stage 7: Involves the building of a loss function (i.e., reconstruction error function) employing autoencoder trained by previous designs of Stage 1. This step checks for design novelty.

Stage 8: Design options obtained from iterative design exploration have to be evaluated on the basis of various design attributes that are essential to the designers.

Conclusions: Many designs starting from a small number of designs was generated. The proposed framework offered diverse designs in comparison with the conventional generative design. Moreover, the robustness on quality of designs is improved.

3. DATASET GENERATION

3.1 Constructing a Mesh

A mesh is a representation of a larger geometric domain by smaller discrete cells.

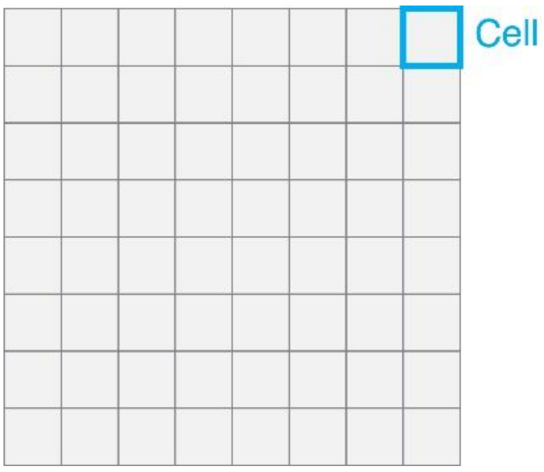


Figure 3.1. A mesh and its cells.

A Mesh is constructed after specifying the two following lists:

- A lists with the vertex coordinates
- A list containing the connection of the vertices to form faces.

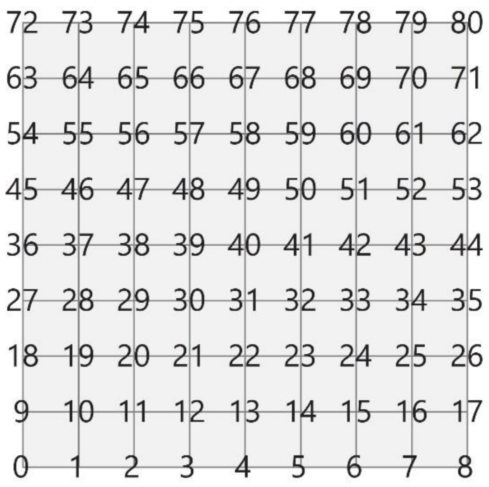


Figure 3.2. Rectangular mesh and the indexes of its vertices.

```
Vertices
0. [0, 0, 0]
1. [1.875, 0, 0]
2. [3.75, 0, 0]
3. [5.625, 0, 0]
…
78. [11.25, 15, 0]
79. [13.125, 15, 0]
80. [15, 15, 0]
```

```
Faces
0. [0,1,10,9]
1. [1,2,11,10]
2. [2,3,12,11]
3. [3,4,13,12]
…
61. [68,69,78,77]
62. [69,70,79,78]
63. [70,71,80,79]
```

3.2 Creating subdivision patterns

In his research R. Oval describes the strip method for changing a mesh’s subdivision pattern. The ‘adding strip’ method is described in the images below, where strips are inserted along edges

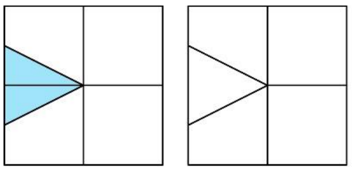


Figure 3.3 Inserting a stripe.

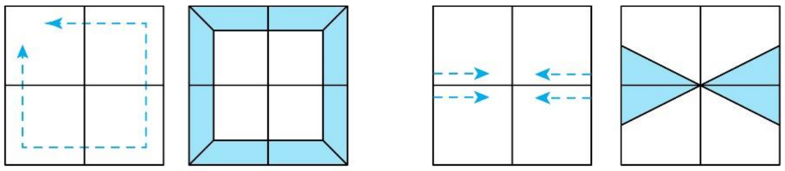


Figure 3.4 & 3.5 Changing the subdivision pattern with the strip method.

To create the dataset for this thesis another method will be followed that can result in the same patterns by merging points:

- First a set of points is chosen. ○
- Then they are either merged to their centre point or if one of them is an extreme point they are merged at this one. ●

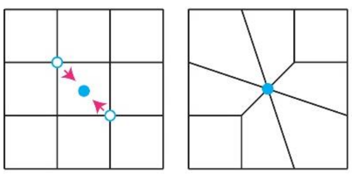


Figure 3.6 Joining two points in their centre.

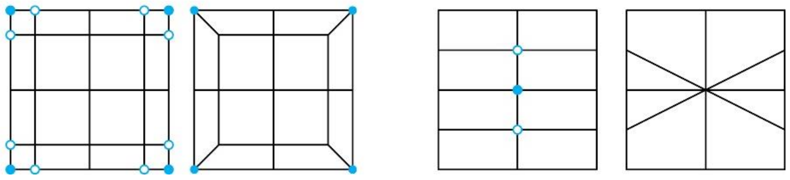


Figure 3.7 & 3.8 Creating the new subdivision pattern through joining points.

3.3 Method explanation

Every vertex v of the mesh has a set of neighbouring vertices that are connected to v by an edge. We can chose a neighbour for each vertex by selecting the index from the list of its neighbours. Various mesh tessellations can occur when merging random vertices with random neighbours.

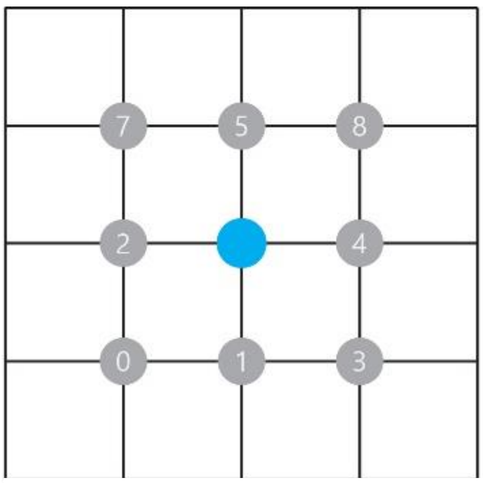


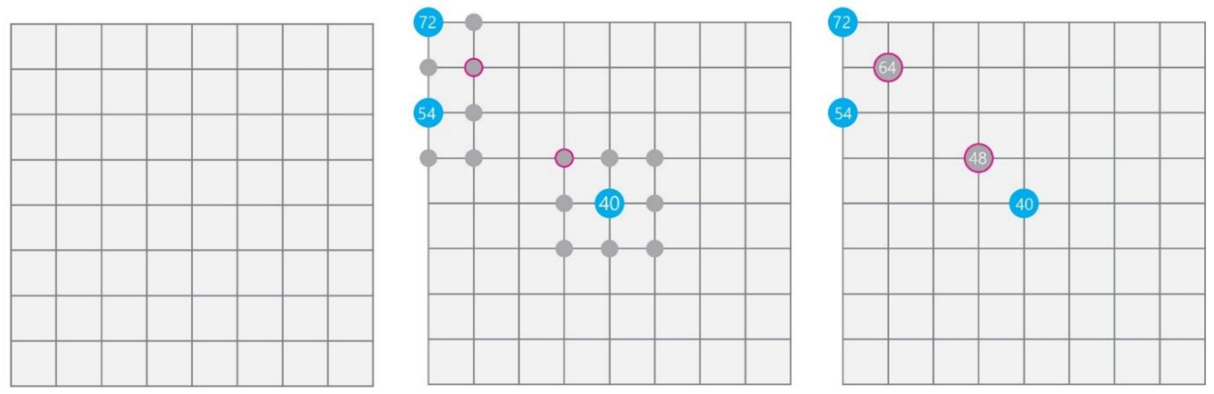
Figure 3.9 A vertex with its neighboors.

The dataset is created using python and the `COMPAS` framework. The python code can be found at the Appendix.

The topology exploration is described in the following steps:

Step 1

First an initial mesh is specified. Then, random vertices are selected. Another set of vertices is chosen from the lists of their neighbours, by choosing random indexes as seen at the image in the middle. These random vertices and their selected neighbours form sub-lists are all placed at a list with the `vertices_to_merge`, as seen at the image on the right.



```
Initial Mesh
```

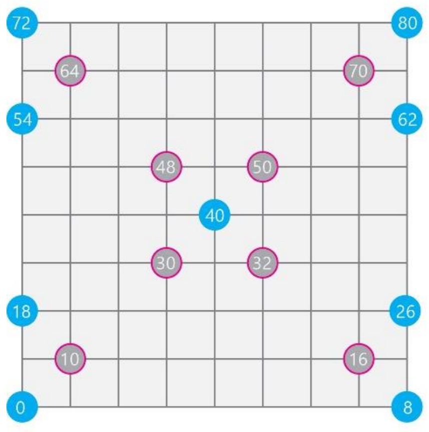
```
Random_vertices=[40,54,72]
Random_neighboors=[0,3,1]
```

```
Vertices_to_merge=[72,64],
[54,64],[40,48]
```

Figure 3.10 Creating the vertices_to_merge list.

Step 2

In this step the symmetrical vertices of the `vertices_to_merge` are added in a new list. If a vertex exists in more than one sub-list then these sub-lists are merged.

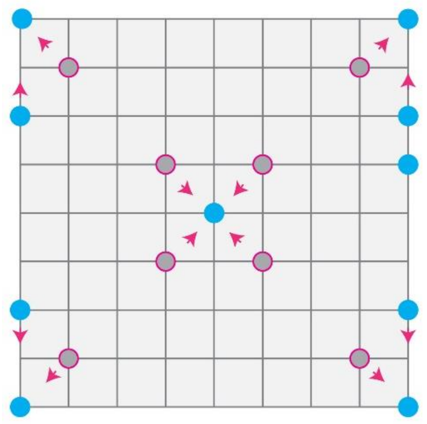


```
vertices_to_merge=[[64, 72], [70, 80],
[10, 0], [16, 8], [64, 54], [70, 62],
[10, 18], [16, 26], [30, 40], [32,
40], [48, 40], [50, 40]]

Vertices_to_merge_new=[[64, 72, 54],
[80, 62, 70], [0, 10, 18], [16, 8,
26],
```

Figure 3.11 Selecting symmetrical vertices.

Step 3

A new list with the merged vertices is created that includes the coordinates of the points where the `vertices_to_merge` are going to be joined. These points are usually the mean point or centre point of the vertices. If some vertices on the sub-lists are extreme vertices, then it is their centre that is needed for the new list. If one vertex of the sub-lists is a corner point, then this point is the one that is placed on the `merged_vertices` list.

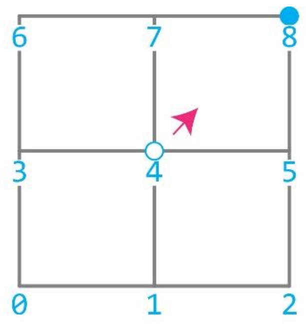


vertices_to_merge		Merged_vertices
[64, 72, 54]	→	[0.0, 15.0, 0.0]
[80, 62, 70]	→	[15.0, 15.0, 0.0]
[0, 10, 18]	→	[0.0, 0.0, 0.0]
[16, 8, 26]	→	[15.0, 0.0, 0.0]
[32, 40, 48, 50, 30]	→	[7.5, 7.5, 0.0]

Figure 3.12 Creating the merged_vertices list.

Step 4

The merged vertices are going to replace the initial vertices. A simpler mesh is used to explain the method. In the image below the goal is to merge the vertex 4(v4) with vertex 8(v8).



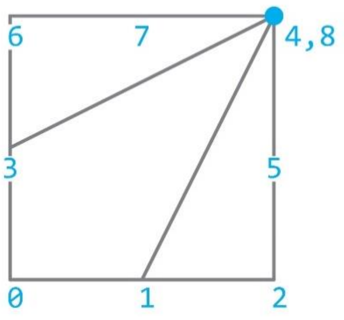
```
Indices of points to merge: [4,8]
Merged Point: [3.75, 3.75, 0]
```

```
Vertices

0. [0, 0, 0]
1. [1.875, 0, 0]
2. [3.75, 0, 0]
3. [0, 1.875, 0]
4. [1.875, 1.875, 0]
5. [3.75, 1.875, 0]
6. [0, 3.75, 0]
7. [1.875, 3.75, 0]
8. [3.75, 3.75, 0]
```

```
Faces

0. [0;1;4;3]
1. [1;2;5;4]
2. [3;4;7;6]
3. [4;5;8;7]
```

Figure 3.13 The starting Mesh with its list of vertices and faces.

As the vertex 8 is an extreme point, then it is the merged vertex as well. In the list of vertices the v8 replaces the v4. After this step a new topology is created.



```
Vertices

0. [0, 0, 0]
1. [1.875, 0, 0]
2. [3.75, 0, 0]
3. [0, 1.875, 0]
4. [3.75, 3.75, 0]
5. [3.75, 1.875, 0]
6. [0, 3.75, 0]
7. [1.875, 3.75, 0]
8. [3.75, 3.75, 0]
```

```
Faces

0. [0,1,4,3]
1. [1,2,5,4]
2. [3,4,7,6]
3. [4,5,8,7]
```

Figure 3.14 The generated mesh with its lists of vertices and faces.

However double and unused vertices occur (vertices 4 and 8)as well as zero-area faces (face 3). Therefore, removing unnecessary vertices and faces is needed.

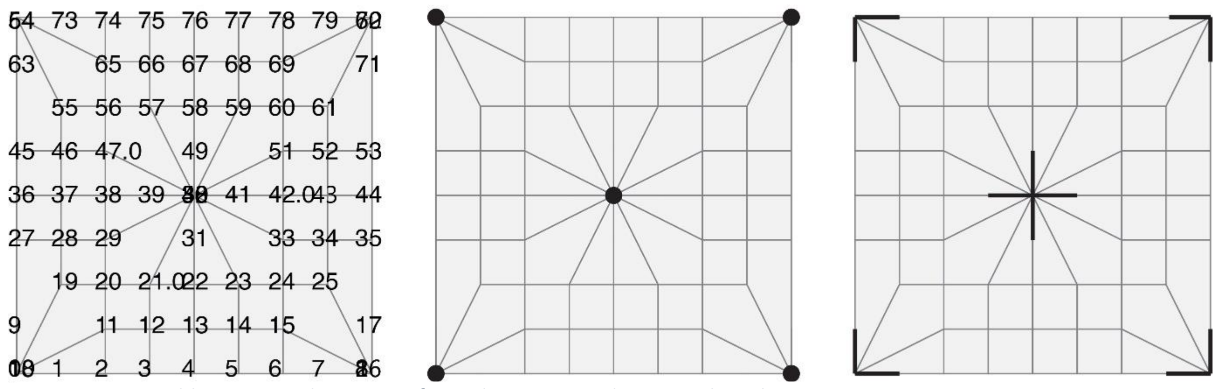


Figure 3.15 Double points and zero area faces that occur at the original mesh.

Step 5

`Compas` can create meshes from either lists or dictionaries as inputs. After clearing the unnecessary the vertices and faces, `Compas` creates new meshes that are dictionaries whose key numberst are not in order. This would be ok, however the relaxation algorithm didn’t seem to work in this way so the order needed to be fixed.

```
Vertices_dict = {
 0: [1.875, 0, 0],
 1: [3.75, 0, 0],
 2: [0, 1.875, 0],
 3: [3.75, 1.875, 0],
 5: [3.75, 1.875, 0],
 6: [0, 3.75, 0],
 7: [1.875, 3.75, 0],
 8: [3.75, 3.75, 0],
}
```

```
Faces_dict = {
 0: [0,1,4,3],
 1: [1,2,5,4],
 2: [3,4,7,6]
}A
```

Figure 3.16 The problem with the order of keys at the dictionaries that describe the simpler mesh .

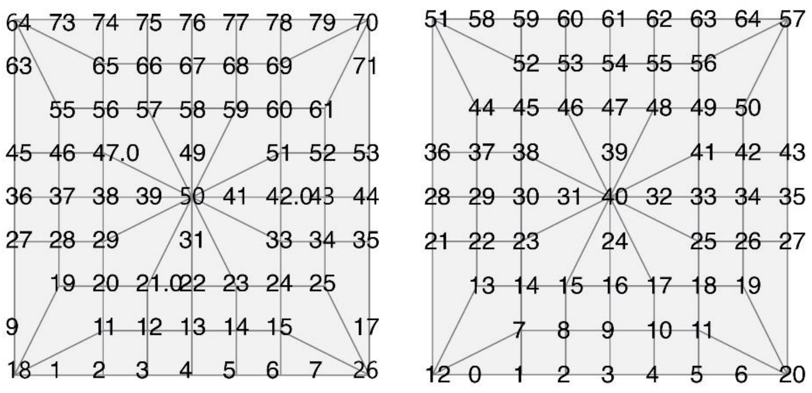


Figure 3.17 Fixing the key numbers of the vertices at the original mesh.

Step 6

Finally the mesh is relaxed. The boundary points are the anchor points and the force density algorithm is used with a load of 2kN/m^3 prescribed in the edges

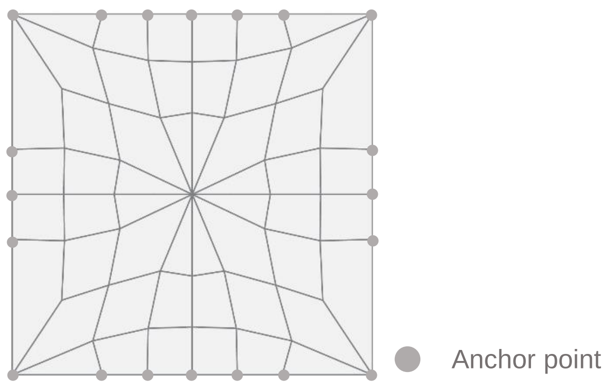


Figure 3.18 The relaxed mesh.

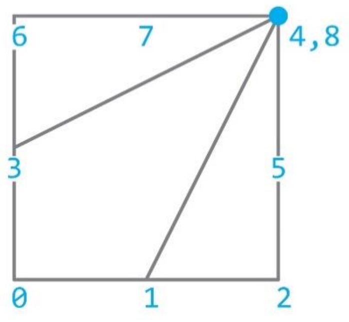
3.4 Dataset pre-processing for training AI models

In order to train the model, tensors that will occur from arrays with the same shape need to be used. As the generated meshes have different number of vertices and faces, their final lists cannot be used as they are in the dataset. The process of making appropriate arrays is described below. Two options were tested. The python code can be found at the Appendix.

All the training data are exported after generating the new meshes and before relaxation.

Option 1: Using the structure of vertices as training data

The first attempt is to train the model using the list of vertices without removing double occurring vertices as they occur in step 4. The list of faces remains the same



```
Vertices

0. [0, 0, 0]
1. [1.875, 0, 0]
2. [3.75, 0, 0]
3. [0, 1.875, 0]
4. [3.75, 3.75, 0]
5. [3.75, 1.875, 0]
6. [0, 3.75, 0]
7. [1.875, 3.75, 0]
8. [3.75, 3.75, 0]
```

```
Faces

0. [0;1;4;3]
1. [1;2;5;4]
2. [3;4;7;6]
3. [4;5;8;7]
```

Figure 3.19 The simpler mesh along with its lists that will be used to create the training data.

The training tensors occur from arrays after replacing the indices of the vertices at the `faces` list with the coordinates of the `Vertices`.

.

```
array([[0, 0, 0],[1.875, 0, 0];[3.75, 3.75, 0], [0, 1.875, 0],
[1.875, 0, 0], [3.75, 0, 0], [3.75, 1.875, 0], [3.75, 3.75, 0]]
[0, 1.875, 0], [3.75, 3.75, 0], [1.875, 3.75, 0], [0, 3.75, 0]]
[3.75, 3.75, 0]; [3.75, 1.875, 0], [3.75, 3.75, 0], [1.875, 3.75, 0]])
```

Figure 3.20 The final array that will be used for trainin.g

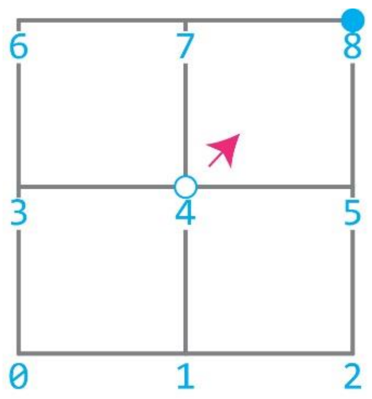
Exporting the training data as they occur at step 4 means that the generative model will be trained with unrelaxed meshes.

Option 2: Using an adjacency matrix as training data

The second option requires pre-processing the generated meshes. Meshes can be described as graph data using adjacency matrices.

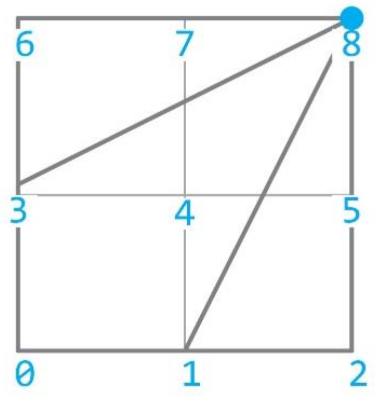
As various new points are creating after merging random initial points, a denser mesh, that is occurring after subdividing the initial one by two, is used.

Multiple graphs are created but they all share vertices with the same coordinates. This means that the training data describe an unrelaxed mesh.



	0	1	2	3	4	5	6	7	8	9
0	0	1	0	1	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0
2	0	1	0	0	0	1	0	0	0	0
3	1	1	0	0	1	0	1	0	0	0
4	0	1	0	1	0	0	0	1	0	0
5	0	0	1	0	1	0	0	0	1	0
6	0	0	0	1	0	0	0	1	0	0
7	0	0	0	0	1	0	1	0	1	0
8	0	0	0	0	0	1	0	1	0	0
9	0	0	0	0	0	0	0	0	0	0

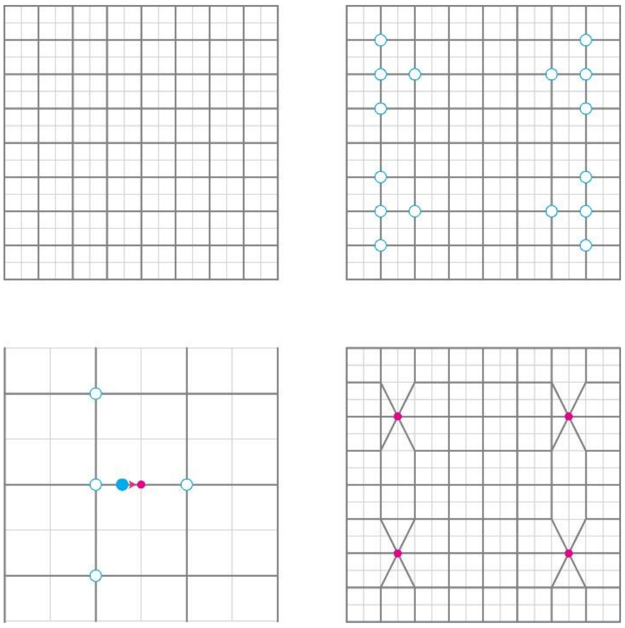
Figure 3.21 A starting mesh with its adjacency matrix.



	0	1	2	3	4	5	6	7	8	9
0	0	1	0	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	1	0
2	0	1	0	0	0	1	0	0	0	0
3	0	1	0	0	1	0	1	0	1	0
4	0	1	0	1	0	0	0	0	0	0
5	0	0	1	0	1	0	0	0	1	0
6	0	0	0	1	0	0	0	1	0	0
7	0	0	0	0	0	0	1	0	1	0
8	0	1	0	1	0	1	0	1	0	0
9	0	0	0	0	0	0	0	0	0	0

Figure 3.22 A final mesh with its adjacency matrix.

A problem that may occur with this strategy is that the merged points may not coincide with the vertices of the denser mesh. This can be prevented by taking by taking into account only unique coordinates when calculating the average point.



```
Vertices
0. [1.875, 1.875, 0]
1. [1.875, 3.75, 0]
2. [1.875, 5.625, 0]
3. [3.75, 3.75, 0]
```

```
x Coordinates
0. 1.875
1. 1.875
2. 1.875
3. 3.75
```

Take only unique values

```
Average x Coordinate
(1.875+3.75)/2
```

Figure 3.23 Vertices whose mean point doesn't snap at the vertices of the base mesh and the trick to fix this issue.

Option 3: Using the flatten array that occurs from the adjacency matrix

The adjacency matrix that describes the connections of nodes is always symmetrical to its diagonal axis. As described in Chapter 4 the tensors are flattened before passing inside fully connected layers.

The size of the training sample can be minimized by removing the double sequences that occur during the dataset pre-processing step.

The highlighted part of the adjacency matrix (that describes a simpler mesh) bellow represents the flattened information that is kept.

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	1	0
2	0	1	0	0	0	1	0	0	0	0
3	0	1	0	0	1	0	1	0	1	0
4	0	1	0	1	0	0	0	0	0	0
5	0	0	1	0	1	0	0	0	1	0
6	0	0	0	1	0	0	0	1	0	0
7	0	0	0	0	0	0	1	0	1	0
8	0	1	0	1	0	1	0	1	0	0
9	0	0	0	0	0	0	0	0	0	0

$\longrightarrow$

```
[
0,
1,0,
0,1,0,
0,1,0,0,
0,1,0,1,0,
0,0,1,0,1,0,
0,0,0,1,0,0,0,
0,0,0,0,0,0,1,0
0,1,0,1,0,1,0,1,0
0,0,0,0,0,0,0,0,0,0
]
```

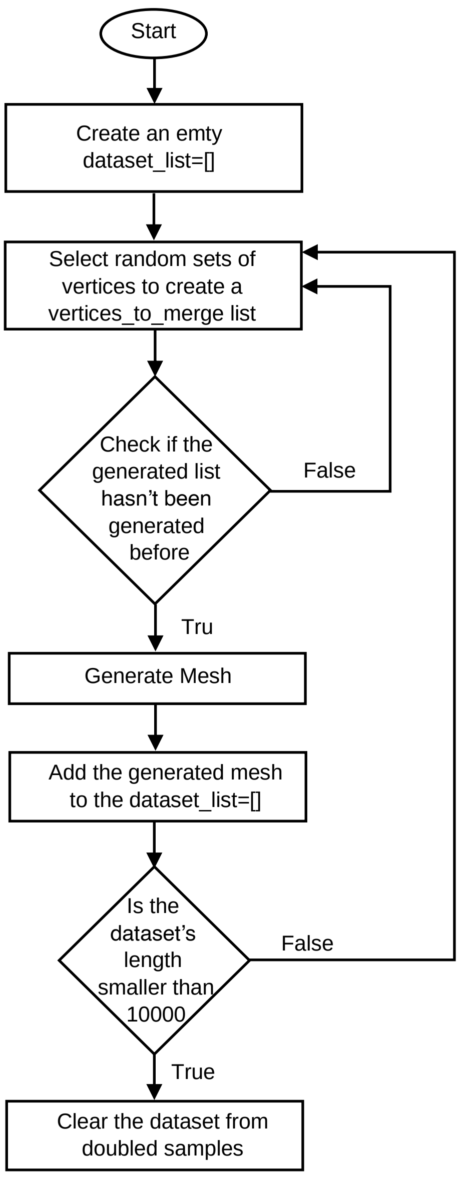
Figure 3.24 The adjacency matrix to a flattened array.

3.5 Dataset Generation

The following process describes the generation of two databases: one for creating the data appropriate for training the generative model and another for creating mesh data that will be simulated with FEM software to create the labels that the surrogate models will learn to predict.

These are the following steps:

- First an empty `dataset_list` is created
- Then random sets of vertices are selected so a list with “`vertices_to_merge`” is created. The combination of the vertices and the length of the list is also random. Since the symmetrical vertices are going to be added, the random choice is limited to one quarter of the mesh.
- If the generation of the “`vertices_to_merge`” hasn’t been generated before, the algorithm moves to the next step. If not another random set is generated.
- A new mesh is created according to the list of “`vertices_to_merge`” and it is added to the `dataset_list`
- If the desired `dataset_list` length is reached (10000 in this thesis) the algorithm is moved to the next step, if not the second step is repeated
- The `dataset_list` is cleared from doubled samples



Start
Create an emty dataset_list=[]
Select random sets of vertices to create a vertices_to_merge list
Check if the generated list hasn’t been generated before
False
Tru
Generate Mesh
Add the generated mesh to the dataset_list=[]
Is the dataset’s length smaller than 10000
False
True
Clear the dataset from doubled samples

Figure 3.25 Creating the dataset.

- The `dataset_list` is converted to data appropriate for training a machine learning model. For this thesis 5890 training samples were generated.
- The new meshes are relaxed and data appropriate for FEM simulation are exported.
- The FEM simulation creates the labels that will be used in the surrogate model

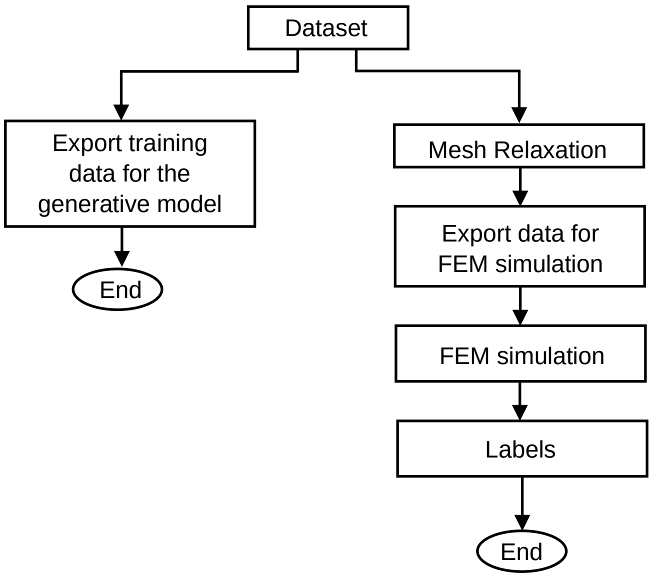

Dataset
Export training data for the generative model
End
Mesh Relaxation
Export data for FEM simulation
FEM simulation
Labels
End

Figure 3.26 Exporting the trainning dataset and data for FEM simulation.

3.6 FEM Simulation

All the data including mesh information are exported in text documents. Then they are imported inside the `Grasshopper` environment in `Rhinoceros`. For FEM testing `Karamba3D` is used.

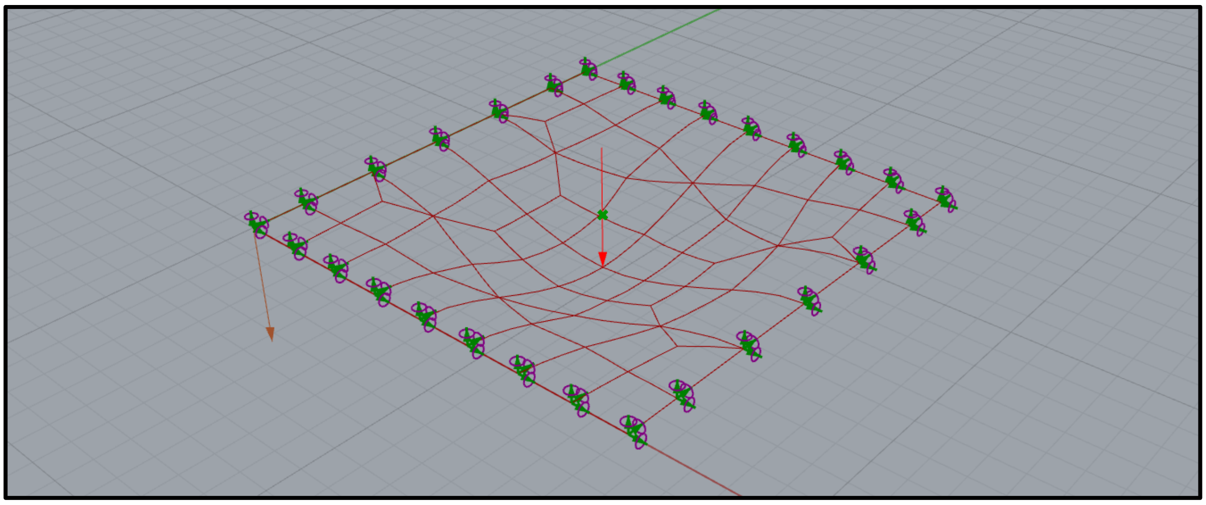


Figure 3.27 Assigning loads and supports to the mesh's edges.

The mesh’s edges are tested as steel beams network(Steel S235) with a IPE80 cross section.
A point load of 1 kN applied at the centre of the network. All the boundary points are used as fixed supports.

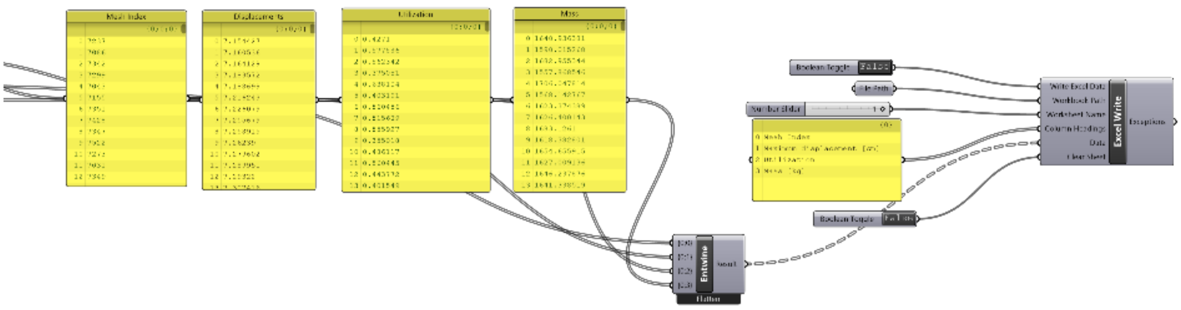


Figure 3.28 Mass, Displacement and Utilization as performance indicators.

After testing the following performance indicators are extracted and stored in a csv file:

1. The **Maximum Displacement** in cm.
2. The **Maximum Utilization** (ratio between the tensile or compressive strength and the maximum allowable stress)
3. The **Mass** of the structure in kg.

All performance indicators are normalized to fit in the range between 0 and 1. Then a performance value is assigned to each index using the following formula:

$$\mathbf{Performance} \ =$$

$$\mathbf{0.4} \ \times \boldsymbol{Normalized\ Displacement} + \mathbf{0,4} \times \boldsymbol{Normalized}\ \mathbf{Utilization} + \mathbf{0,2} \times \boldsymbol{Normalized}\ \mathbf{Mass}$$

The 10 best performed mesh tessellations are excluded from the training dataset (the highlited ones) to see if Artificial Intelligence can solutions better than them.

Mesh Index	Maximum displacement [cm]	Utilization	Mass [kg]	Norm displacement	Norm Utilization	Norm Mass	Performance Score	Norm Performance
1592	7.222685	0.31688	1467.04	0.198500114	0.038938908	0.06568577	0.108112762	0
916	7.306213	0.331459	1525.86	0.216281778	0.056459133	0.23781521	0.156659406	0.085279428
2871	7.316427	0.383731	1495.76	0.218456162	0.119276694	0.14971763	0.165036668	0.099995339
3178	7.336311	0.334702	1534.57	0.222689121	0.060356389	0.26328098	0.165874401	0.101466942
585	7.198498	0.35392	1539.1	0.19335112	0.083451505	0.27654819	0.166030688	0.101741484
2448	8.088958	0.284613	1479.04	0.382914632	0.000162235	0.10080725	0.173392197	0.114673073
468	7.40201	0.343348	1535.19	0.2366753	0.070746668	0.26510758	0.175990302	0.119237033
1093	7.193045	0.39318	1525.4	0.192190271	0.130631973	0.23645109	0.176419116	0.11999031
3374	7.112629	0.357602	1570.46	0.175071097	0.087876326	0.36831927	0.178842824	0.124247914
2286	7.576214	0.379444	1487	0.273760315	0.114124818	0.12409274	0.179972602	0.126232537
3487	7.139086	0.4255	1514.93	0.180703335	0.169472338	0.20581014	0.181232297	0.12844538
3659	7.632908	0.312611	1537.29	0.285829486	0.033808663	0.27125253	0.182105766	0.129979758
3143	7.720711	0.370308	1479.8	0.304521224	0.103145685	0.10302932	0.183672627	0.132732184
370	7.183572	0.375051	1557.87	0.190173634	0.108845557	0.33147061	0.185901798	0.136648056
3401	7.33804	0.386916	1526.39	0.223057195	0.123104249	0.23936516	0.18633761	0.137413625
131	7.32969	0.419356	1504.03	0.221279625	0.162088823	0.17393769	0.188134916	0.140570862

Table 3.1 The table with all the performance indicators, their normalized values and the final performance score

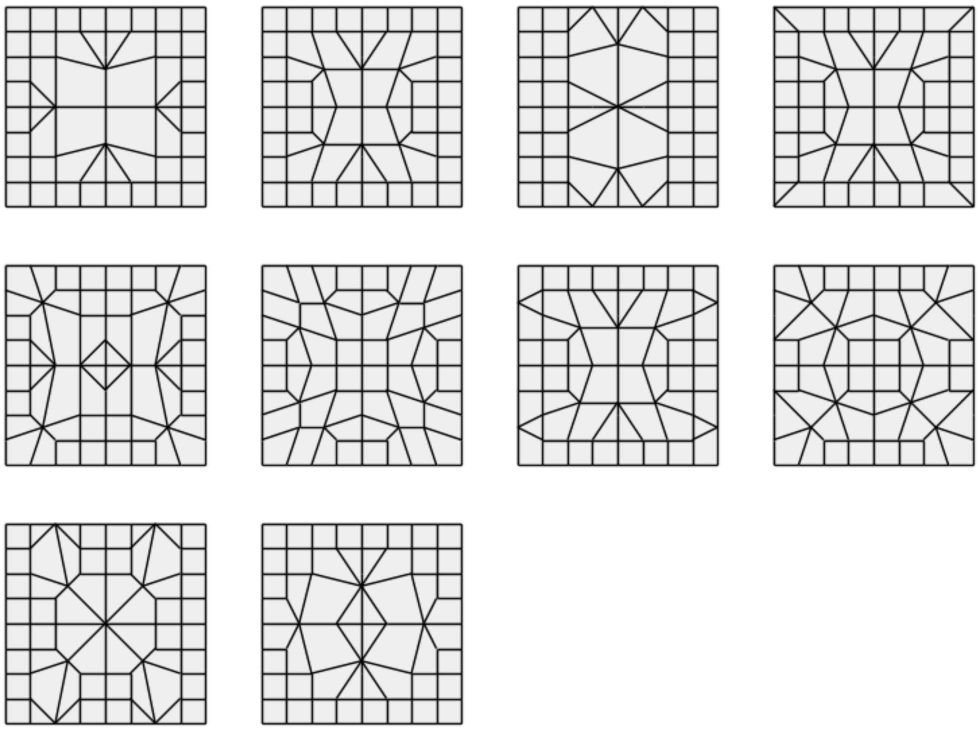


Figure 3.29 The test best performed meshes.

3.7 Dataset Augmentation

The generated samples are symmetrical on either one or two axes. To augment the dataset the samples that are symmetrical on just one axis are rotated 90 degrees. To do this, the dataset is first purged of samples that have the same performance-score (flipped patterns have the same score). The next step is to rotate all the samples 90 degrees. Finally a last step is performed were the dataset is cleared from doubled meshes (symmetrical meshes on both axes occur two times after the rotation step). Overall 7.338 samples were created.

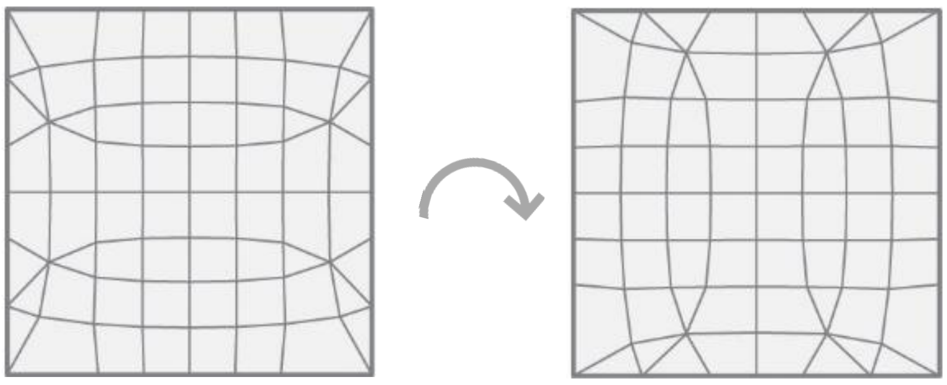


Figure 3.30 Rotating the meshes 90 degrees.

4. AI WORKFLOW

4.1 The Variational Autoencoder

4.1.1 Architecture

The Variational Autoencoder is based on F. Chollet's VAE model (Chollet, 2020) with changes on hyperparameters (layers, the batch size, epochs and learning rate). `Keras` and `Tensorflow` python libraries were used for implementing the VAE.

The VAE is a deep learning model that consists of an encoder and a decoder with hidden fully connected layers (dense layers). The input data pass through an encoder that is able to shrink the data to fit in a latent space. A final layer called sampling layer in the encoder is able to perform a reparameterization trick to make sure that the latent space is regular. The decoder then attempts to reconstruct the input data. The model is trained by dividing the dataset in batches. After passing the entire dataset through the neural network one epoch is completed. The neural network measures the error of reconstruction and updates the weights at its nodes.

A workflow of the whole network is presented below.

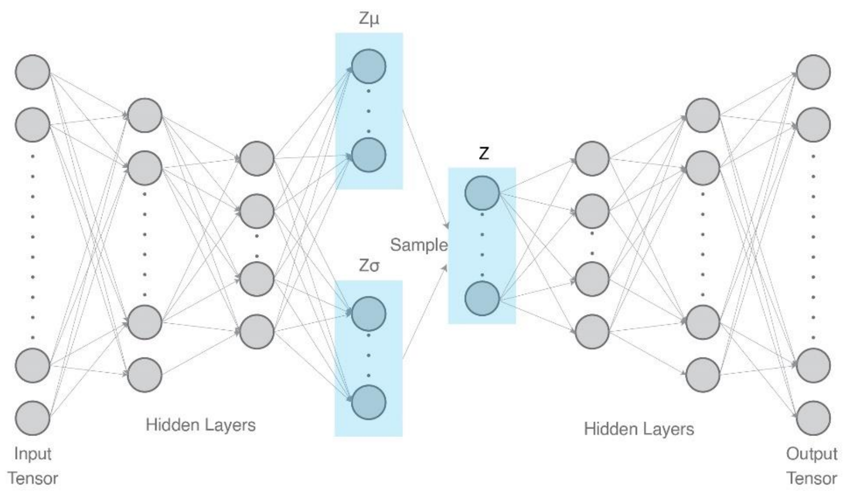


Figure 4.1 The architecture of a VAE

4.1.2 Data pre-processing

The input data have to be normalized between 0 and 1 to enter the neural network. Normalizing the data generally speeds up learning and leads to faster convergence.

In the first option the training data that include the mesh's coordinates have to be divided with the dimensions of the mesh (in this thesis it is 15). In the second option and third option where adjacency matrices are used, no normalization is required as the input are arrays with only 0 and 1values.

Data that enter the NN also are arrays that include float numbers.

The general architecture of the VAE, also explained in the first chapter is demonstrated below. It is a autoencoder with a slight modification that ensures the regularization of the latent space.

4.1.3 VAE - loss function

The loss function used for the training VAEs is the sum of two losses. The first one measures the error between the original input and its reconstruction after exit the NN. The second one is the Kullback-Leibler divergence (kl divergence) loss. The purpose of the kl divergence in the loss function is to make the distribution of the encoder output as close as possible to a standard multivariate normal distribution.(Lunot, 2019)

$$\text{Total Loss} = \text{Reconstruction Loss} + \text{kl coefficient} * \text{kl loss}$$

A demonstration of the effect of kl divergence in the latent space is demonstrated in the image below on a latent space with two vectors z[0] and z[1] and the outputs are clustered depending on their performance. In this thesis a kl coefficient of 1 was used.

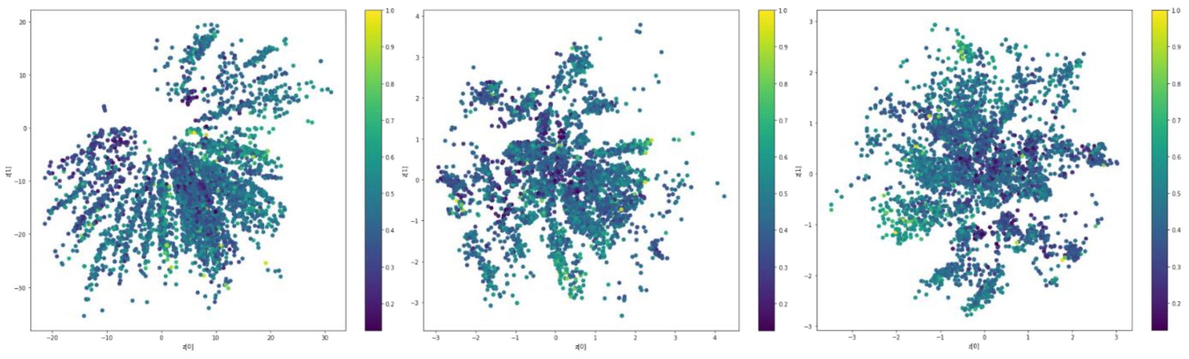


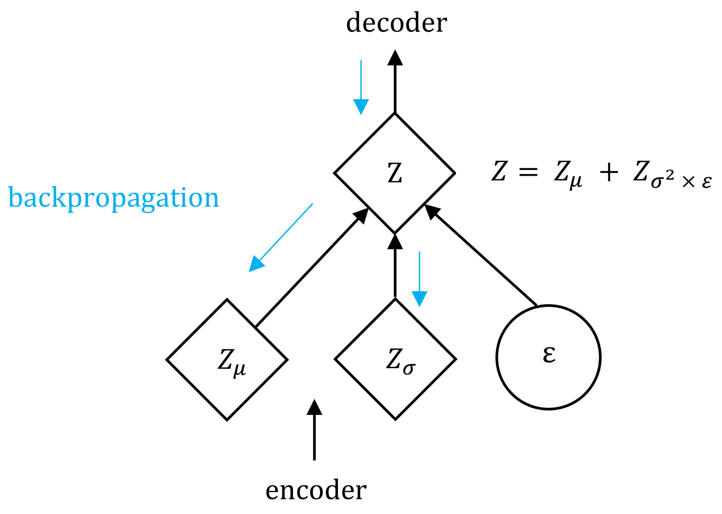
Figure 4.2 From left to the right: kl coefficient:0, kl coefficient:1, kl coefficient:2. The encoder and decoder have to dense layers with 100 nodes each. The training used 100 epochs and a batch size 64 on an input tensor [81,81] (data input are adjacency matrices with one quarter of the information)

During the `fit()` process we also separate a a portion of the training data to evaluate the performance of your model on a validation dataset. The evaluation of the training data is called `train step` and that of the validation dataset is called `test_step`. We can customize what is happening during those steps by overriding the We can customize the training loss by overriding the `VAE.train_step()` and the `VAE.test_step()`, while retaining built-in infrastructure features. The code for these steps can be found at the Appendix.

4.1.4 Sampling Layer

In autoencoders input data are compressed through the encoder without taking into account the regularity of the latent space. This is ok when the goal is to achieve compression of large databases, but in generative models we need to be able sample vectors from a regular latent space to generate valid outputs. Therefore VAEs introduce a sampling layer in the network that ensures that the input data get mapped to latent variables with a **normal distribution.** This distribution is parameterized by a **mean** (μ) and a **variance** (σ) which are the learnable parameters of the network.

VAEs also introduce **stochasticity** in the network. Through backpropagation the neural network learns a normal distribution that needs also to be probabilistic. This is achieved by adding a random noise to the vector by multiplying the square of the variance with a random variable ε. This variable has also a low value so as to ensure that the result does not deviate a lot from the true distribution. Adding this variable is what allows the **reparameterization trick**, demonstrated in the following diagrams. On the left an input is sampled as a latent vector Z from a normal distribution. This does not allow to compute the gradients to approximate the latent space, hence we cannot backpropagate back to the NN. This is fixed by multiplying the variance square by inserting the variable ε to calculating the vector Z. (Doersch, 2016)



Z_{μ}: *A vector with a mean of* μ

Z_{σ}: *A vector with a variance of* σ

ε : *A a random variable sampled from a standard normal distribution*

The code for creating the Sampling layer can be found the Appendix

4.1.5 Encoder- Decoder

Option 1 : Using the structure of vertices as training data

The first attempt was to train the VAE using as input the coordinates of the vertices as arrays. The input shape of the tensors was `[81,3]`. The encoder and decoder used one dense layer each with 81 nodes. The latent space had 9 nodes. Batch size was 64 and the optimizer was Adam with learning rate 0.001.

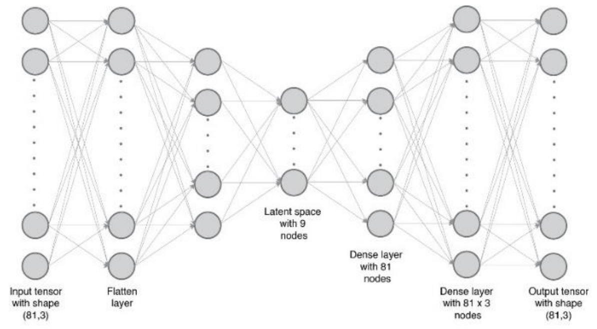

Input tensor with shape (81,3)
Flatten layer
Latent space with 9 nodes
Dense layer with 81 nodes
Dense layer with 81 x 3 nodes
Output tensor with shape (81,3)

Figure 4.3 The architecture of the first's option VAE

Other combinations of hyperparameters were tested, but the model was not able to decode correct outputs in none of the cases. An original input along with its decoded output is demonstrated below, along with the loss chart.

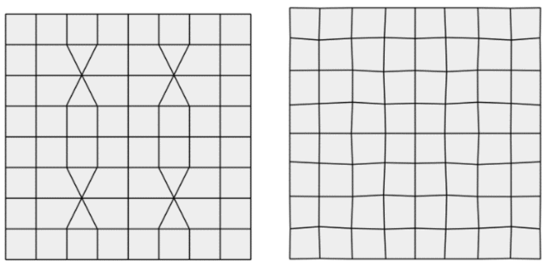


Figure 4.4 From left to the right: The original input and the decoded output

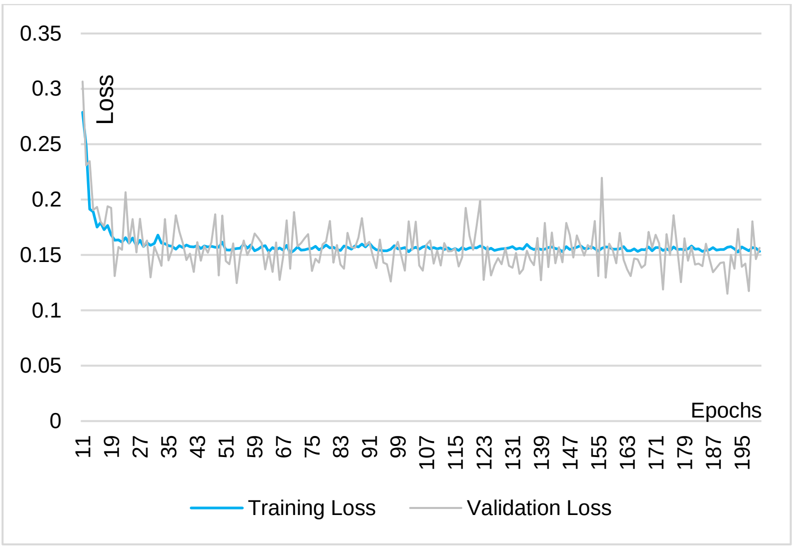


Chart 4.1 Training loss and validation loss after 200 epochs for option 1

Option 2 : Using adjacency matrices as training data

The second option of training the NN with adjacency matrices was more successful. The encoder and the decoder consisted of feel forward fully connected layers (dense layers). Various numbers of layers and nodes were tested. For training TU Delft's supercomputer Delft Blue was used.

Below different variations are demonstrated. All layers in both encoder and decoder are fully connected. ReLU was used as an activation function in all layers apart from the last one in the decoder, which used sigmoid.

The last two variations consider the shape of input which is (289,289). The layers that are used in those variations have a number of nodes that consists of multiples and dividers of the number 289. The architecture that was converged better was the **Revision 4**.

Schematic diagrams with the tested architectures of the encoder and the decoder are demonstrated below.

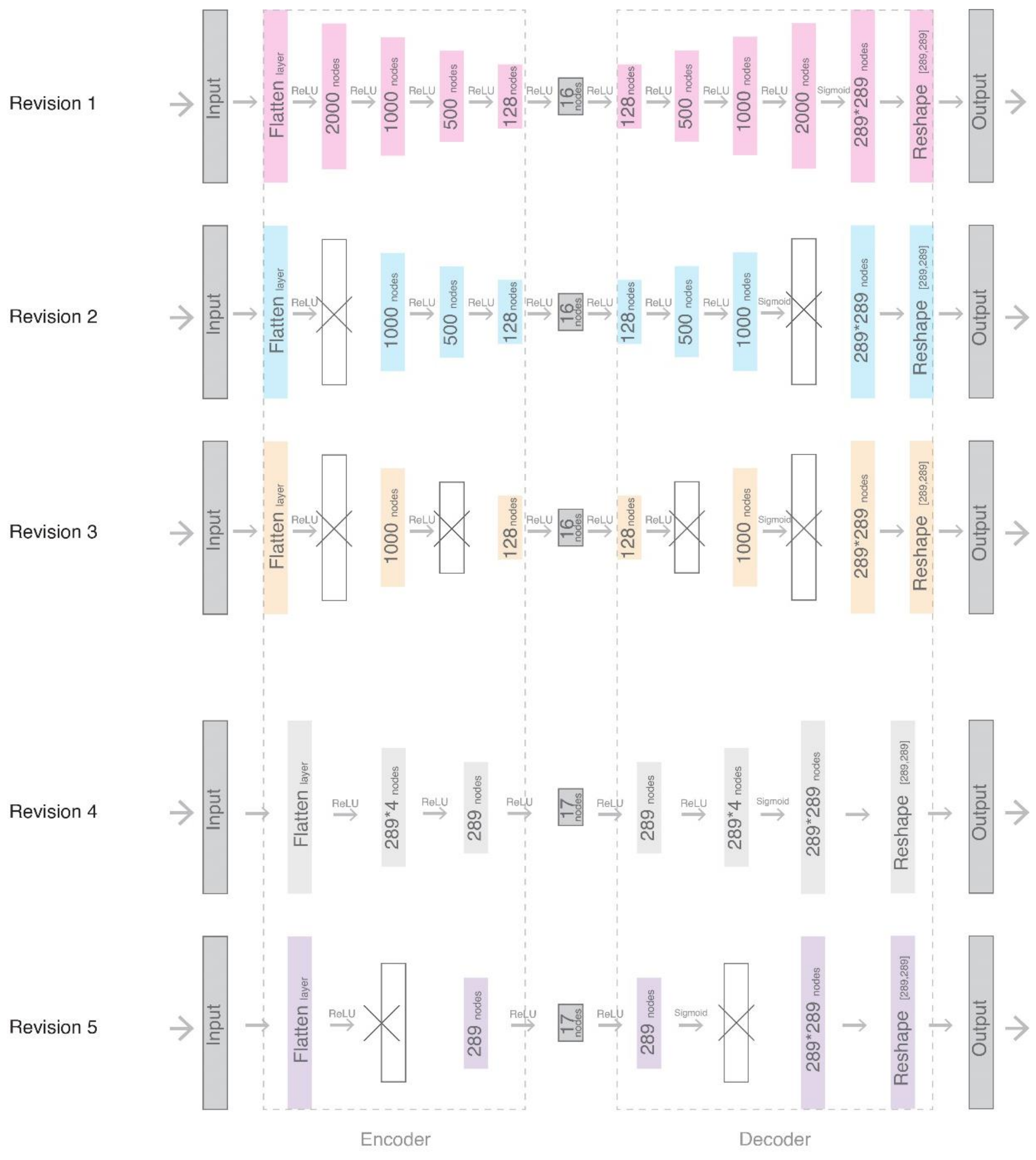


Figure 4.5 The tested architecture of the second's option VAE

Charts demonstrating the loss function during training, depending on the architecture of the encoder-decoder are shown below.

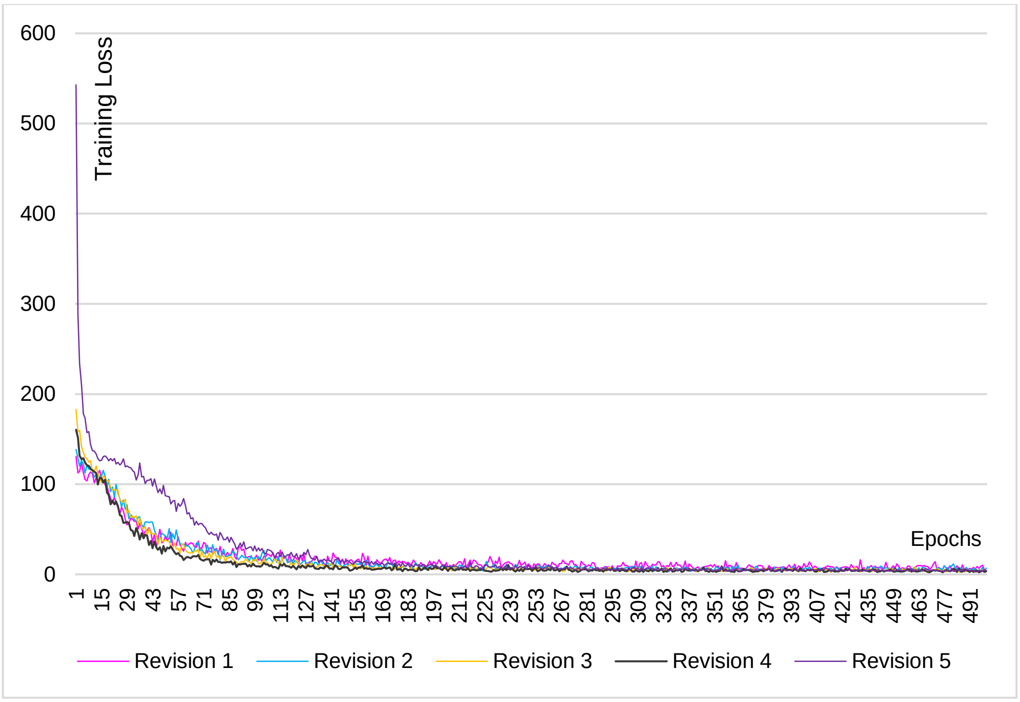


Chart 4.2 TheTraining Loss after 500 epochs

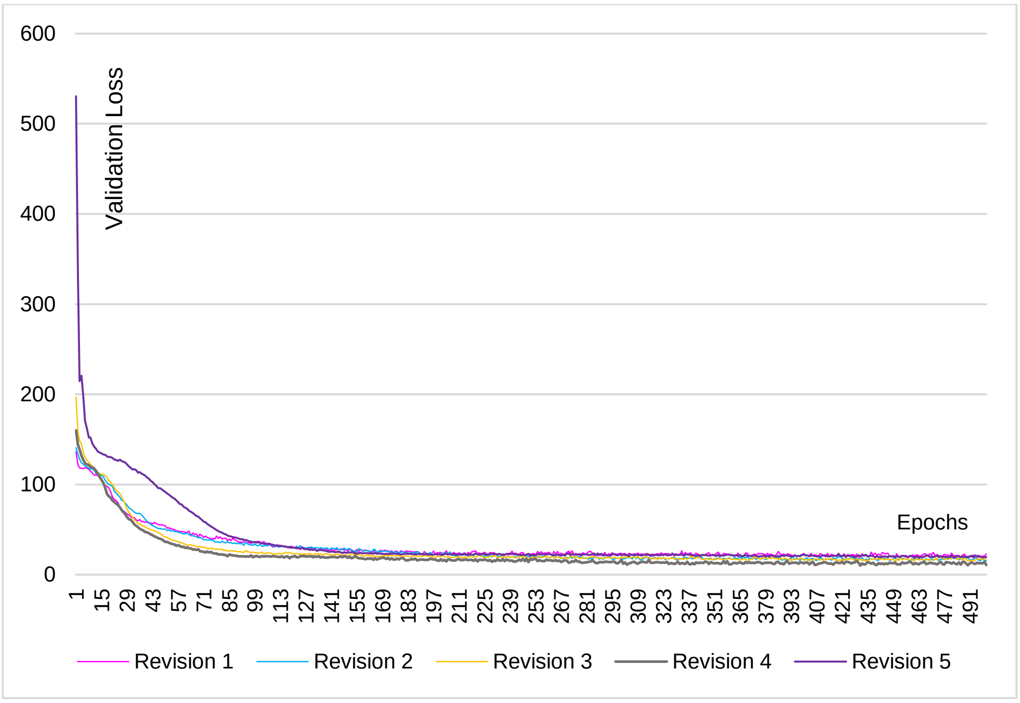


Chart 4.3 The Validation Loss after 500 epochs

For the best performing architecture (Revision 4) charts demonstrating the loss function during training, depending on the size of the latent space are shown below. Increasing the latent space leads to a drop in both loss and validation loss.

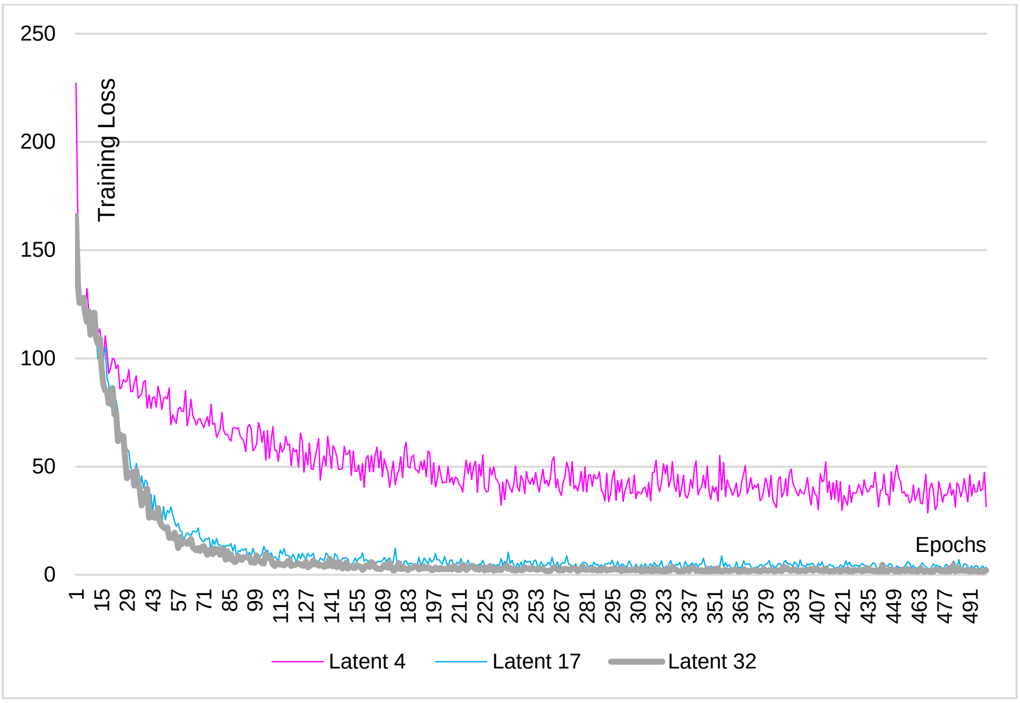


Chart 4.4 The Training Loss after 500 epochs

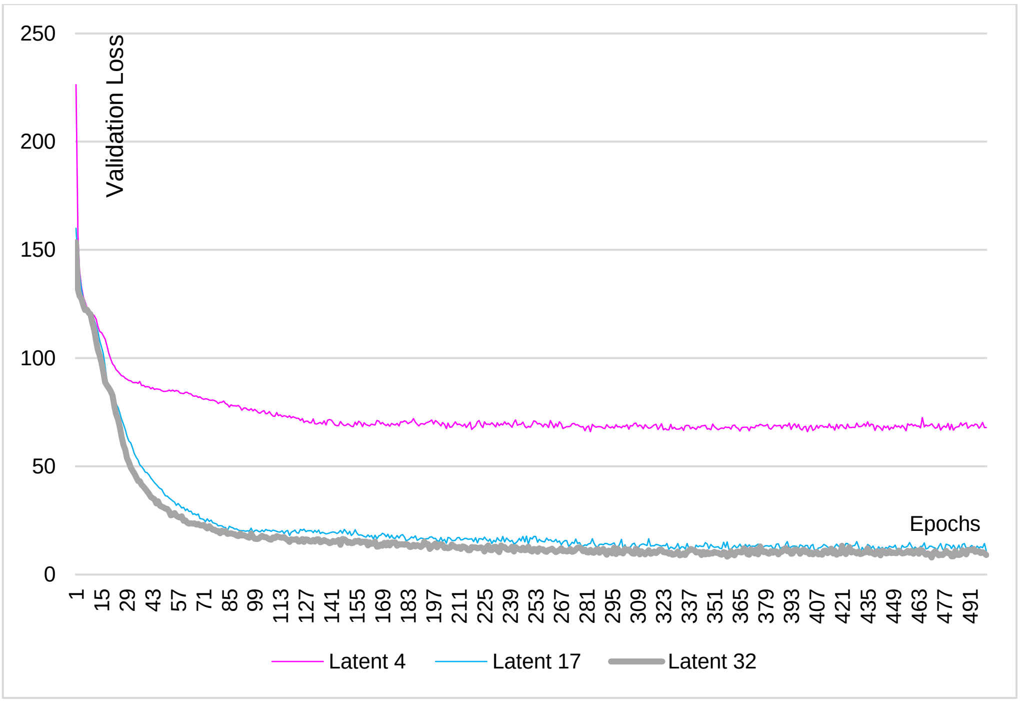


Chart 4.5 The Validation Loss after 500 epochs

Results

Below you can see results of some of the best performed meshes, which were not also used when training

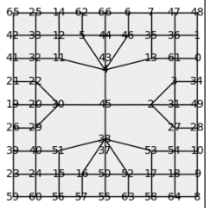
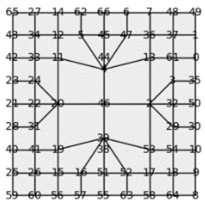
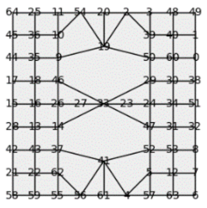
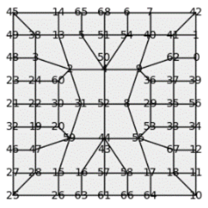
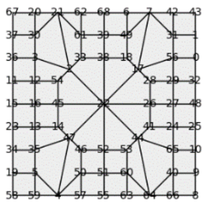
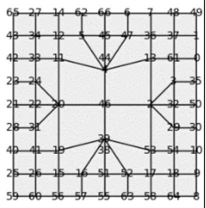
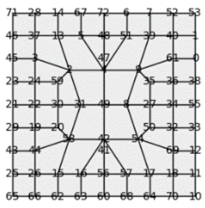
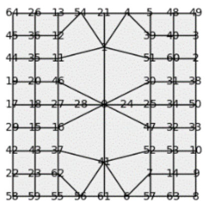
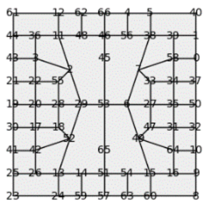
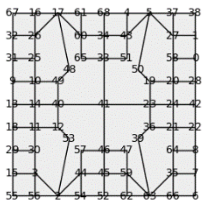
Original Samples					
Decoded Samples					

Table 4.1 Some of the best performed samples and their decoded result.

These are some decoded meshes from within the train dataset

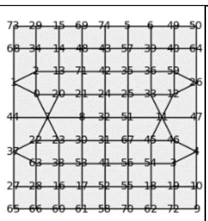
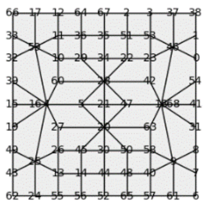
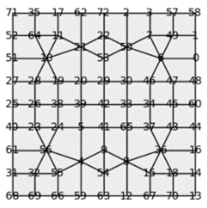
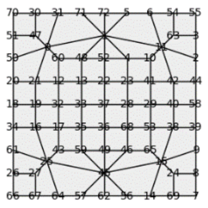
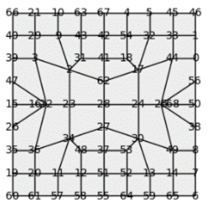
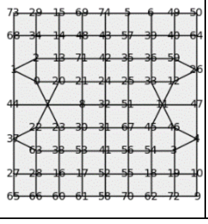
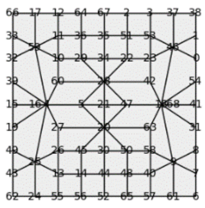
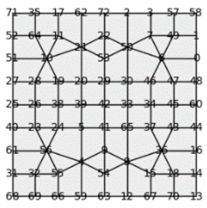
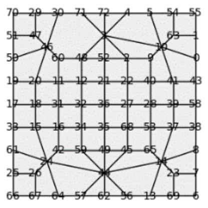
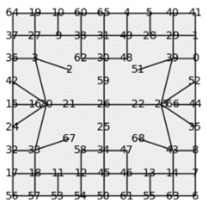
Original Samples					
Decoded Samples					

Table 4.2 Some of the training samples and their decoded result.

These are some random generated meshes

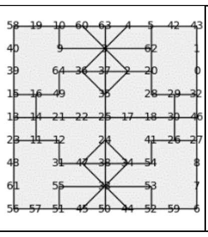
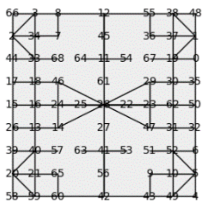
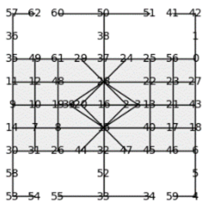
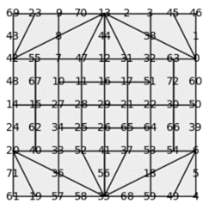
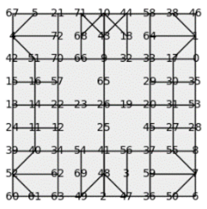
Original Samples					

Table 4.3 Random AI generated meshes.

Using a quarter of each of the adjacency matrices

A final attempt was done to test the performance of the network by minimizing the input information. Since the meshes are symmetrical we can keep one quarter of the adjacency matrices.

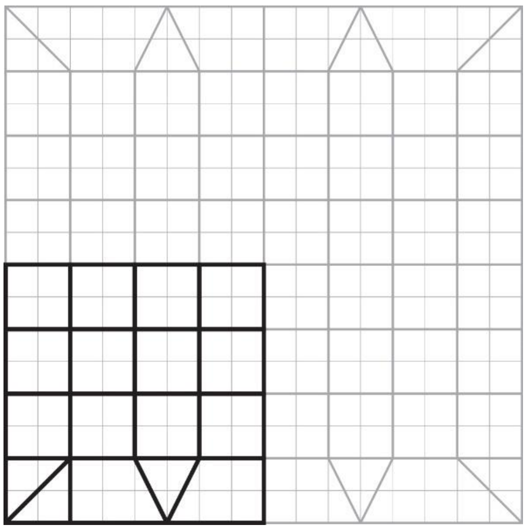


Figure 4.6 A quarter of the mesh

The original adjacency matrix has a shape of [289,289]. Keeping one quarter of the information means the new matrix has a shape of [81,81]. Therefore we need to keep in both axis the indices included in the shape bellow.

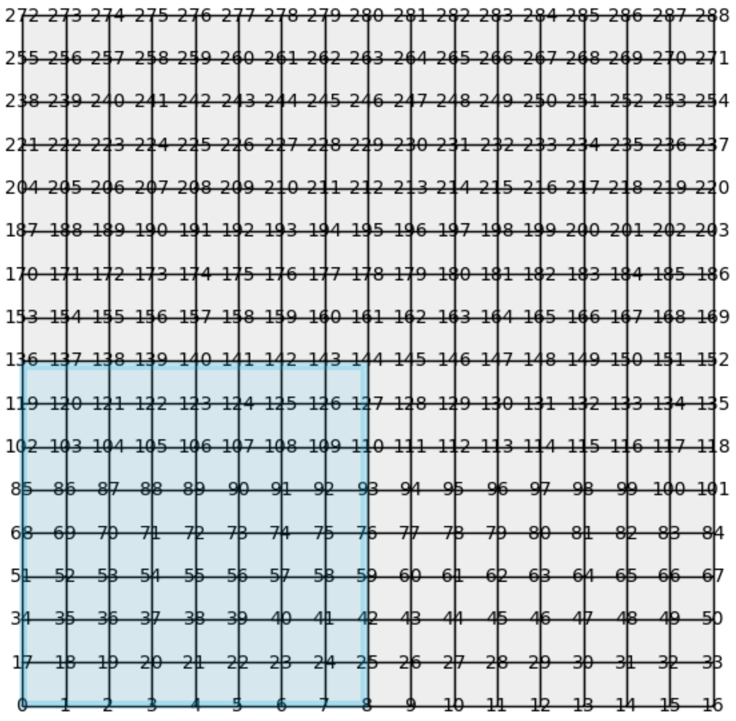


Figure 4.7 The selected vertices in the one quarter of the mesh

Again here fully connected layers were used for the encoder and the decoder. ReLU was used in all layers apart from the last one in the decoder. Schematic diagrams with the tested architectures of the encoder and the decoder are demonstrated below. The VAE architecture that performs better is that of **Revision 2**.

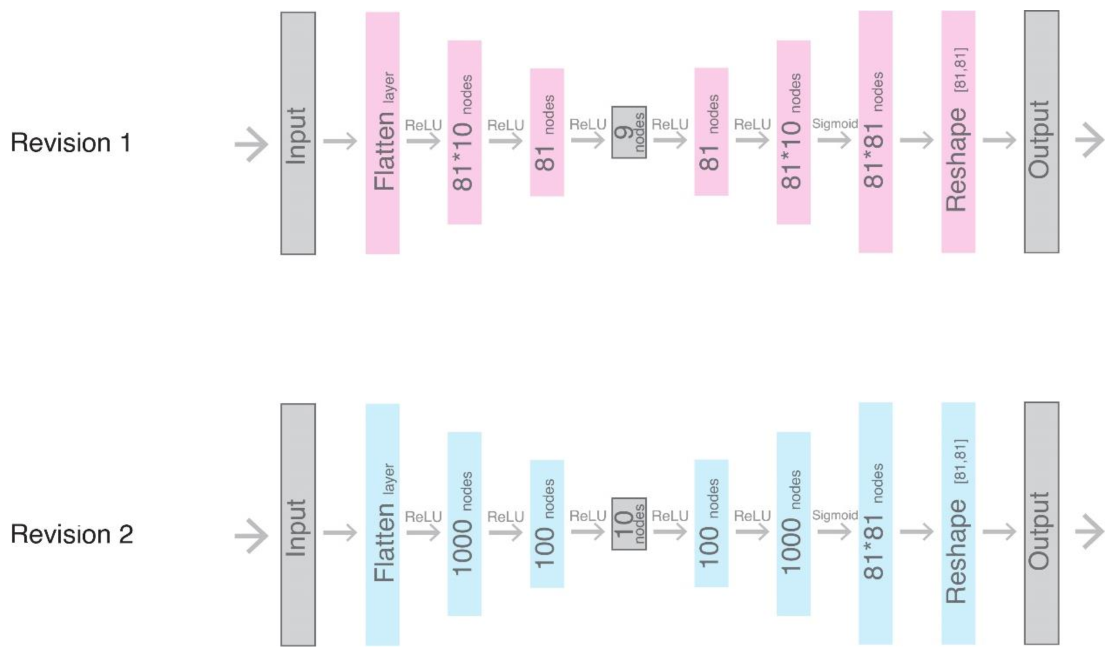


Figure 4.8 The tested architectures of the VAE for the second option using the information of one quarter of the mesh

Charts demonstrating the loss function during training, depending on the architecture of the encoder-decoder are shown below.

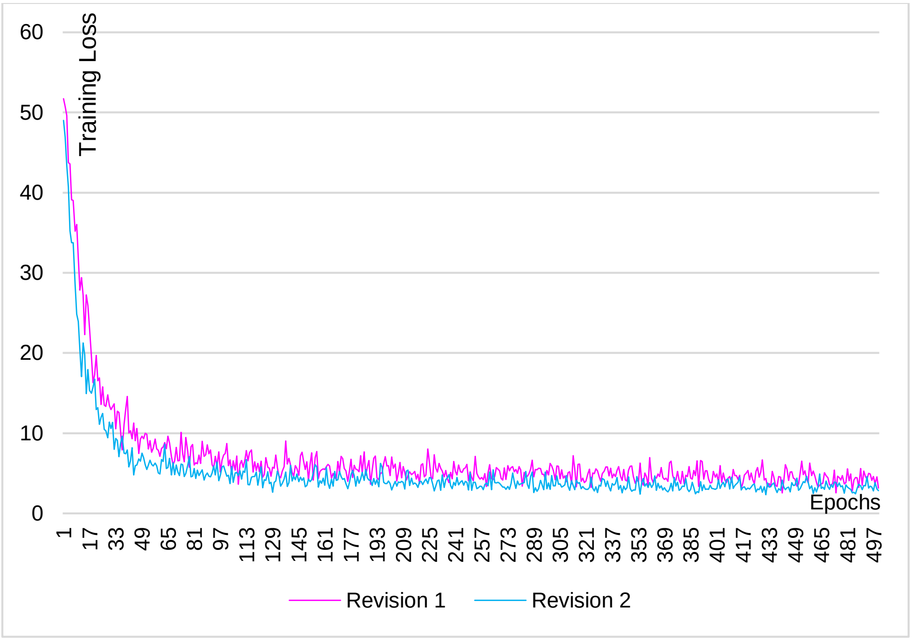


Chart 4.6 Training Loss after 500 epochs

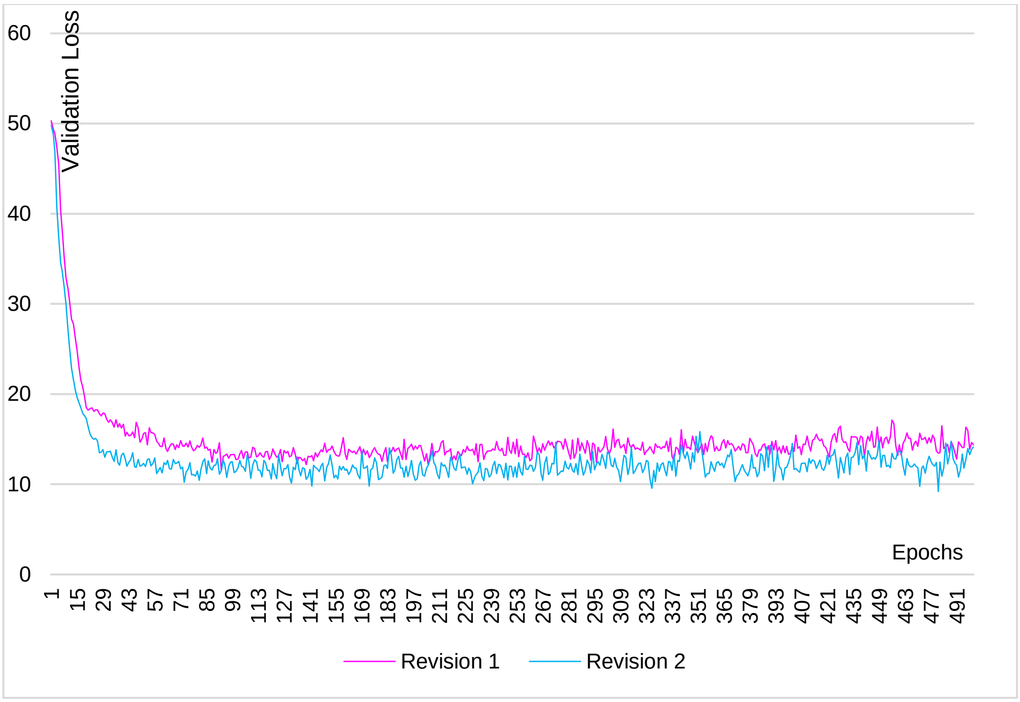


Chart 4.7 Validation Loss after 500 epochs

Results

Below results of original samples and their decoded outputs are presented

Results of some of the best performed meshes, which were not also used when training

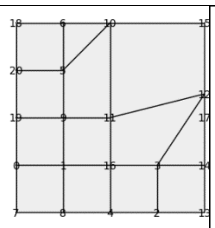
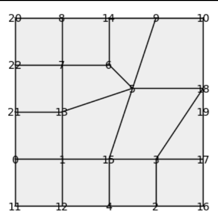
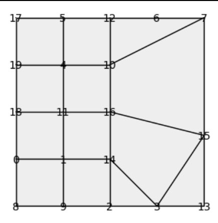
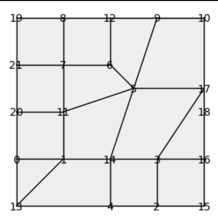
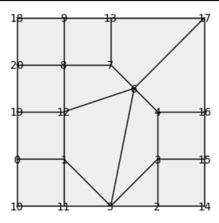
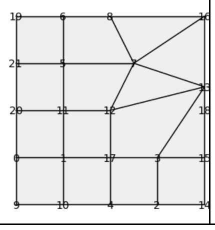
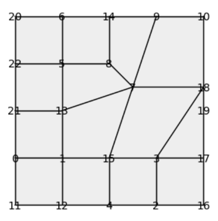
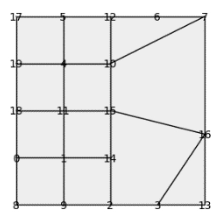
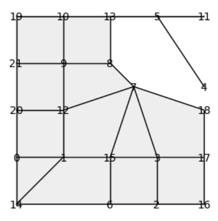
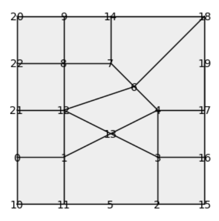
Original Samples					
Decoded Samples					

Table 4.4 Some of the best performed samples and their decoded result.

Decoded meshes from within the train dataset

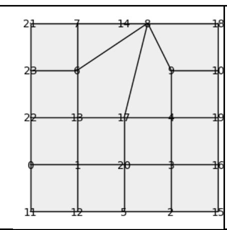
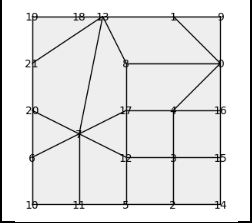
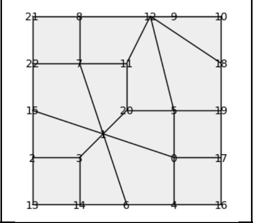
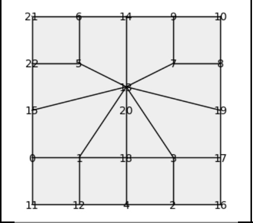
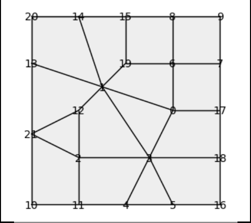
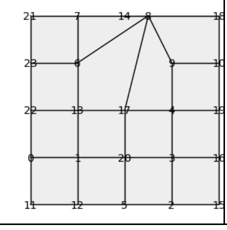
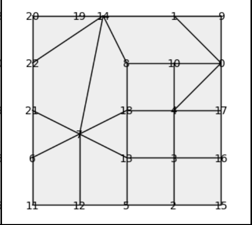
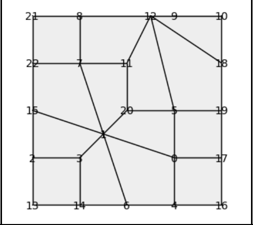
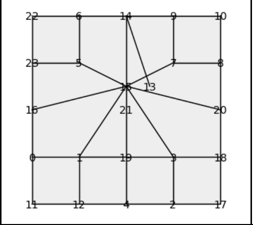
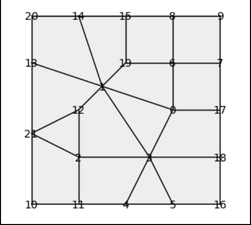
Original Samples					
Decoded Samples					

Table 4.5 Some of the training samples and their decoded result.

Random AI generated meshes

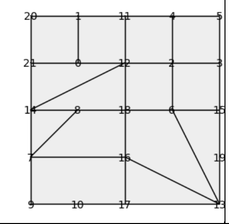
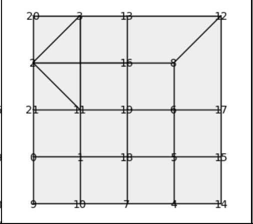
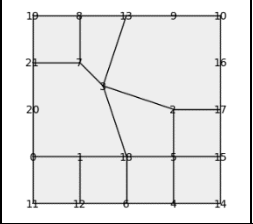
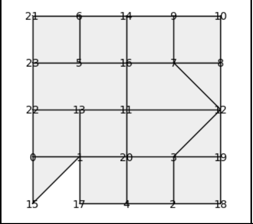
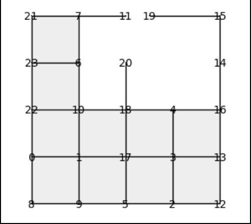
Original Samples					

Table 4.6 Random AI generated meshes

Option 3 : Using flattened arrays from the adjacency matrices

This option aimed to ease the training by minimizing the size of the training samples. This option managed to converge the model faster and minimize more the function loss. For simplification reason the information that represents the one quarter of the mesh was also used.

The tested architectures are demonstrated below.

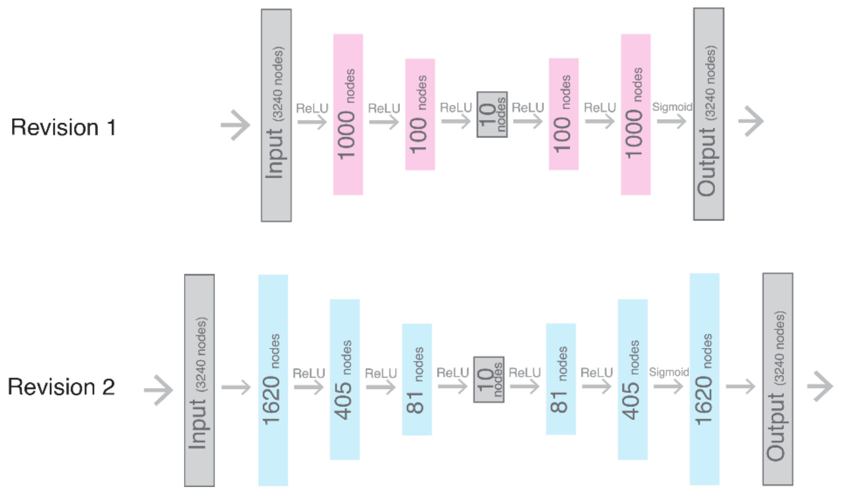


Figure 4.9 The tested architectures of the VAE for the third option

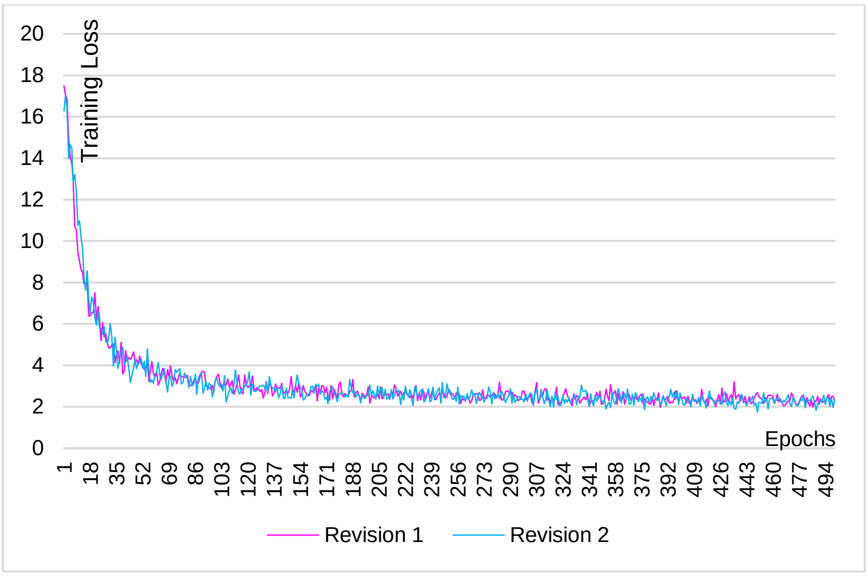


Chart 4.8 Loss after 500 epochs

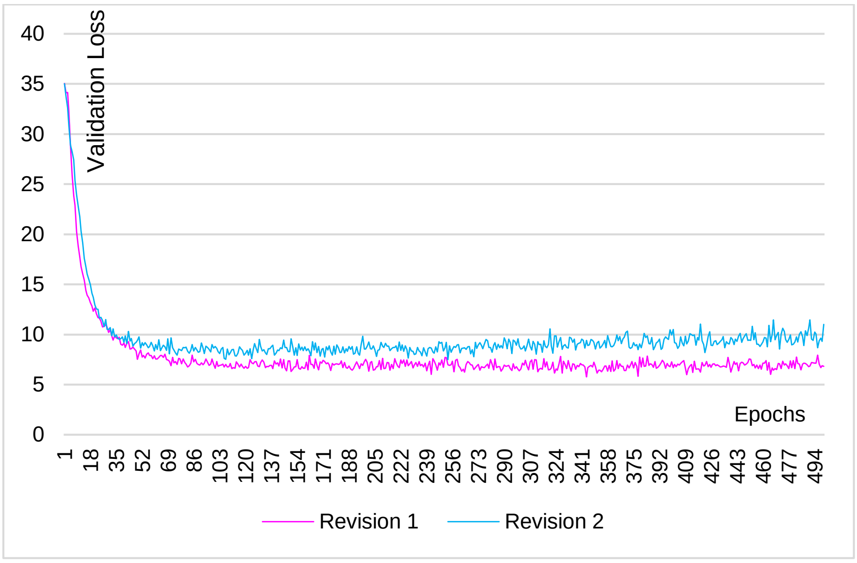


Chart 4.9 Validation loss after 500 epochs

The first revision manages to drop the validation loss more than the second. Moreover the model is smaller on size. Therefore this is the final architecture, also chosen for the next steps.

Results of some of the best performed meshes, which were not also used when training

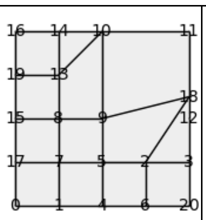
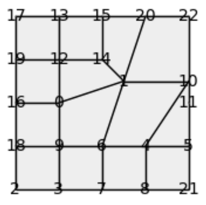
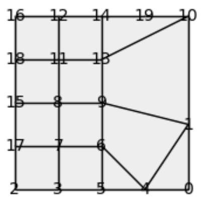
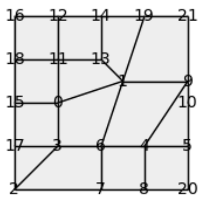
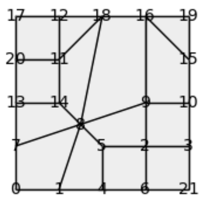
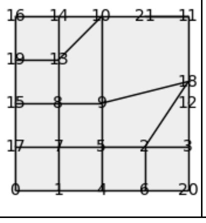
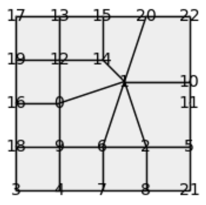
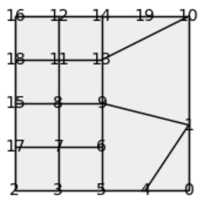
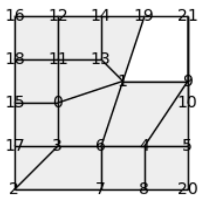
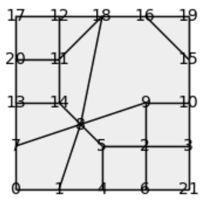
Original Samples					
Decoded Samples					

Table 4.7 Some of the best performed samples and their decoded result.

Random AI generated meshes

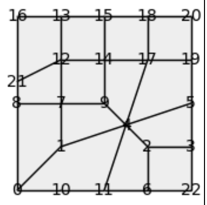
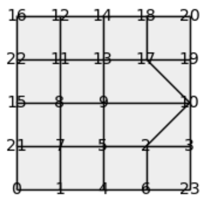
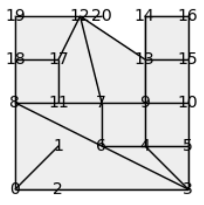
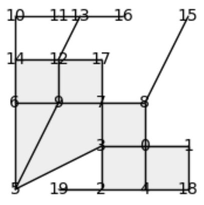
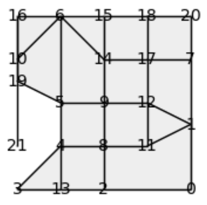
Original Samples					

Table 4.8 Random AI generated meshes

4.2 Surrogate Model

Training a surrogate model to predict the normalized performance out of the VAE's decoded data

The Surrogate Model attempts to fit the input data `x_train` to the performance scores `y_train`. Input data are the decoded samples. 50 samples are excluded from the training process to evaluate the training. Input shape is [1,3240] and the output is a single score.

The architecture tested included an input layer, a flatten layer and two dense layers.

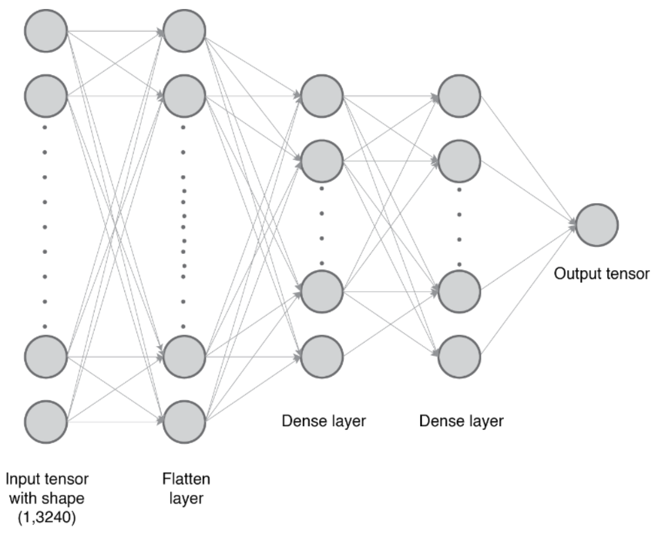


Figure 4.10 The architecture of the Surrogate Model.

The following architectures were tested. The architecture whose loss function is better converged is that of the **Revision 1**.

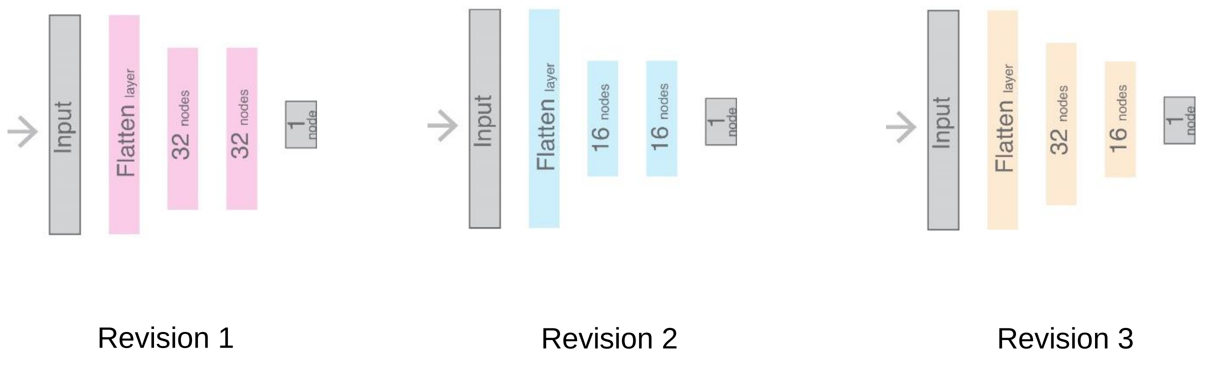


Figure 4.11 The tested architectures for the Surrogate Model.

Charts demonstrating the loss function during training, depending on the architecture of the Surrogate Model are shown below.

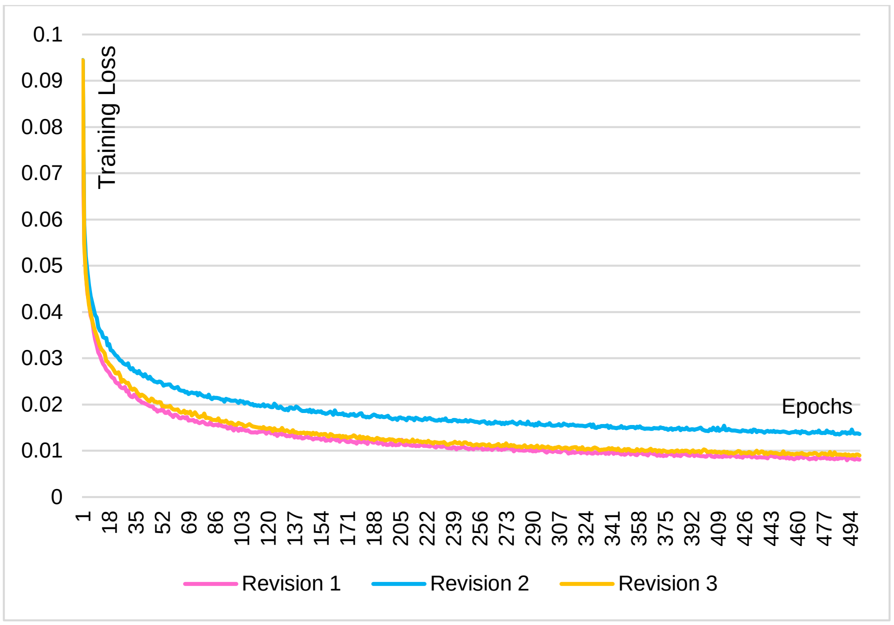


Chart 4.10   Lloss after 500 epochs

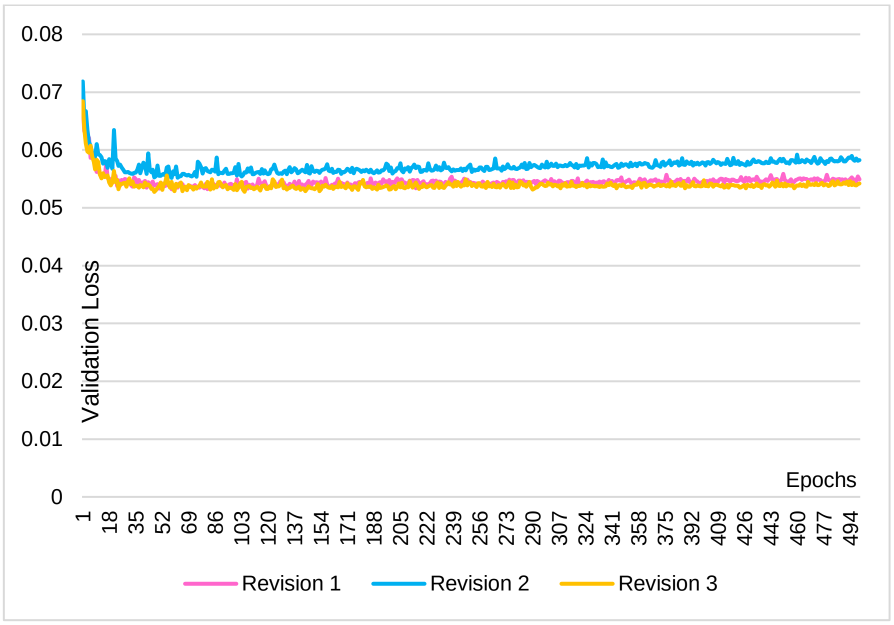


Chart 4.11 Validation loss after 500 epochs

The following Chart demonstrates the loss function after 1000 epochs. The models starts validation loss starts increasing slightly after 200 epochs therefore the training stops there

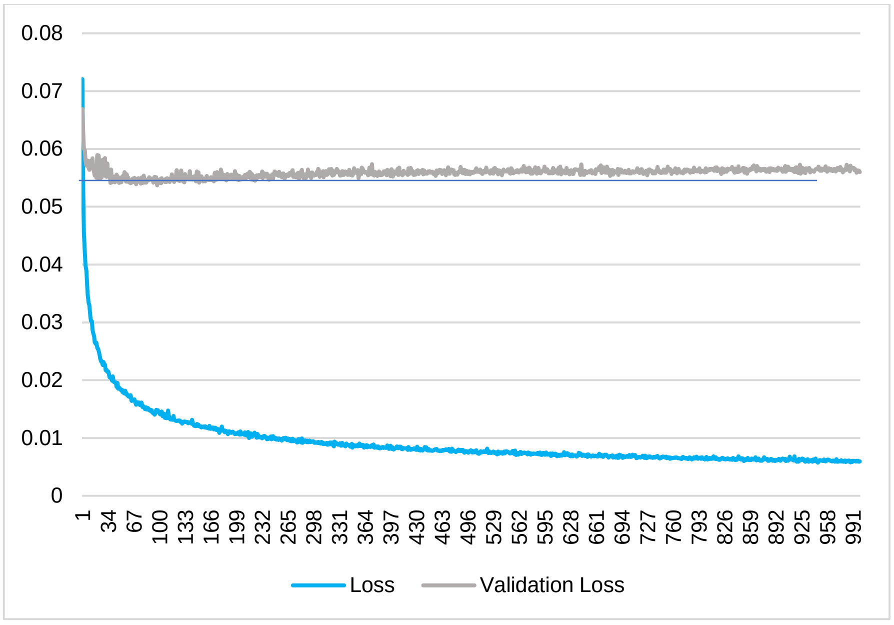


Chart 4.12 Loss and Validation loss after 1000 epochs for Revision 1

The following chart demonstrates the evaluation of the results on 50 samples that were excluded from training. The cyan dots are the true performance scores and the grey dots are the model's estimated performance

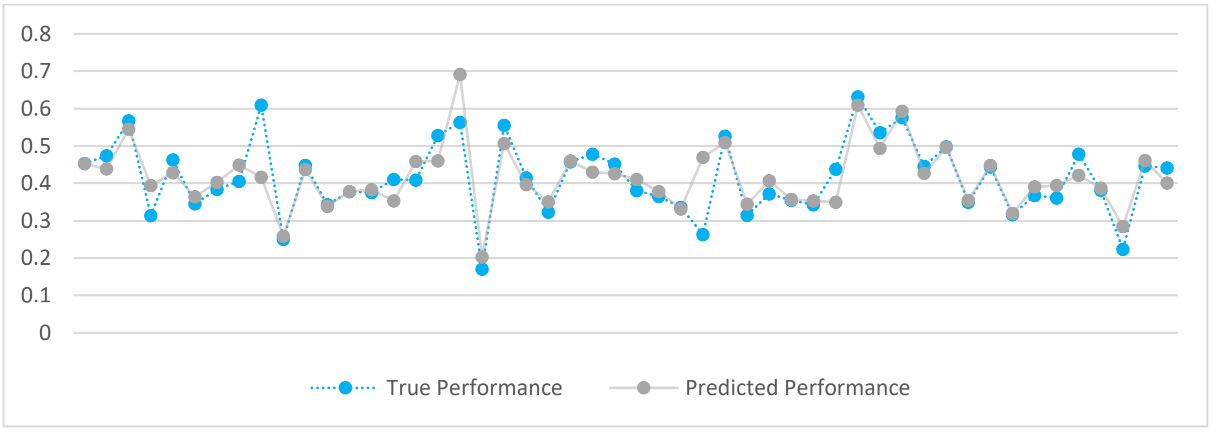


Chart 4.13 Comparison of predicted and actual performance for 50 test data that were not used when training the surrogate model

The following chart demonstrates the evaluation of the results for the 10 best samples that were also excluded from training. Their predicted scores are numbers around 0.20. That can be explained from the fact that below 0.20 (about 150 out of the 7.338) and none below 0.12. Therefore the best performed samples are scored around the number 0.20.

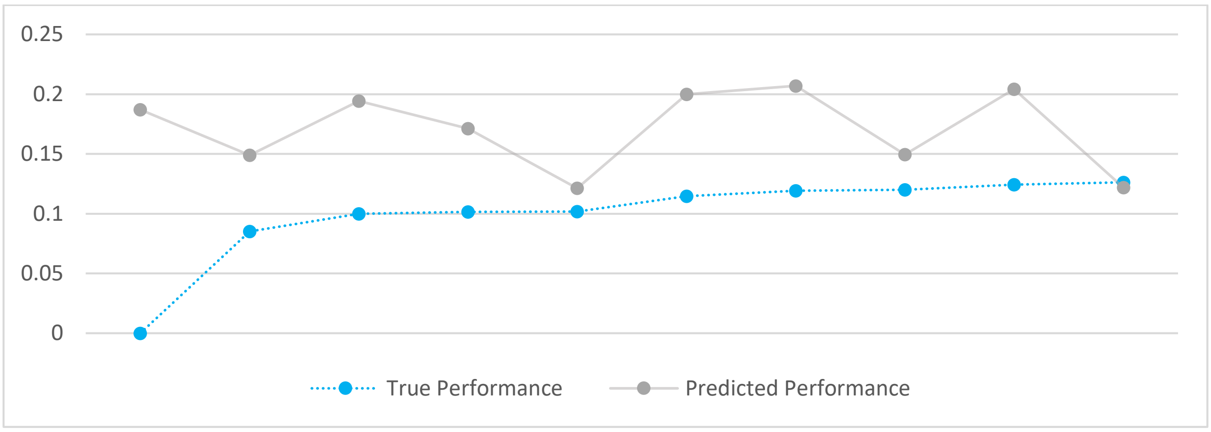


Chart 4.14 Comparison of predicted and actual performance for 10 best performed samples

Attempt to train the Surrogate model to predict the normalized performance out of the VAE's encoded data

Another option that was explored is if the model can be trained with the encoded vectors for the case of the second's option fourth revision. As shown in the picture below the model was overfitting therefore this effort was abandoned.

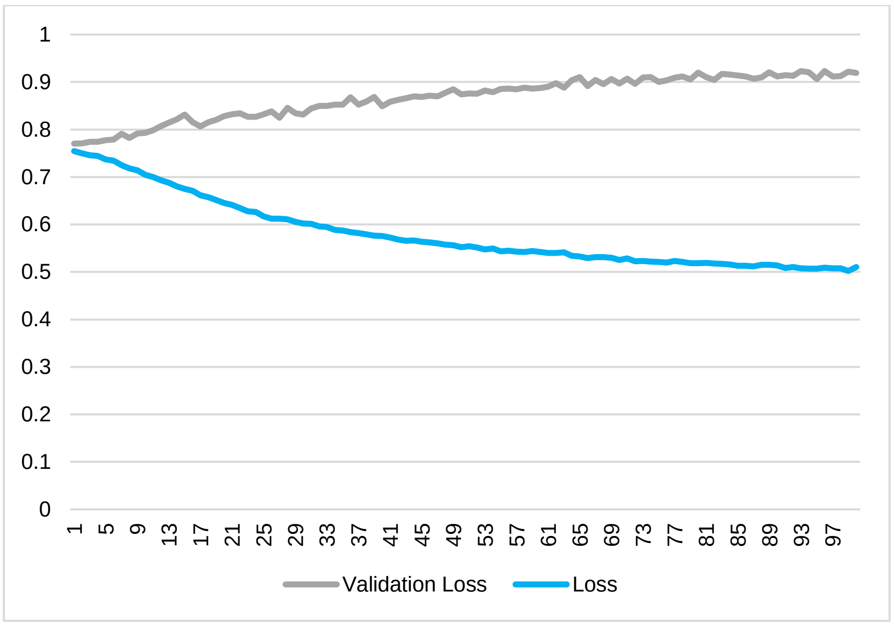


Chart 4.15 Loss and validation loss for the surrogate model trainned with encoded vectors

4.3 Gradient Descent Optimization

After training the VAE and the Surrogate model we are able to retrieve the Model's gradients. The gradient we are interested in is that of the structural 'performance' (y vector) with respect to the latent space (z vector). The Gradient Descent algorithm searches for the minimum y vector.

$Z = Z - lr \frac{\partial y}{\partial z} (Z_0, Z_{n,})$

lr : Learning rate that determines how large the update or moving step is.
Z: The latent's space z vector to be updated
Y: Structural Performance

The following diagram explains how the gradient descent optimization works:

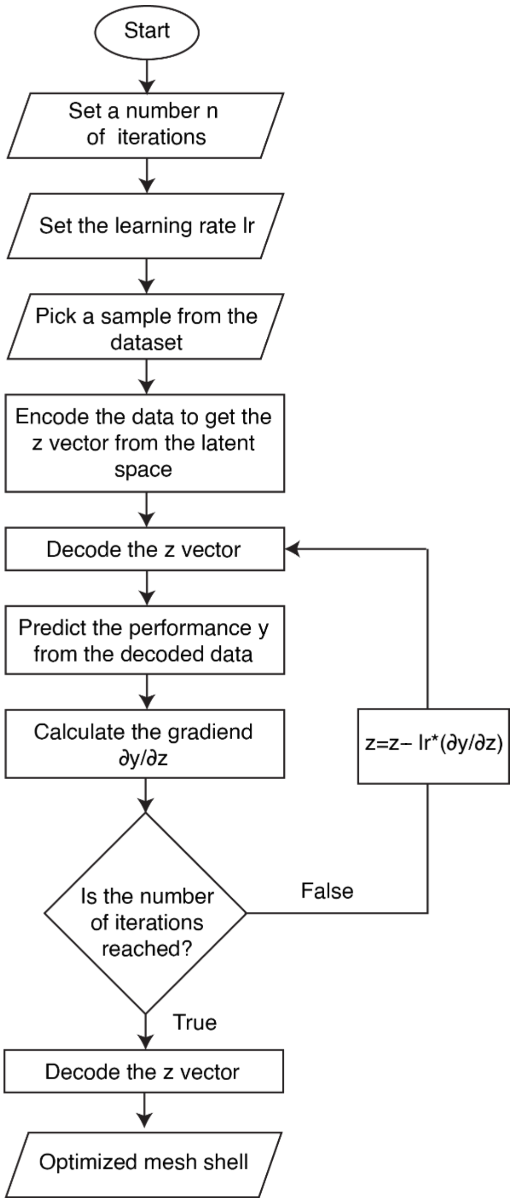

Start
Set a number n of iterations
Set the learning rate lr
Pick a sample from the dataset
Encode the data to get the z vector from the latent space
Decode the z vector
Predict the performance y from the decoded data
Calculate the gradiend ∂y/∂z
z=z– lr*(∂y/∂z)
Is the number of iterations reached?
False
True
Decode the z vector
Optimized mesh shell

Figure 4.12 Flow Chart for the Gradient Descent.

Some generated adjacency matrices are not symmetrical and have extra edges. To check if they already exist in the dataset the extra edges are cleared manually and a final step is executed to make the array symmetrical if it is not. The code for this is described below:

```
for ix in range(adj_mtx.shape[0]):
    for iy in range(adj_mtx.shape[1]):
        if adj_mtx [ix,iy]+ adj_mtx [ix,iy]==1:
            adj_mtx [ix,iy]=1
            adj_mtx [iy,ix]=1
```

The following examples present optimization cases when running the GD for random design. In Cases 1,2,3,4 the GD managed to find novel and better designs than the provided training dataset. In Cases 2 and 3 with a learning rate of 2.5 it managed to find designs almost identical to best performing one that was excluded from the training process.

Case 1.

Learning rate: **0.5**
Iterations: **1000**

Performance score of initial design: **0.37526757**
Estimated performance score of the optimized design: **0.1391386**
Actual performance score of the optimized design: **0.113302**
Novel Design: **Yes**

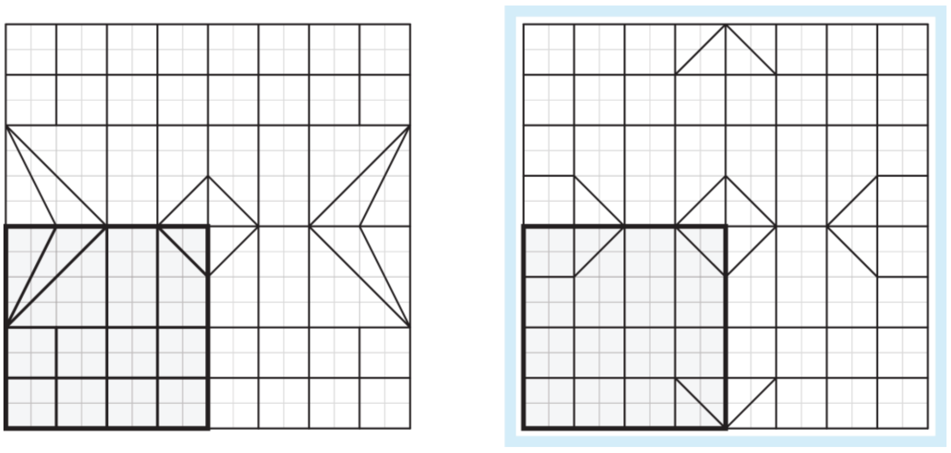


Figure 4.13 For the Case 1 2 and a learning rate of 0.5, an initial mesh(on the left) and the optimized one(on the right).

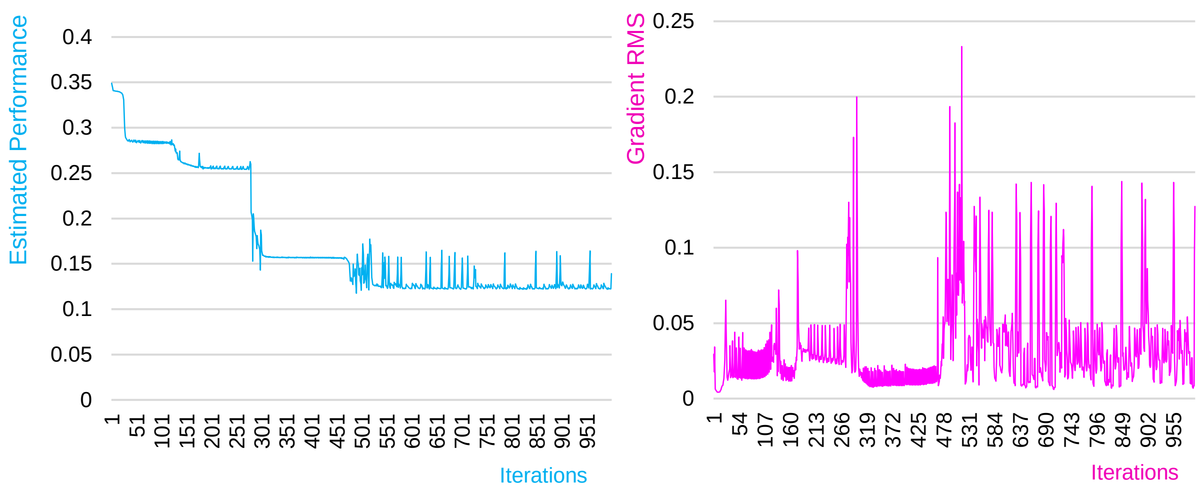


Chart 4.16 For Case 1 with a learing rate of 0.5: Charts demonstrating the values of the performance score (on the left) and the root mean square of the gradients of the performance with respect to latent's space vector (on the right), during the optimization after 1000 iterations.

With a learning rate of 0.5 the GD finds a similar but better performing design, Increasing the learning rate at 2.5 results also in an better design, not as good as the one the one found with the 0.5 learning rate, as the GD skipped it.

Learning rate: **2.5**
Iterations: **1000**

Performance score of initial design: **0.37526757**
Estimated performance score of the optimized design: **0.32426813**
Actual performance score of the optimized design: **0.302315**
Novel Design: **No**

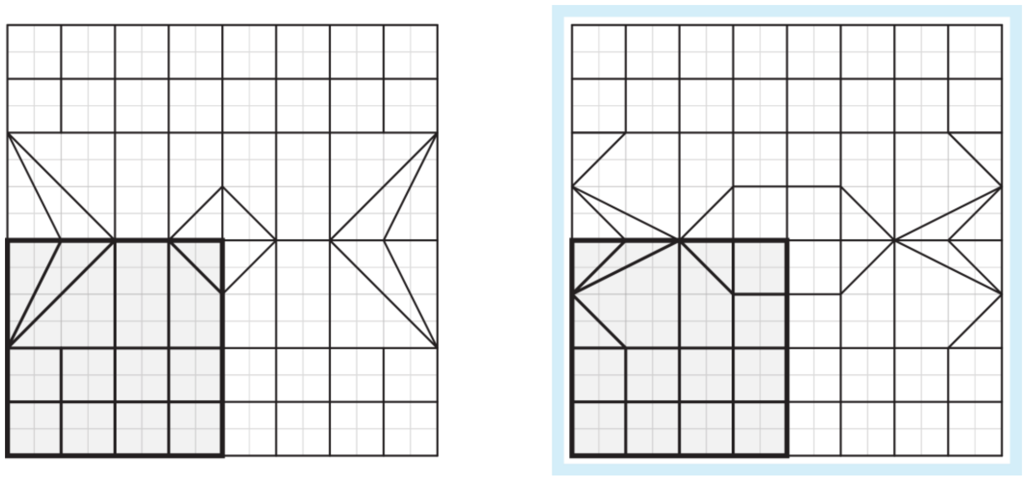


Figure 4.14 For the Case 1 and a learning rate of 2.5, an initial mesh(on the left) and the optimized one(on the right).

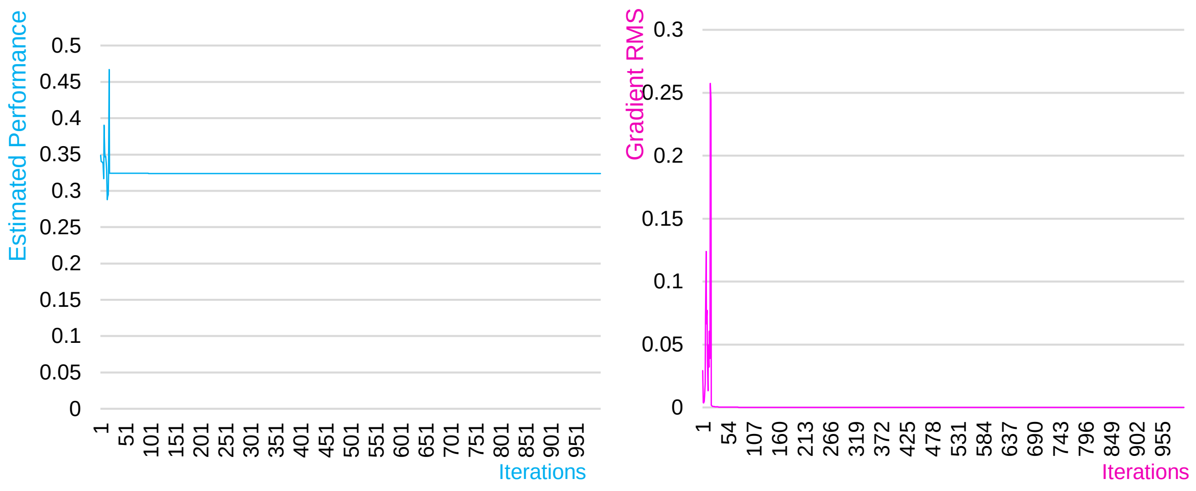


Chart 4.17 For Case 1 with a learing rate of 2.5: Charts demonstrating the values of the performance score (on the left) and the root mean square of the gradients of the performance with respect to latent's space vector (on the right), during the optimization after 1000 iterations.

Case 2.

Learning rate: **0.5**
Iterations: **1000**
Performance score of initial design: **0.17705911**
Estimated performance score of the optimized design: **0.1425786**
Actual performance score of the optimized design: **0.085932**
Novel Design: **Yes**

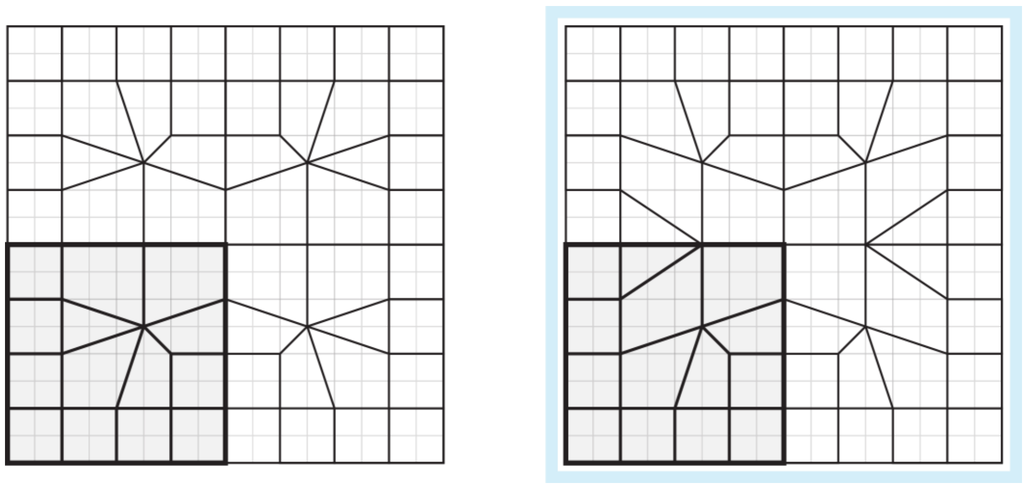


Figure 4.15 For the Case 2 and a learning rate of 0.5, an initial mesh(on the left) and the optimized one(on the right).

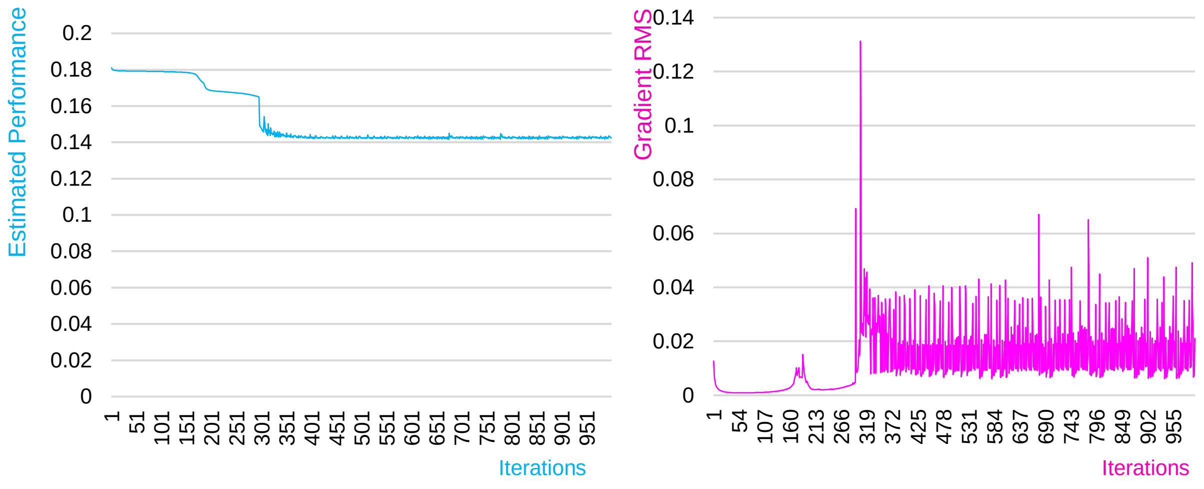


Chart 4.18 For Case 2 with a learing rate of 0.5: Charts demonstrating the values of the performance score (on the left) and the root mean square of the gradients of the performance with respect to latent's space vector (on the right), during the optimization after 1000 iterations.

Increasing the learning rate at 1.5 results in an optimized design, with GD jumping faster into new solutions.

Learning rate: **2.5**
Iterations: **1164**
Performance score of initial design: **0.17705911**
Estimated performance score of the optimized design: **0.16579011**
Actual performance score of the optimized design: **0.044936816**
Novel Design: **Yes**

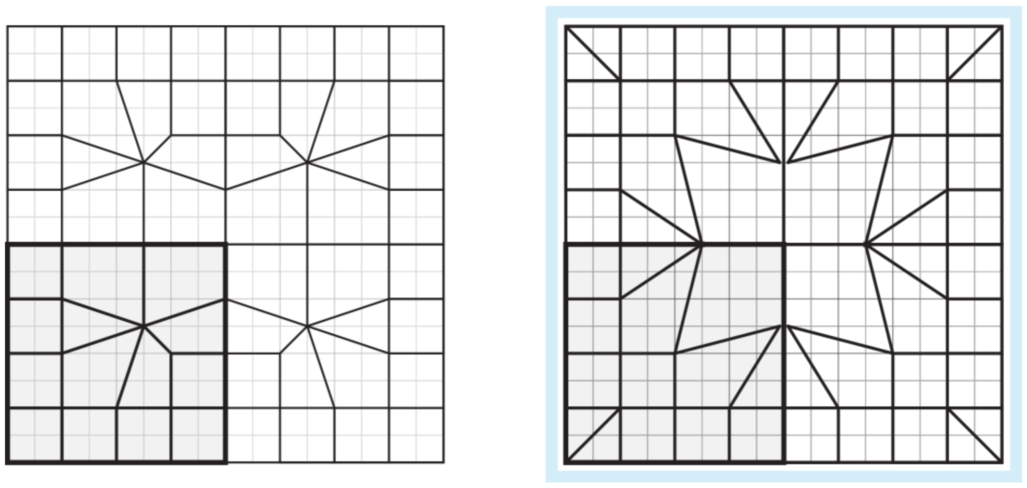
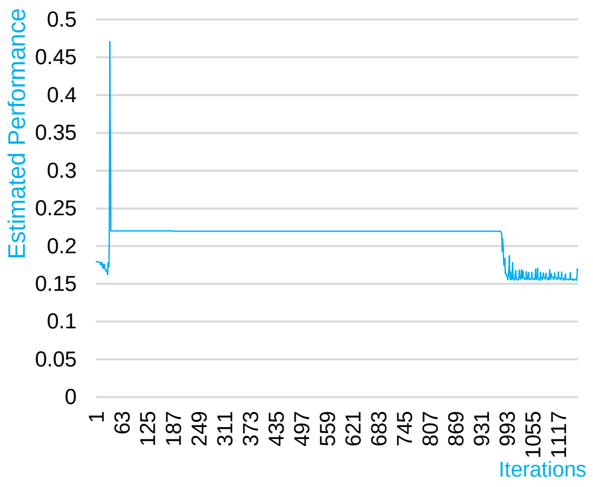


Figure 4.16 For the Case 2 and a learning rate of 2.5, an initial mesh(on the left) and the optimized one(on the right).



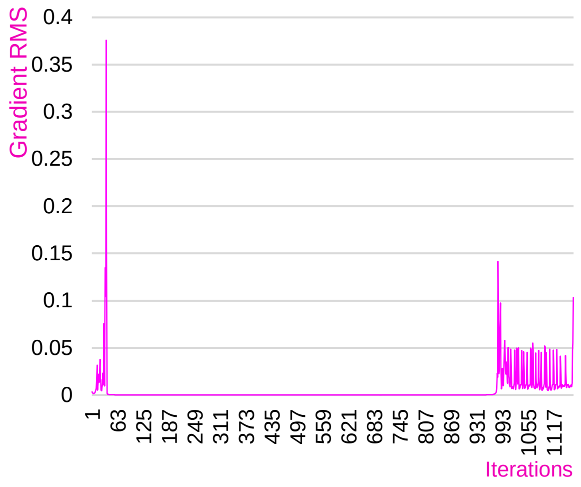


Chart 4.19 For Case 2 with a learing rate of 1.5: Charts demonstrating the values of the performance score (on the left) and the root mean square of the gradients of the performance with respect to latent's space vector (on the right), during the optimization after 1000 iterations.

Case 3.

Learning rate: **2.5**
Iterations: **1000**

Performance score of initial design: **0.4659505**
Estimated performance score of the optimized design: **0.15925622**
Actual performance score of the optimized design: **0.01798574**
Novel Design: **Yes**

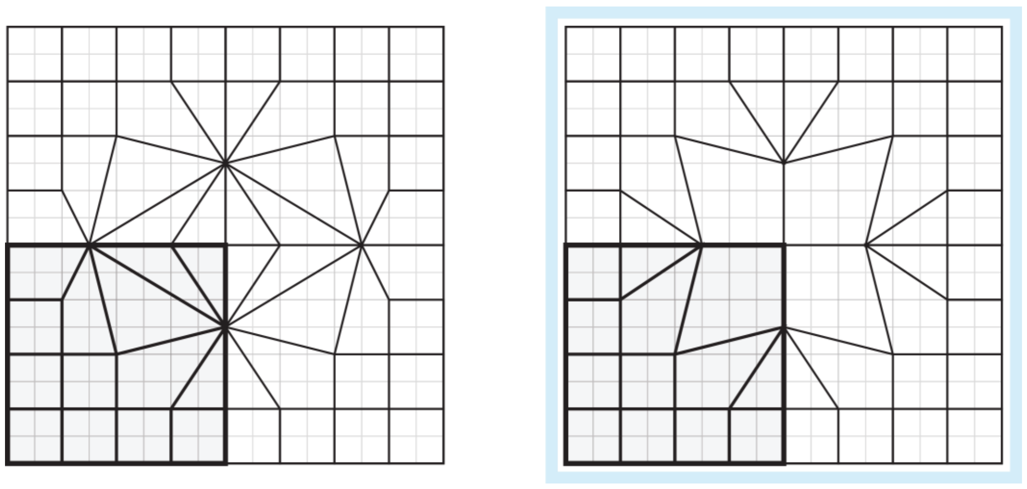
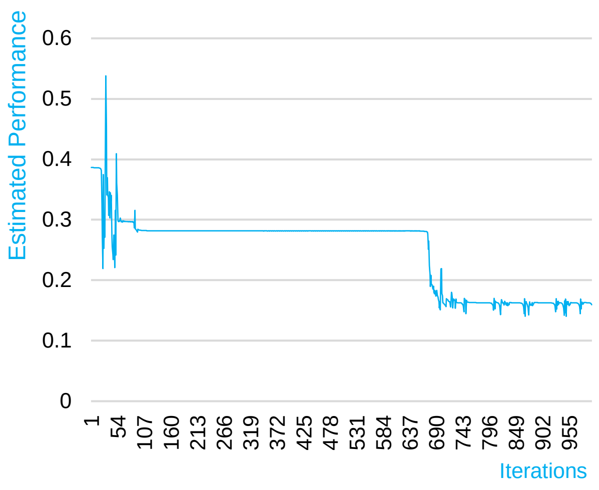


Figure 4.17 For the Case 3, an initial mesh(on the left) and the optimized one(on the right).



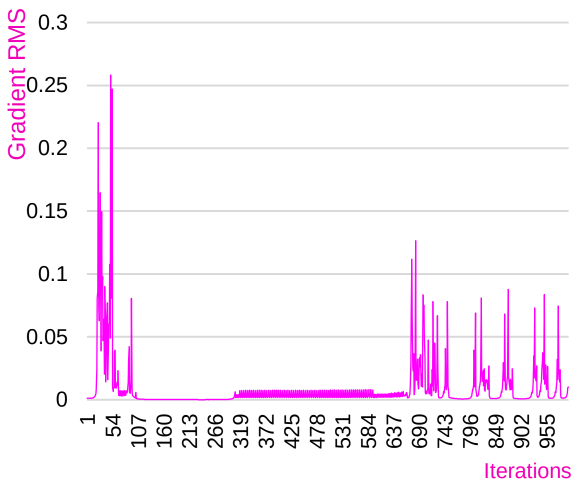


Chart 4.20 For Case 3 with a learing rate of 2.5: Charts demonstrating the values of the performance score (on the left) and the root mean square of the gradients of the performance with respect to latent's space vector (on the right), during the optimization after 1000 iterations.

Case 4.

Learning rate: **2.5**
Iterations: **1000**

Performance score of initial design: **0.18231553**
Estimated performance score of the optimized design: **0.12000429**
Actual performance score of the optimized design: **0.113239**
Novel Design: **Yes**

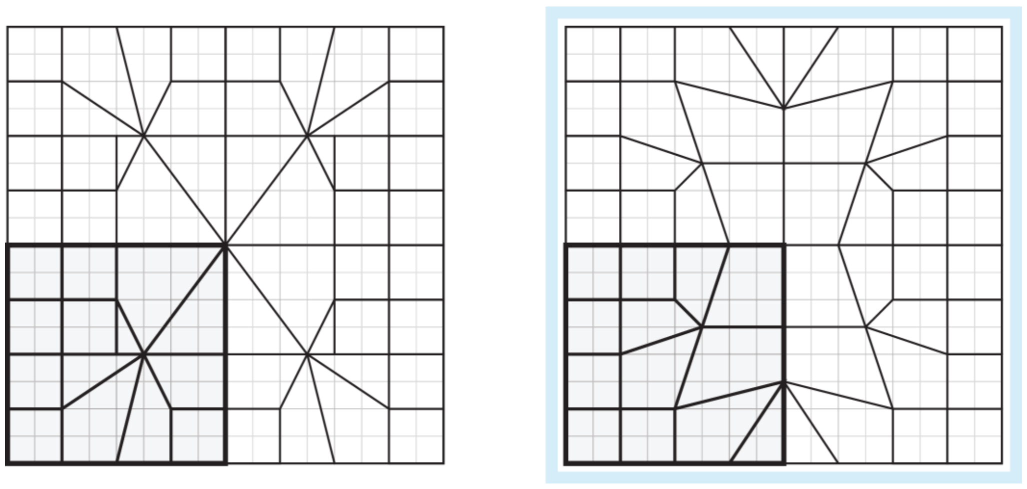
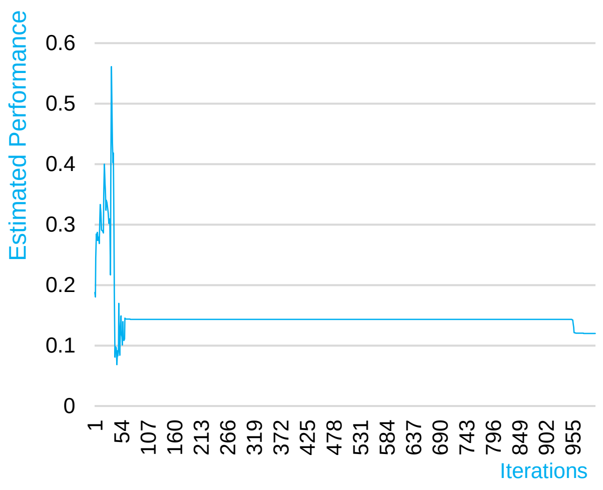


Figure 4.18 For the Case 4, an initial mesh(on the left) and the optimized one(on the right).



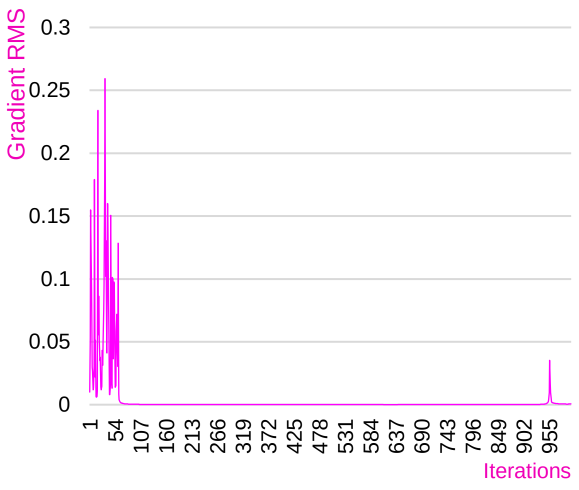


Chart 4.21 For Case 4 with a learing rate of 2.5: Charts demonstrating the values of the performance score (on the left) and the root mean square of the gradients of the performance with respect to latent's space vector (on the right), during the optimization after 1000 iterations.

Case 5.

Learning rate: **5**
Iterations: **1000**

Performance score of initial design: **0.3534903**
Estimated performance score of the optimized design: **0.1897085**
Actual performance score of the optimized design: **0.115608**
Novel Design: **Yes**

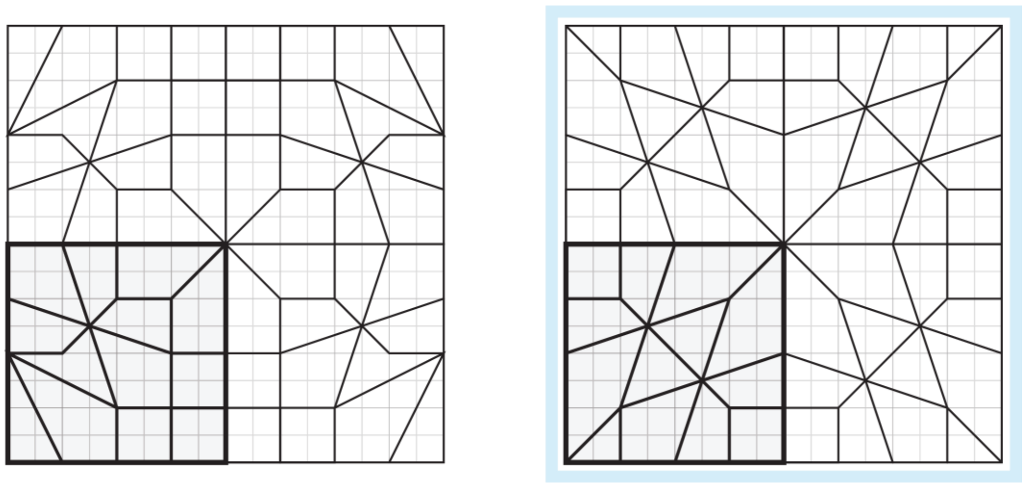


Figure 4.19 For the Case 5, an initial mesh(on the left) and the optimized one(on the right).

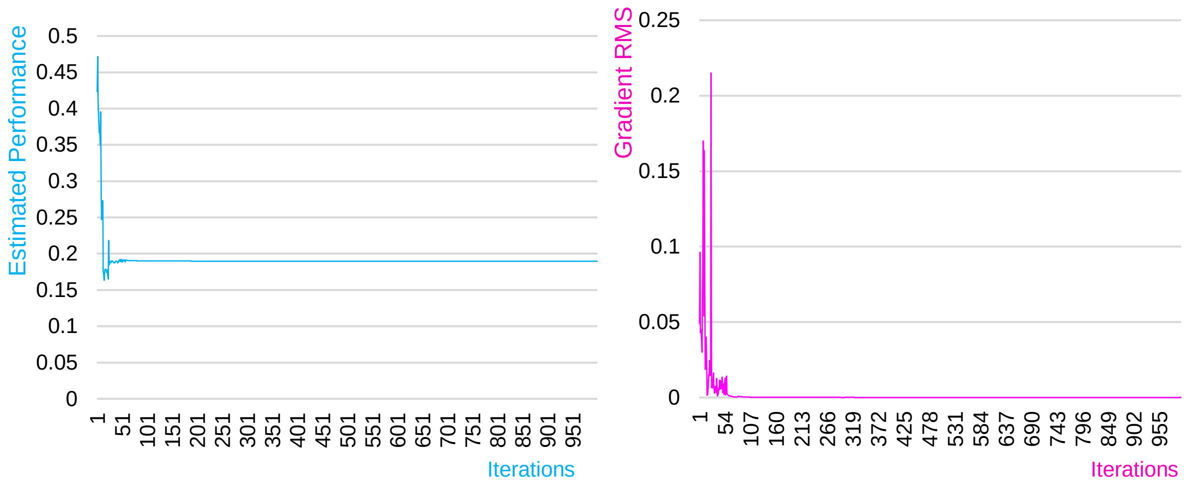


Chart 4.22 For Case 5 with a learing rate of 5: Charts demonstrating the values of the performance score (on the left) and the root mean square of the gradients of the performance with respect to latent's space vector (on the right), during the optimization after 1000 iterations.

Invalid Case

VAE may produce invalid designs that the surrogate model has not been trained to score their performance. Therefore, often Gradient Descent converged on invalid solutions. Below an invalid case is presented..

Learning rate: **0.5**
Iterations: **1000**

Performance score of initial design: **0.415089337**
Estimated performance score of the optimized design: **0.1145393**

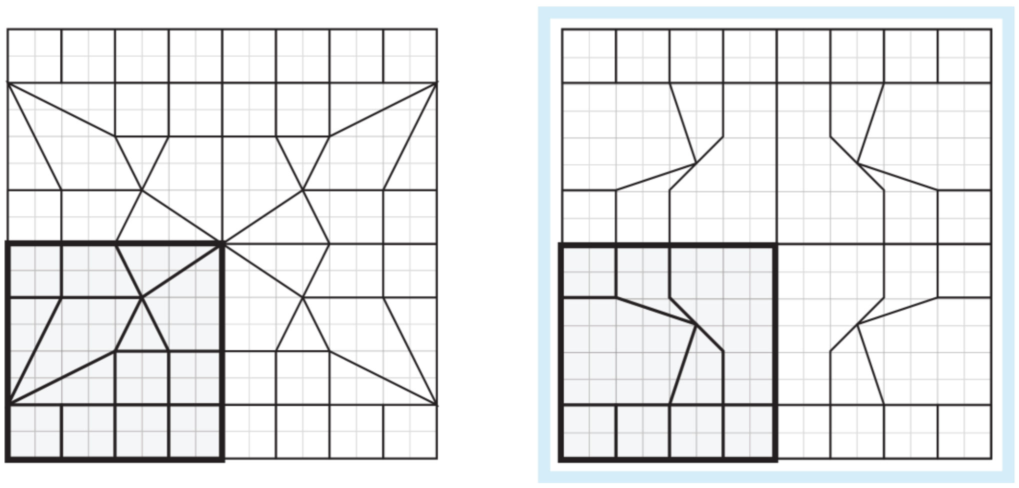


Figure 4.20 For an Invalid Case, an initial mesh(on the left) and the optimized one(on the right).

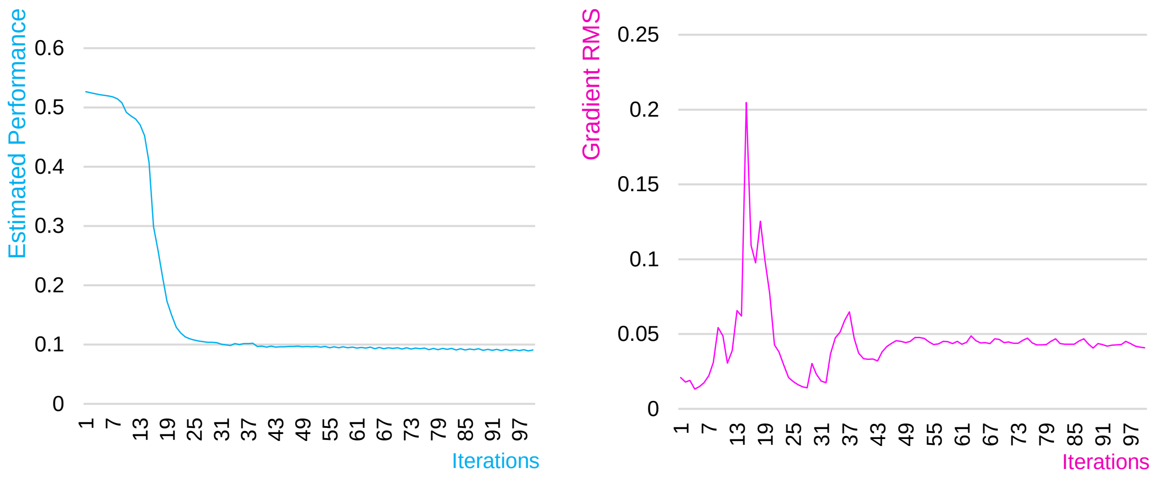
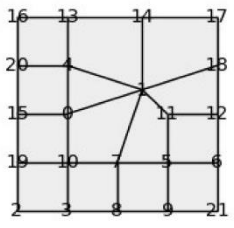


Chart 4.23 For an Invalid Case: Charts demonstrating the values of the performance score (on the left) and the root mean square of the gradients of the performance with respect to latent's space vector (on the right), during the optimization after 1000 iterations.

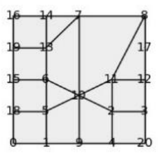
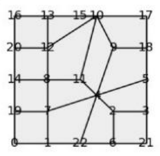
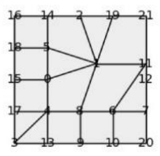
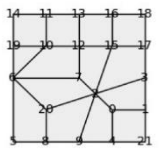
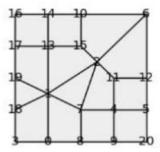
Case 2: Various Learning Rates

Some other results for the Case 2, with the Gradient Descent running with various learning rates are presented below. The more the learning rate decreases the more similar solutions to the initial design the GD outputs.

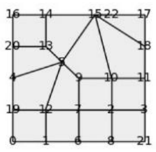
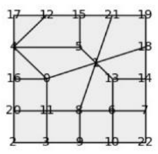
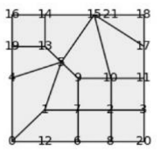
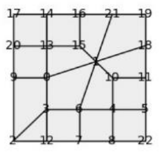
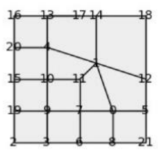


Initial Mesh

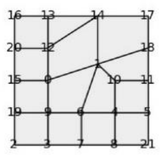
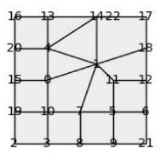
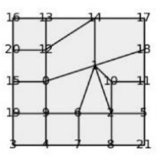
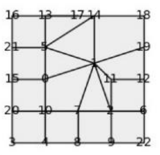
Learning Rate 5

					
Predicted Perfomance	0.18982741	0.17134124	0.14119774	0.10963255	0.18809715
Novelty	no	yes	yes	yes	no
Iterations	500	500	500	500	500

Learning Rate 2.5

					
Predicted Perfomance	0.16329785	0.17972642	0.1726144	0.19679457	0.21983096
Novelty	no	no	yes	no	no
Iterations	500	500	500	500	500

Learning Rate 0.5

				
Predicted Perfomance	0.14270523	0.12407419	0.14612302	0.1620233
Novelty	yes	yes	yes	yes
Iterations	500	500	500	500

4.4 Results

VAE

Using the flatten product of the adjacency matrix (3rd Option) seams to ease training. Training the VAE using with adjacency matrices resulted in a training loss of 2.82371 and a validation loss of 14.14076 after 500 epochs. In the case of the flattened simplified array the training loss was 2.308509827 and the validation loss 6.81249094. The model also was lighter. Increasing the number of layers may result in a slight overfitting as the validation loss seems to increase slowly (Chart 4.11). Increasing the dimensions of the latent space can also improve the training (Charts 4.4 and 4.5), resulting however in a slower optimization as more gradients have to be calculated in the end. In any case, VAE may generate invalid designs.

Surrogate Model

The performance of the dataset ranges from 0.12844538 to 1 with lower values representing better solutions. The Surrogate Model can predict with a good accuracy the performance of the shells. For the best designs that were excluded from the dataset (with a performance from 0 to 0.126232537) it doesn't predict with accuracy their performance but scores them around 0.15, which is close to the value that the model has learned to estimate as good. The Surrogate Model is not able to predict the performance of invalid designs that the VAE may produce.

Gradient Descent

The Gradient Descent managed to optimize designs and find novel solutions. The scores of those solutions are close to 0.15 but in reality they are smaller. Also, in Case 3 the Gradient descent with a learning rate of 2.5 managed to find the best performed sample that was excluded from the dataset. In general, smaller learning rates result in optimized designs more similar to the initial ones. Finally, the GD may result in invalid solutions as the VAE is very likely to generate invalid designs for which the Surrogate has not yet been trained to predict their performance.

5. APPLICATION

5 Application

The framework of this thesis focuses around the case of a shell structure with a quad based topology. The optimization process for this thesis aims to minimize the deflection, the utilization and the mass of the structure to prove that AI can help to optimize shell structures. In the future more criteria can fit in this framework .

The example below demonstrates an optimized shell structure in the case of a flat roof. A point load of 1 kN is taken into account at the middle, for repair access.

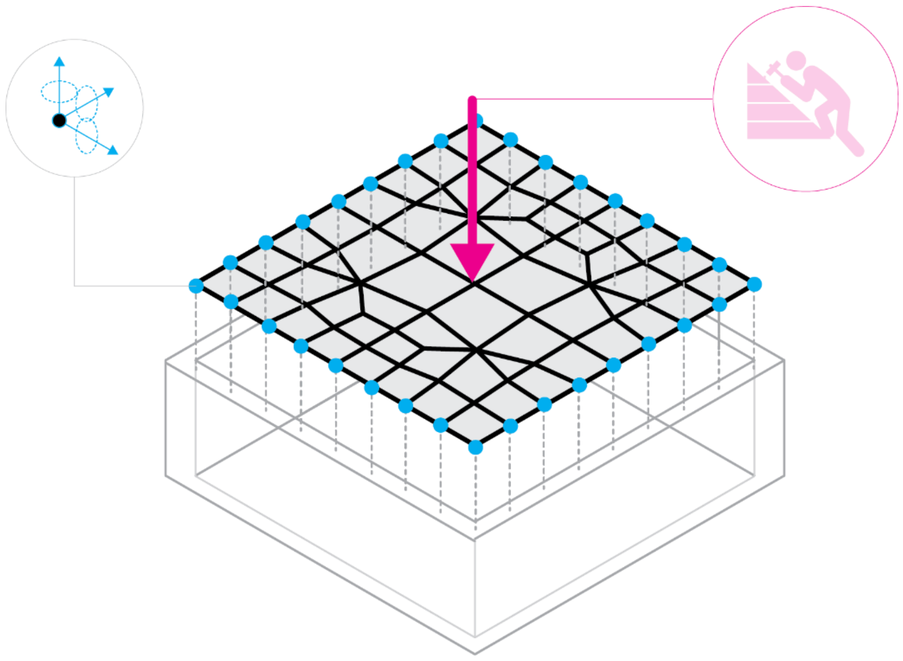


Figure 5.1 The study case

The images below correspond to the Case 2 in the Chapter 3.2 and demonstrate the initial design before and after relaxation. The starting performance score is 0**.17705911.**

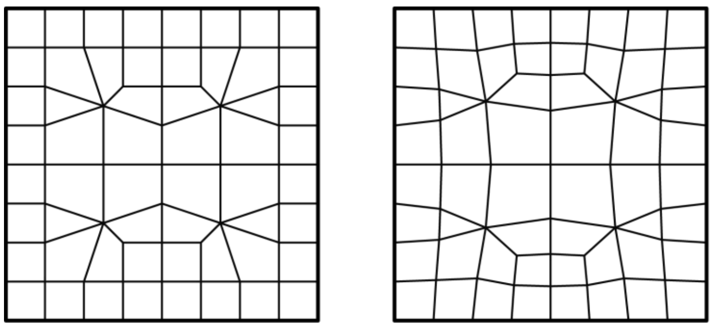


Figure 5.2 From left to the right: The initial design before and after relaxation.

The proposed AI workflow will attempt to optimize the topology of the Mesh first with a learning rate of 0.5. The optimized design has a performance score of **0.085932,** meaning that is the AI workflow managed to optimize it by 206%.

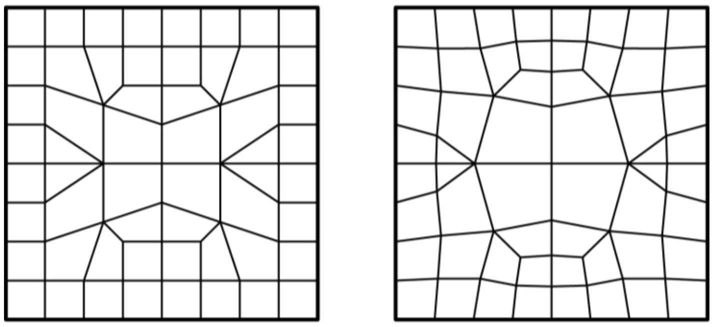


Figure 5.3 From left to the right: Theoptimized design using a learning rate of 0.5, before and after relaxation.

A second attempt to optimize the structure is made, using a larger learning rate of 2.5. In terms of its topology the result is more different compared to the initial design than the optimized case above. The optimized design has a performance score of **0.044936816,** meaning that this time the AI workflow managed to optimize it by 394%**.**

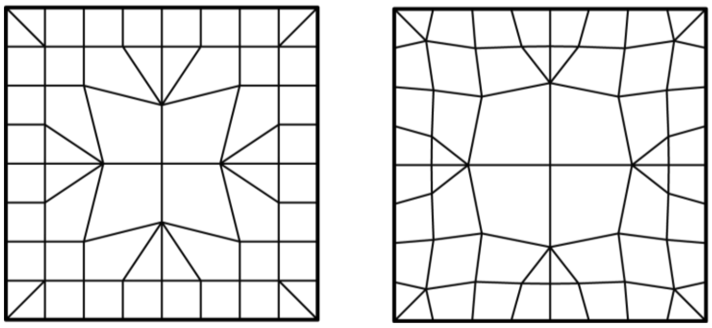
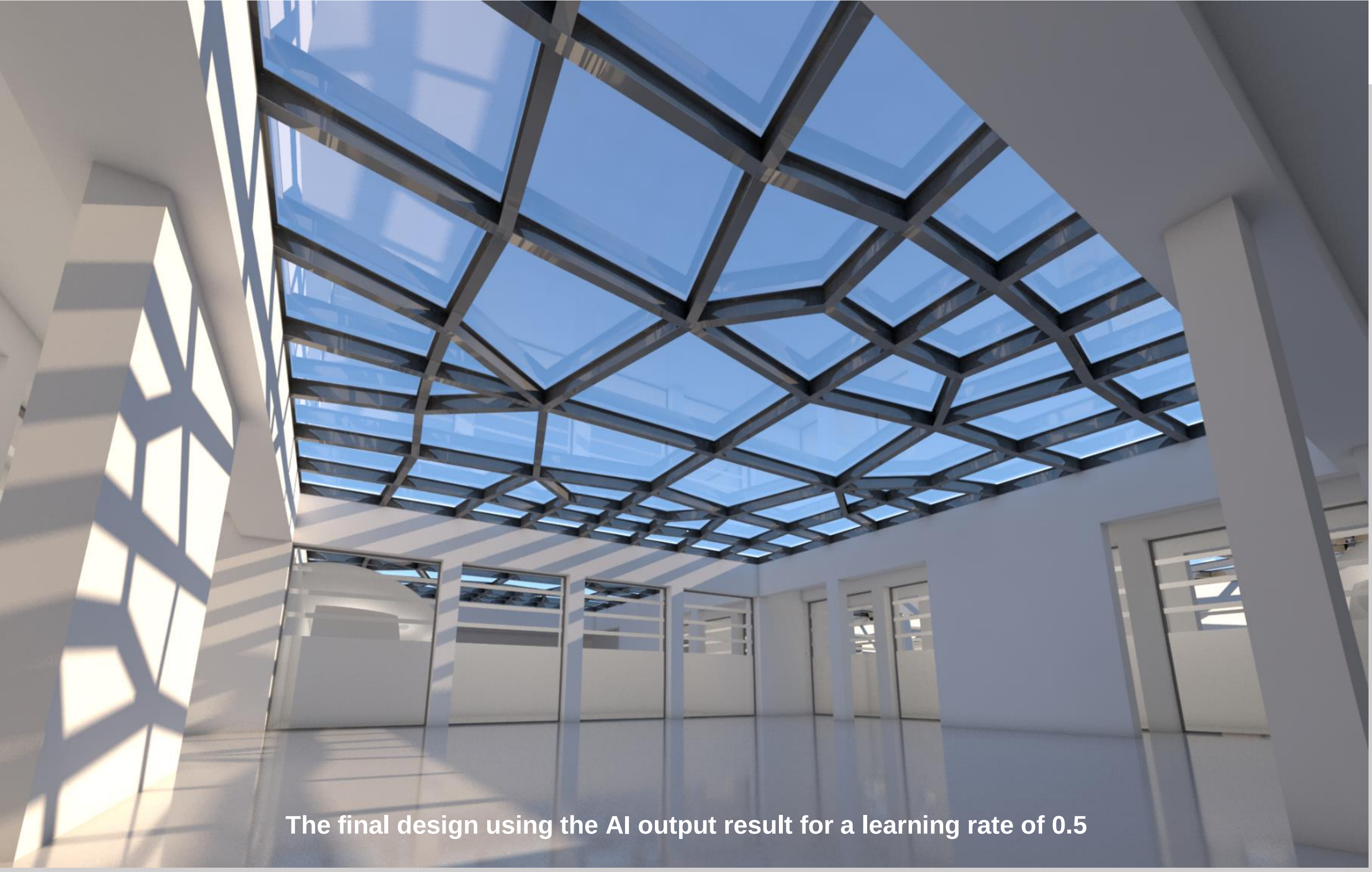
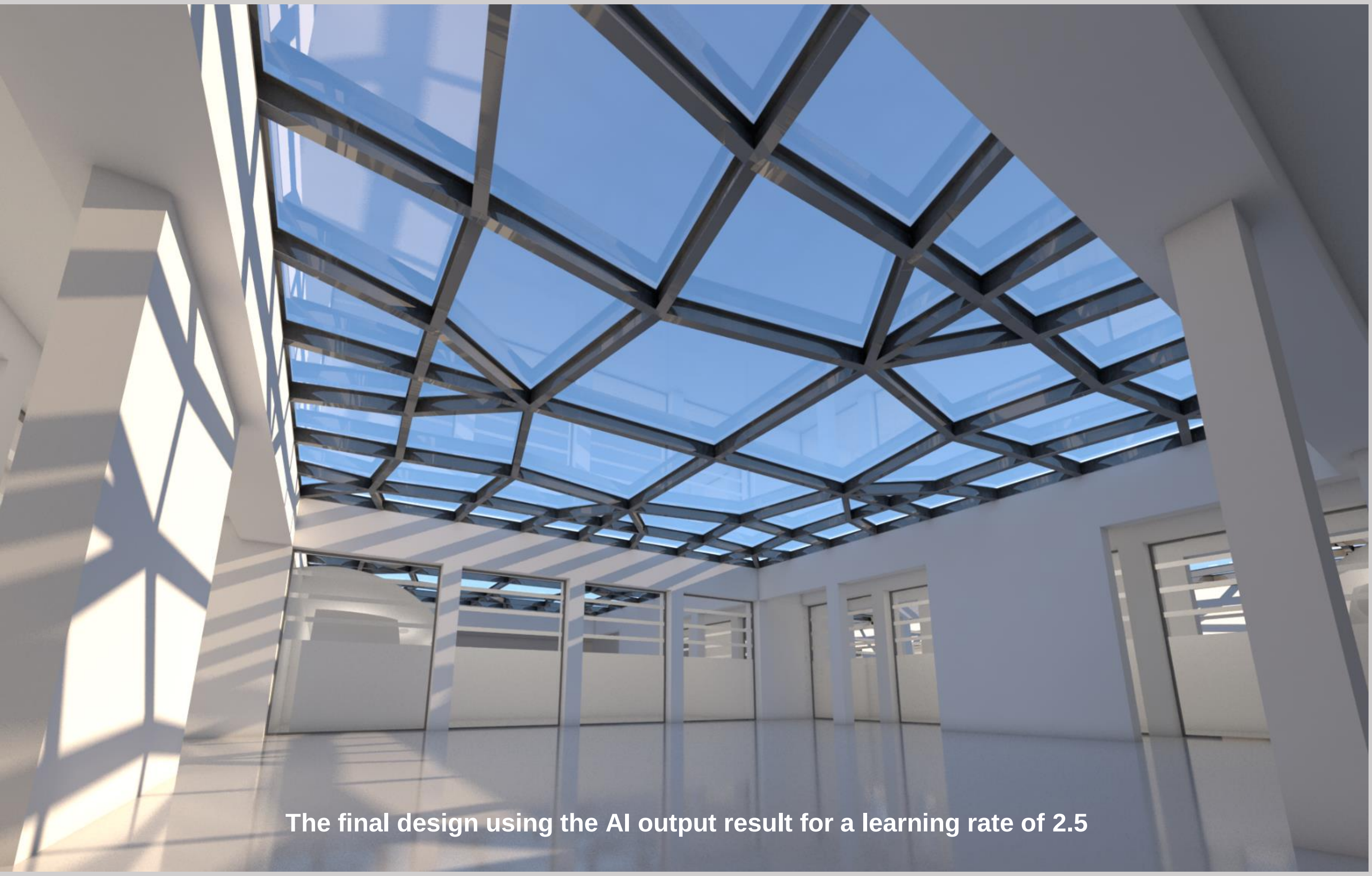


Figure 5.4 From left to the right: Theoptimized design using a learning rate of 2.5, before and after relaxation.

The designer- engineer can now choose the preferred solution.


The final design using the AI output result for a learning rate of 0.5


The final design using the AI output result for a learning rate of 2.5

In the future more criteria can fit in to the workflow. For instance a relaxation in the z axis, various boundaries and application other load cases.

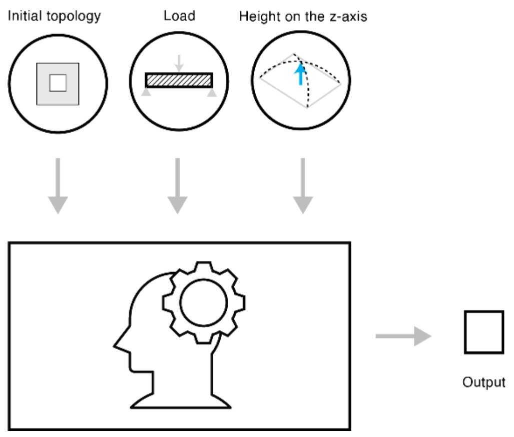


Figure 5.5 Future development of the AI workflow.

Some scenarios of shell structures with different boundaries and load cases, where AI could be used for topology optimization are demonstrated below.

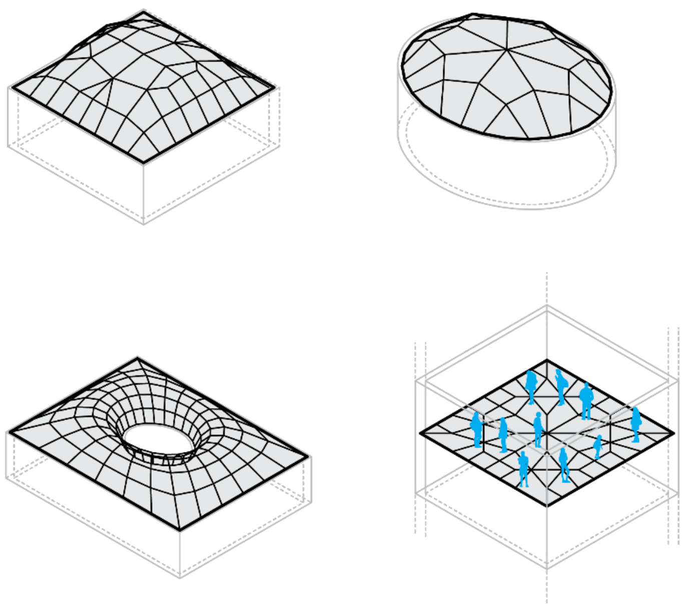


Figure 5.6 Various shell structures.

6. CONCLUSIONS

2.4 Research Questions

Main Question

Can an AI based framework generate new and structurally effective solutions?

Even though the Gradient Descent may produce invalid solutions, for some cases it was able to converge to structurally better designs than the provided dataset. There is room for further improvement in the AI workflow, however the research results indicate that an AI workflow can indeed expand the capabilities of Generative Design and reveal novel and structurally more effective solution.

Sub-questions

Can a Variational Autoencoder be trained to generate mesh tessellations?

Yes, The Variational Autoencoder can generate new mesh tessellations. It may also generate some invalid meshes. A dataset augmentation and an increase in the latent space can improve the training process, however invalid solutions still occur. The form of the dataset has also a big impact on the model's performance. Further explanation can be found at the next sub-question**.**

What form of data can be used to train a Variational Autoencoder to generate mesh tessellations?

Mesh tessellations can be described successfully using graph data structures. In this thesis adjacency matrices were used successfully. Using a denser mesh as a base to create adjacency matrices allows to train mesh data with different amount of faces and vertices. However, this method increases the size of the database.

To reduce the size of the dataset, a flattened and simplified product, resulting from the adjacency matrix, can be used.

Another option that was explored was to use arrays with the face's structure (Option 1 at the Variational Autoencoder). This dataset proved to be no appropriate for training VAEs to generate new meshes.

Can a surrogate model learn to predict the structural performance of decoded graph networks?

Yes, if the loss of the VAE is low it can.

Can a surrogate model learn to predict the structural performance of encoded graph networks?

No, the model in this case was overfitting. Increasing the size of the latent space did not improve the model's training

Can a Gradient Descent Optimizer propagate back to encoded data to search for optimum solutions?

Yes, The Gradient Descent was able to optimize mesh tessellations and discover novel and better solutions. However, in many cases invalid designs were produced. This is due to two main problems:

1)The VAE often generates invalid samples.

2) The surrogate model has not yet been trained to predict the performance of invalid tessellations

The result was that the loss of the Gradient Descent was converged in cases of invalid designs.

6.2 Limitations

- A dataset was created with a specific mesh boundary and limitations when it comes to pattern exploration.
- The only generative model that was used was a Variational Autoencoder.
- The optimization ran based on the structural performance. Other criteria such as similarity, singularity points, etc were excluded.
- For simplification reasons the dataset at some point was limited only to one quarter of the mesh**.**

6.3 Discussion and Future Development

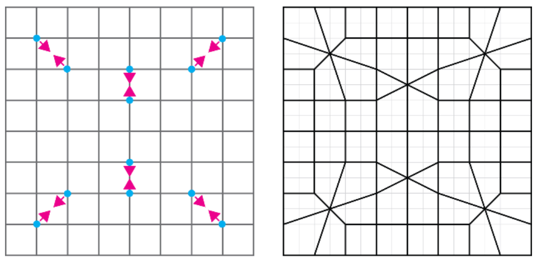
This section presents the discussion for the dataset creation, dataset labelling, the Variational Autoencoder and the Gradient Descent Optimization.

Dataset

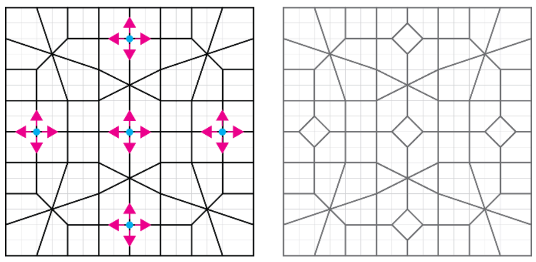
a. Dataset Generation

The dataset created for this thesis describes meshes with different number of faces and vertices. In order to create an appropriate dataset for training purposes, arrays of same shape are needed. A base mesh was used in that direction and was used as a base to create arrays that describe adjacency matrices. This strategy proved to be successful for training generative models.

The dataset was created from one original quadrilateral mesh using python libraries such as NumPy and Compas. It could be enriched with further pattern exploration (for instance creating new faces in random vertices).

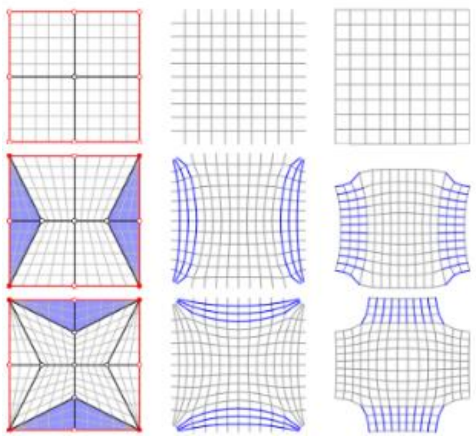


“Joining points” Method



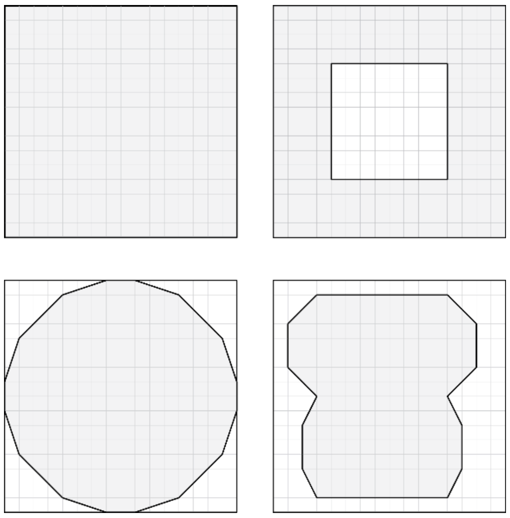
*“*Creating new faces in *random vertices”* Method

Another option would be to explore more configurations starting from various coarse meshes and with the strip method shown in the research of Robin Oval. (Oval et al., 2019)



The Method also explained in literature review suggests starting with a Coarse Mesh (bold black lines), Subdivide it with a edge target length and start adding stripes.

If the AI workflow proves successful then the dataset could be enriched with further mesh boundaries.

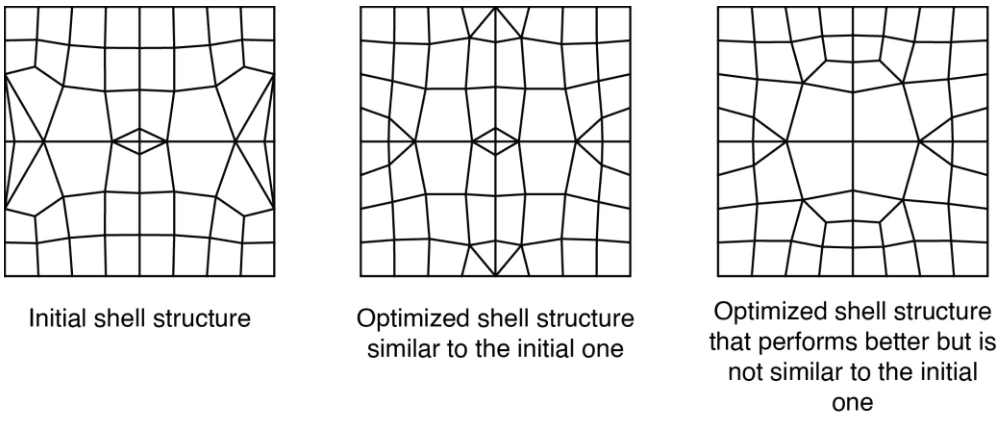


More boundary shapes can span into a denser grid.

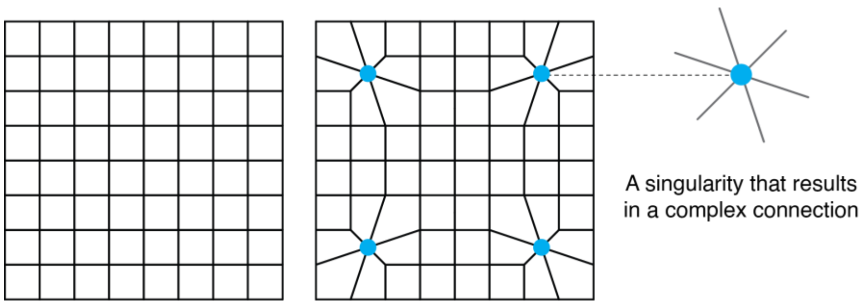
b. Labelling the dataset

The dataset was labelled with a performance score based on the mesh's maximum displacement, utilization and mass. For future development other criteria could be taken into account, like **different load cases, similarity, number of singularities, maximum length of edges**, etc.

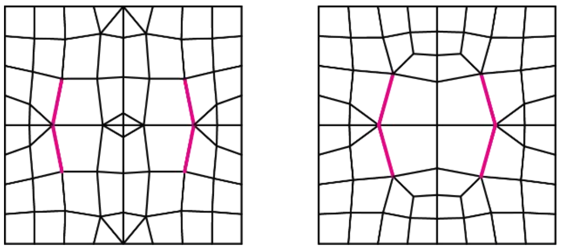
Similarity:


Initial shell structure
Optimized shell structure similar to the initial one
Optimized shell structure that performs better but is not similar to the initial one

Singularities:



Maximum Length:



The dataset could also be enriched with meshes that are relaxed after after applying loads on the z- axis.

VAE

The chosen architecture of the workflow's generative model was that of the VAE. The VAE was able to decode successfully the initial dataset as well as some of the best meshes that were excluded from training.

For future development more architectures worth to be explored. One option would be to include Graph Convolutional Layers in the model's architecture. These layers were used from Victor Basu to train a Graph VAE(Basu, 2022). Also according to bibliography GANs could be more effective for graph generation(Kensert, 2022).

Surrogate Model

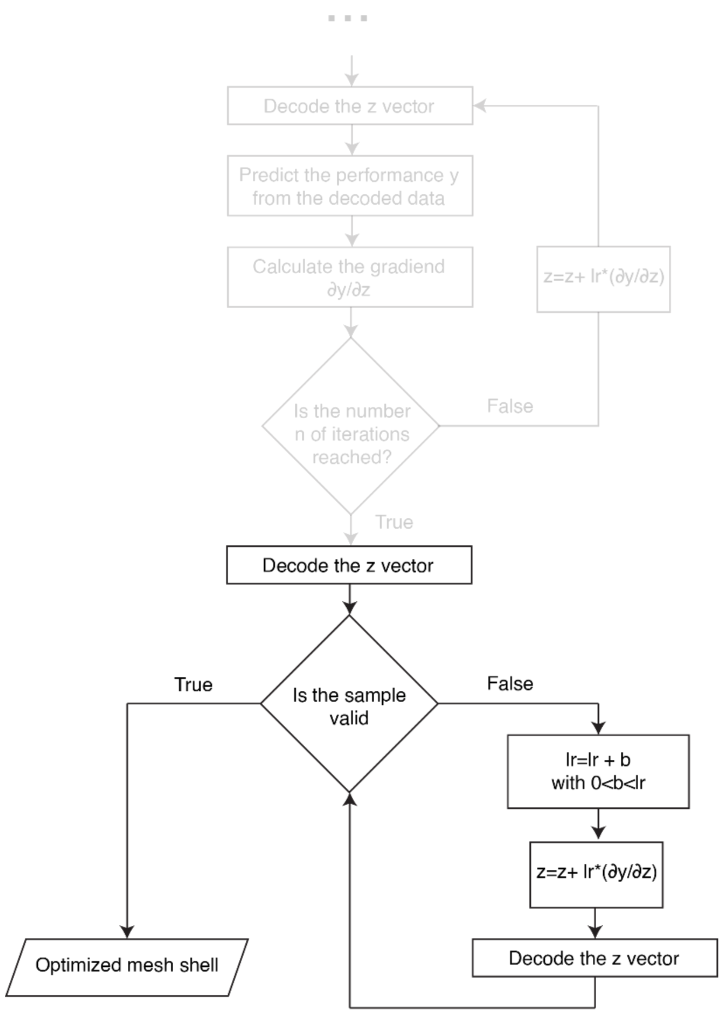
The Surrogate model was trained using decoded data. 50 samples were excluded from the training set to evaluate the models performance. The Surrogate Model was able to estimate their performance with a good accuracy. For designs performance scores lower than those in the dataset, it can't provide accurate estimates, but it does but score them close to the lowest labels in the dataset.

The model could be improved with decoded data that occur from a better trained Generative Model. A dataset augmentation with some invalid meshes that are scored negatively can also be tested.

Gradient Descent Optimization

The Gradient Descent Algorithm is able to retrieve the gradients from the VAE and the Surrogate Model and through back propagation to the encoded data search for the minima that corresponds to the performance score. The algorithm so far generates various invalid meshes. This is due to the fact that the surrogate model was not trained to predict the performance of the invalid data that the VAE generates, as explained in conclusions. Despite this the GD was able to generate novel solutions and better designs than the provided dataset.

Improving the VAE and the Surrogate Model will improve the performance of the GD. Another solution is to integrate to the workflow a model that can predict if the a design is valid of not. In case the GD converges to invalid solutions the learning rate can slightly start to increase until reaching a valid solution as the example below:


...
Decode the z vector
Predict the performance y from the decoded data
Calculate the gradiend ∂y/∂z
Is the number n of iterations reached?
False
z=z+ lr*(∂y/∂z)
True
Decode the z vector
Is the sample valid
True
False
lr=lr + b with 0<b<lr
z=z+ lr*(∂y/∂z)
Decode the z vector
Optimized mesh shell

7. REFERENCES

Basu, V. (2022). Implementing a Convolutional Variational AutoEncoder (VAE) for DrugDiscovery. Keras. https://keras.io/examples/generative/molecule_generation/

Bennett, J., Mahrous, K., Hamann, B., & Joy, K. I. (2003). Bicubic subdivision-surface wavelets for large-scale isosurface representation and visualization. Spring Conference on Computer Graphics, SCCG 2003 - Conference Proceedings, February, 9–16. https://doi.org/10.1145/984952.984954

Bradshaw, L. (n.d.). Big Data and what it Means. Us Chamber Foundation. Retrieved March 28, 2022, from https://www.uschamberfoundation.org/bhq/big-data-and-what-it-means

Brownlee, A. (2019). Overfitting and Underfitting With Machine Learning Algorithms. Machine Learning Mastery. https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/

Chen, L. (2014). Basic Data Structure Representing a Mesh Contents. UCI Department of Mathematics. https://www.math.uci.edu/~chenlong/iFEM/doc/html/meshbasicdoc.html

Chen, W., Zheng, X., Lei, N., & Luo, Z. (2018). Metric Based Quadrilateral Mesh Generation. November.

Chollet, F. (2016). Building Autoencoders in Keras. The Keras Blog. https://blog.keras.io/building-autoencoders-in-keras.html

Chollet, F. (2020). Keras. Variational AutoEncoder. https://keras.io/examples/generative/vae/

Daoud, M. (2020). Neurons, Activation Functions, Back-Propagation, Epoch, Gradient Descent: What are these? Medium. https://towardsdatascience.com/neurons-activation-functions-back-propagation-epoch-gradient-descent-what-are-these-c80349c6c452

Doersch, C. (2016). Tutorial on Variational Autoencoders. 1–23. http://arxiv.org/abs/1606.05908

Feed Forward Neural Network. (2019). DeePai. https://deepai.org/machine-learning-glossary-and-terms/feed-forward-neural-network

Fogg, H. J., Sun, L., Makem, J. E., Armstrong, C. G., & Robinson, T. T. (2018). Singularities in structured meshes and cross-fields. CAD Computer Aided Design, 105, 11–25. https://doi.org/10.1016/j.cad.2018.06.002

Ganesh, S. (2020). *What's The Role Of Weights And Bias In a Neural Network?* Medium. https://medium.com/p/4cf7e9888a0f

Gladstone, R. J., Nabian, M. A., Keshavarzzadeh, V., & Meidani, H. (2021). Robust Topology Optimization Using Variational Autoencoders. 1–20. http://arxiv.org/abs/2107.10661

GOYAL, C. (n.d.). Complete Guide to Gradient-Based Optimizers in Deep Learning. 2021. Retrieved March 21, 2022, from https://www.analyticsvidhya.com/blog/2021/06/complete-guide-to-gradient-based-optimizers/#:~:text=Gradient descent is an optimization,function to its local minimum.

Guo, S. (2020). An introduction to Surrogate modeling, Part I: fundamentals. Towards Data Science. https://towardsdatascience.com/an-introduction-to-surrogate-modeling-part-i-fundamentals-84697ce4d241

Kensert, A. (2022). WGAN-GP with R-GCN for the generation of small molecular graphs. Keras. https://keras.io/examples/generative/wgan-graphs/

Kingma, D. P., & Welling, M. (2014). Auto-encoding variational bayes. 2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings, Ml, 1–14.

Löffler, M., & Vaxman, A. (2016). Mesh Data Structures & Traversal. Utrecht University. https://www.enseignement.polytechnique.fr/informatique/INF562/Slides/MeshDataStructures.pdf

Lunot, V. (2019). *Vincent's Blog*. On the Use of the Kullback–Leibler Divergence in Variational Autoencoders. https://www.vincent-lunot.com/post/on-the-use-of-the-kullback-leibler-divergence-in-variational-autoencoders/

Manager, E. K. (2020). AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: *What's the Difference?* IBM Cloud. https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks

McKnight, M. (2017). Generative Design: What it is? How is it being used? Why it's a game changer. KnE Engineering, 2(2), 176. https://doi.org/10.18502/keg.v2i2.612

Millan, P. P., & Ochoa, D. (2020). Graph theory: adjacency matrices. EMBL's European Bioinformatics Institute. https://doi.org/DOI: 10.6019/TOL.Networks_t.2016.00001.1

Mitteroecker, P., Gunz, P., Windhager, S., & Schaefer, K. (2013). A brief review of shape, form, and allometry in geometric morphometrics, with applications to human facial morphology. Hystrix, 24(1). https://doi.org/10.4404/hystrix-24.1-6369

Oh, S., Jung, Y., Kim, S., Lee, I., & Kang, N. (2019). Deep generative design: Integration of topology optimization and generative models. Journal of Mechanical Design, Transactions of the ASME, 141(11). https://doi.org/10.1115/1.4044229

Oval, R., Rippmann, M., Mesnil, R., Mele, T. Van, Baverel, O., & Block, P. (2019). Automation in Construction Feature-based topology fi nding of patterns for shell structures. Automation in Construction, 103(May 2018), 185–201. https://doi.org/10.1016/j.autcon.2019.02.008

Papagiannopoulos, A., Clausen, P., & Avellan, F. (2021). How to teach neural networks to mesh: Application on 2-D simplicial contours. Neural Networks, 136, 152–179. https://doi.org/10.1016/j.neunet.2020.12.019

Ramsundar, B., & Zadeh, R. B. (2022). TensorFlow for Deep Learning. https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html

Rocca, J. (2019). Understanding Variational Autoencoders (VAEs). Towards Data Science. https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

Sauras-Altuzarra, L. (2022). Adjacency Matrix. Wolfram MathWorld. https://mathworld.wolfram.com/AdjacencyMatrix.html

Shafkat, I. (2018). Intuitively Understanding Variational Autoencoders. Towards Data Science. https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf

Sharma, S. (n.d.). Activation Functions in Neural Networks. Towards Data Science. Retrieved March 21, 2022, from https://medium.com/towards-data-science/activation-functions-neural-networks-1cbd9f8d91d6

Sharma, S. (2017). Epoch vs Batch Size vs Iterations. Towards Data Science. https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9

Shepherd, P., & Pearson, W. (2013). Topology Optimization of Algorithmically Generated Space Frames. Proceedings of the International Association for Shell and Spatial Structures (IASS) Symposium 2013, 1–7.

Soni, D. (2022). Supervised vs. Unsupervised Learning. Towards Data Science. Supervised vs. Unsupervised Learning

Stojiljković, M. (n.d.). Stochastic Gradient Descent Algorithm With Python and NumPy. Real Python. Retrieved March 28, 2022, from https://realpython.com/gradient-descent-algorithm-python/?fbclid=IwAR3-887K1sRhZZ2onwNdh2p8brseVQC7ZU0oIJMmYSpiPo7X_0cwo2ra2jw%0A%0A

Vinodhkumar, B. (2020a). Activation functions and its types. Medium. https://medium.com/@vinodhb95/activation-functions-and-its-types-8750f1287464

Vinodhkumar, B. (2020b). What is Loss in Neural Nets? Is cost function and loss function *are same ?* https://medium.com/@vinodhb95/what-is-loss-in-neural-nets-is-cost-function-and-loss-function-are-same-ef069a570e95

What are Neural Networks? (2020). IBM Cloud Education. https://www.ibm.com/cloud/learn/neural-networks#:~:text=Neural networks%2C also known as,neurons signal to one another.

Xu, K., Akram, M. N., & Chen, G. (2020). Semi-global Quad Mesh Structure Simplification via Separatrix Operations. SIGGRAPH Asia 2020 Technical Communications, SA 2020, December. https://doi.org/10.1145/3410700.3425436

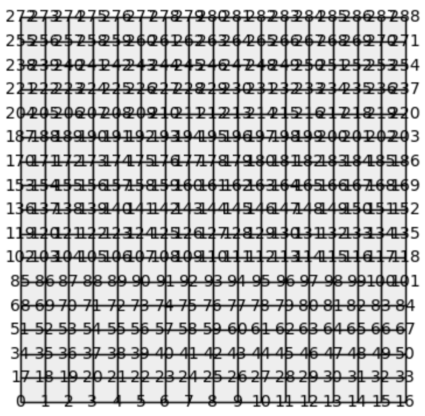
Young, N. & Foster + Partners. (n.d.). Robert and Arlene Kogod Courtyard [Photograph]. https://www.ggnltd.com/the-robert-and-arlene-kogod-courtyard

Gehry Partners, LLP and Frank O. Gehry, & Baan, I. (2014). Fondation Luis Vuitton [Photograph]. https://www.fondationlouisvuitton.fr/en/fondation

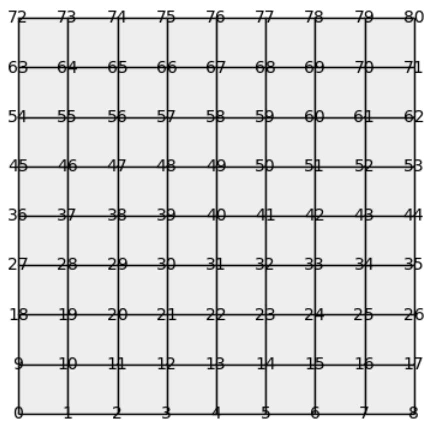
8. APPENDIX

DATASET GENERATION

```
import time
import numpy as np
from compas.datastructures import Mesh
import networkx
from networkx.algorithms.components.connected import connected_components
from compas import datastructures
from compas.datastructures import mesh_delete_duplicate_vertices
from compas_plotters.meshplotter import MeshPlotter
from compas.numerical import dr_numpy
from compas.geometry import matrix_from_axis_and_angle
from compas.datastructures import Mesh as CompasMesh
from compas.geometry import area_polygon
from random import seed
from random import randint
from random import choices
from stopit import threading_timeoutable as timeoutable
import os



#Open dense mesh
job_directory=os.getcwd()
data=os.path.join(job_directory, "dense_mesh")
meshdense = Mesh.from_obj(data)
plotter = MeshPlotter(meshdense, figsize=(4, 4))
plotter.draw_edges()
plotter.draw_vertices(text='key', radius=0.01)
plotter.draw_faces()
plotter.show()
```



```
#Open mesh to transform
data=os.path.join(job_directory, "bese_mesh")
mesh = Mesh.from_obj(data)
plotter = MeshPlotter(mesh, figsize=(4, 4))
plotter.draw_edges()
plotter.draw_vertices(text='key', radius=0.01)
plotter.draw_faces()
plotter.show()
```



```
#find the centre of xi points
def find_centre(mesh,xi):
    cxi=[]
    cyi=[]
    czi=[]
    for xa in xi:
        i=Mesh.vertex_coordinates(mesh, xa, axes='x')
        i=i[0]
        cxi.append(i)
        cx=((sum(set(cxi)))/(len(set(cxi))))

        i=Mesh.vertex_coordinates(mesh, xa, axes='y')
        i=i[0]
        cyi.append(i)
        cy=((sum(set(cyi)))/(len(set(cyi))))

        i=Mesh.vertex_coordinates(mesh, xa, axes='z')
        i=i[0]
        czi.append(i)
        cz=((sum(set(czi)))/(len(set(czi))))
        cnt=[cx,cy,cz]
    return cnt


#create a list of the faces structure with sublists of the coordinated of the
points
def face_coor(m,fkey):
    z=m.face_vertices(fkey)
    coor=[]
    for i in z:
        temp=Mesh.vertex_coordinates(m, i, axes='xyz')
        coor.append(temp)
    return coor

#list of coordinates for all vertices
def mesh_vertex_coordinates(m):
    vertices = list(m.vertices())
    vertices_coordinates=[]
    for i in vertices:
        i_coordinates=Mesh.vertex_coordinates(m, i, axes='xyz')
        vertices_coordinates.append(i_coordinates)
    return vertices_coordinates

#get the neighboors of the points
def neigh (m,r):
    faces = list(mesh.faces())
    faces_structure=[]
    for i in faces:
        faces_structure.append(m.face_vertices(i))
    neigh=[]
    for i in faces_structure:
        if r in i:
            for a in i:
                neigh.append(a)
    neigh = list(dict.fromkeys(neigh))
    neigh.remove(r)
    return (neigh)

#get the symmetrical points
def symmetrical_points (mesh,subdivition_number,subdivition_number2,random_point):
    num1=subdivition_number+1
    num2=subdivition_number2+1
    vertices = list(mesh.vertices())
    arr=np.array(vertices)
    arr=arr.reshape(num2, num1)
    arr_b=arr.transpose()
    temp=np.where(arr==random_point)
    xa=int(temp[0])
    xb=int(temp[1])
    mir=[]
    mir.append(arr[xa][xb])
    mir.append(arr[xa][-(xb+1)])
    mir.append(arr[-(xa+1)][(xb)])
    mir.append(arr[-(xa+1)][-(xb+1)])
    return mir

def to_graph(l):
    G = networkx.Graph()
    for part in l:
        # each sublist is a bunch of nodes
        G.add_nodes_from(part)
        # it also imlies a number of edges:
        G.add_edges_from(to_edges(part))
    return G

def to_edges(l):
    """
        treat `l` as a Graph and returns it's edges
        to_edges(['a','b','c','d']) -> [(a,b), (b,c),(c,d)]
    """
    it = iter(l)
    last = next(it)

    for current in it:
        yield last, current
        last = current
```

```
def flatten(t):
    flat_list = []
    for sublist in t:
        for item in sublist:
            flat_list.append(item)
    return flat_list

def checklists(a,b):#check if elements in list a are in b
    lst=[]
    for i in a:
        if i in b:
            lst.append(i)
    return (lst)

#Relax Mesh
@timeoutable()# stop relaxation if it takes too long
def relax(m):
    # extract the coordinates of vertices
    vertices, faces = m.to_vertices_and_faces()
    # extract edges
    edges = list(m.edges())
    # find the nboundary vertices
    boundary_vertices = m.vertices_on_boundary()
    # set loads
    loads = [[0, 0, 0]] * len(vertices)
    # Prescribed force densities in the edges
    qpre = [2] * len(edges)
    # kN/m^3 # TODO: double check the unit
    xyz, q, f, l, r = dr_numpy(vertices, edges, boundary_vertices, loads, qpre)
    relaxed_mesh = CompasMesh.from_vertices_and_faces(xyz, faces)
    return relaxed_mesh

def delete_indices(lst,del_list):
    t=0
    for i in del_list:
        z= i-t
        lst.pop(z)
        t=t+1
    return lst

#select vertices in 1/4 of the mesh
vertices = list(mesh.vertices())
temp=x_sub+1
temp=int(temp)
tempb=x_sub/2
tempb=int(tempb)
v=[]

for i in range(tempb+1):
    v.append(vertices[i])
a=v
t=[i+(temp) for i in a]
v=v+t
for i in range (tempb-1):
    t=[i+(temp) for i in t]
    v=v+t
vertices=v
print (vertices)

#this is the transformation def
def transformmesh(mesh,random_point,random_select,x_sub,y_sub):
    boundary_vertices = mesh.vertices_on_boundary()
    corner_points=[72,80,0,8]
    #find symmetries
    p_lst=[]
    for i in random_point:
        p_lst.append(neigh(mesh,i))
    r_lst=[]
    for i in range(len(random_point)):
        temp=p_lst[i]
        indx=random_select[i]
        tempb=temp[indx]
        r_lst.append(tempb)
    syma=[]
    for i in r_lst:
        a= symmetrical_points(mesh,x_sub,y_sub,i)
        syma.append(a)
    symb=[]
    for i in random_point:
        a= symmetrical_points(mesh,x_sub,y_sub,i)
        symb.append(a)
    syma=flatten(syma)
    symb=flatten(symb)
    temp_lst=[]
    for i in range (len(syma)):
        l=[syma[i],symb[i]]
        temp_lst.append(l)
    G = to_graph(temp_lst)
    lst_new=list(connected_components(G))
    pts_to_merge=[]

    for i in range(len(lst_new)):
        temp=lst_new[i]
        t=list(temp)
        pts_to_merge.append(t)

    print (pts_to_merge)

    #Resulting point after merGing selected points
    merged_points=[]
    for i in pts_to_merge:
        tempb=checklists(i,corner_points)
        if len(tempb)== 0:
            temp=checklists(i,boundary_vertices)
            if len(temp)==1:#if there is one boundary point in the points_to_merge
select it as the resulting point
                tempa=Mesh.vertex_coordinates(mesh, temp[0], axes='xyz')
                merged_points.append(tempa)
            elif len(temp)>1:#if there is more than one boundary point in the
points_to_merge, select their centre as the resulting point
                merged_points.append(find_centre(mesh,temp))
            else:#if there is no boundary point in the points_to_merge, select
their centre as the resulting point
                merged_points.append(find_centre(mesh,i))
        else:#if there is a corner point in the points_to_merge, select it as the
resulting point
            tempc=checklists(i,corner_points)
            tempc=Mesh.vertex_coordinates(mesh, tempc[0], axes='xyz')
            merged_points.append(tempc)

    #Indexes for vertices, edges and faces
    vertices = list(mesh.vertices())
    edges = list(mesh.edges())
    faces = list(mesh.faces())

    #list with face structure
    faces_structure=[]
    for f in faces:
        faces_structure.append(mesh.face_vertices(f))

    #list of coordinates for all vertices
    vertices = list(mesh.vertices())
    vertices_coordinates=[]
    for i in vertices:
        i_coordinates=Mesh.vertex_coordinates(mesh, i, axes='xyz')
        vertices_coordinates.append(i_coordinates)
    x=-1
    for i in pts_to_merge:
        x=x+1
        for e in i:
            vertices_coordinates[e]=merged_points[x]

    #new mesh
    mesh2 = Mesh.from_vertices_and_faces(vertices_coordinates, faces_structure)
    training_vertices = vertices_coordinates
    #dictionaries of vertices&faces
    vertices_dict=dict(zip(vertices, vertices_coordinates))
    faces_dict=dict(zip(faces,faces_structure))

    #clear faces with zero area
    faces = list(mesh2.faces())
    faces_structure=[]
    for i in faces:
        faces_structure.append(mesh2.face_vertices(i))
    vertices_coordinates=mesh_vertex_coordinates(mesh2)
    i=0
    l_temp=[]

    #get a list with the indexes of faces with zero area
    for i in range(len(faces)):
        t= (face_coor(mesh2,i))
        a=area_polygon(t)
        if abs(a)<0.001:
```

```
        #del faces_structure[i]
        l_temp.append(i)
    i=i+1
temp=0

for i in l_temp:
    del faces_structure[i-temp]
    temp=temp+1

for i in l_temp:
    del faces_dict[i]

#remove no used vertices
faces_structure_flatten=flatten(faces_structure)
temp=[]
for i in vertices:
    if i in faces_structure_flatten:
        pass
    else:
        temp.append(i)
for i in temp:
    del vertices_dict[i]

#new clear mesh
mesh3 = Mesh.from_vertices_and_faces(vertices_dict, faces_dict)
mesh_delete_duplicate_vertices(mesh3, precision=None)

#fix the index of vertices
vertices = list(mesh3.vertices())
vertices_coordinates=[]
for i in vertices:
    i_coordinates=Mesh.vertex_coordinates(mesh3, i, axes='xyz')
    vertices_coordinates.append(i_coordinates)

faces = list(mesh3.faces())
faces_structure=[]
for f in faces:
    faces_structure.append(mesh3.face_vertices(f))

i=0
vertices_new=[]
for v in vertices:
    vertices_new.append(i)
    i=i+1
lstold=[]
lstnew=[]
i=0
for v in vertices:
    if vertices[i] != vertices_new[i]:
        lstold.append(v)
        lstnew.append(vertices_new[i])
    i=i+1
res = {lstold[i]: lstnew[i] for i in range(len(lstold))}
e=0
faces_structure_new= faces_structure
```

```
    for f in faces_structure_new:
        z=0
        for i in f:
            if i in lstold:
                faces_structure_new[e][z]=res[i]
            z=z+1
        e=e+1
    mesh4 = Mesh.from_vertices_and_faces(vertices_coordinates,
faces_structure_new)


    return (mesh4,pts_to_merge)


def get_adjmatrix(mesh4,meshdense):
    vertices_old = list(meshdense.vertices())
    v_old=[]
    for i in vertices_old:
        i_coordinates_old=Mesh.vertex_coordinates(meshdense, i, axes='xyz')
        v_old.append(i_coordinates_old)

    vertices_new = list(mesh4.vertices())
    v_new=[]
    for i in vertices_new:
        i_coordinates_new=Mesh.vertex_coordinates(mesh4, i, axes='xyz')
        v_new.append(i_coordinates_new)

    #Create empty matrix
    adjmatrix=np.zeros((289, 289))

    faces_new = list(mesh4.faces())
    faces_structure_new=[]
    for f in faces_new:
        faces_structure_new.append(mesh4.face_vertices(f))

    #Create a dictionary --> (vertex index in new mesh): (vertex index in dense
mesh)
    dict_temp= {}
    for i in range(len(v_old)):
        if v_old[i] in v_new:
            temp={(v_new.index(v_old[i])):i}
            dict_temp.update(temp)

    #Replace the vertex index in faces structure of the dense mesh
    z=0
    for i in faces_structure_new:
        for y in range(len(i)):

            (faces_structure_new[z])[y]=dict_temp[(faces_structure_new[z])[y]]
        z=z+1

    #create adjacency matrix
    for i in faces_structure_new:
        if len(i)==4:
            adjmatrix[(i[0]),(i[1])]=1.0
            adjmatrix[(i[1]),(i[0])]=1.0
            adjmatrix[(i[1]),(i[2])]=1.0
            adjmatrix[(i[2]),(i[1])]=1.0
            adjmatrix[(i[2]),(i[3])]=1.0
            adjmatrix[(i[3]),(i[2])]=1.0
            adjmatrix[(i[3]),(i[0])]=1.0
            adjmatrix[(i[0]),(i[3])]=1.0
        if len(i)==3:
            adjmatrix[(i[0]),(i[1])]=1.0
            adjmatrix[(i[1]),(i[0])]=1.0
            adjmatrix[(i[1]),(i[2])]=1.0
            adjmatrix[(i[2]),(i[1])]=1.0
            adjmatrix[(i[2]),(i[0])]=1.0
            adjmatrix[(i[0]),(i[2])]=1.0

    #relaxed_mesh=relax(timeout = 2,m=mesh4)

    return (adjmatrix)


#dataset creation for option 2: Using an adjacency matrix as training data

for i in range(11000): #this number is not final because the generation will
produce double results
    try:
        ram_range_ver = randint(1, range_ver)
        random_point=choices(vertices, k=ram_range_ver)
        len(random_point)
        random_select = []
        for i in range(len(random_point)):
            temp=neigh(mesh,random_point[i])
            temp=len(temp)-1
            n = randint(0,temp)
            random_select.append(n)
        transform=transformmesh(mesh,random_point,random_select,x_sub,y_sub)
        tmesh=transform[0]
        pointstomerge=transform[1]
        area=Mesh.area(tmesh)
        relaxed_mesh=relax(timeout = 2,m=tmesh)
        mtx=get_adjmatrix(tmesh,meshdense)
        mtx2=mtx.reshape(1,289,289)
        if pointstomerge in pointstomerge_list:
            doublemesh=doublemesh+1
        if area==225 and pointstomerge not in pointstomerge_list :
            dataset.append(mtx)
            vertices_and_faces=relaxed_mesh.to_vertices_and_faces()
            dir="C:/Users/31613/Documents/2021/Graduation/P4/Trainning_data/FEM/"
            str_interations=str(interations).zfill(4)
            dir_vert=dir+str_interations+"_vertices.txt"
            f =open(dir_vert, "w")
            for i in vertices_and_faces[0]:
                r=str(i)
                r=r.replace('[', '{')
                r=r.replace(']', '}')
                f.write(r+"/")
            f.close()

            dir_faces=dir+str_interations+"_faces.txt"
            f =open(dir_faces, "w")
            for i in vertices_and_faces[1]:
                w=str(i)
                w=w.replace('[', '{')
                w=w.replace(']', '}')
                f.write(w+"/")
            f.close()
            interations=interations+1
            pointstomerge_list.append(pointstomerge)
            print("iteration",interations)

    except IndexError:
        print ("error")
    except AttributeError:
        print ("error")
    except ValueError:
        print ("error")

#dataset generation for option 1: Using the structure of vertices as training data
for i in range(2000):
    try:
        print("iteration",interations)
        ram_range_ver = randint(1, range_ver)
        random_point=choices(vertices, k=ram_range_ver)
        len(random_point)
        random_select = []
        for i in range(len(random_point)):
            temp=neigh(mesh,random_point[i])
            temp=len(temp)-1
            n = randint(0,temp)
            #n = randint(0,7)
            random_select.append(n)
        print (("random points"),random_point)
        print (("random select"),random_select)
        w=transformmesh(mesh,random_point,random_select,x_sub,y_sub)
        rmesh=w[0]
        vertices = list(rmesh.vertices())
        #plotter = MeshPlotter(rmesh, figsize=(2, 2))
        #plotter.draw_edges()
        #plotter.draw_vertices(text='key', radius=0.01)
        #plotter.draw_faces()
        #plotter.show()
        dataset.append(w[1])

        #export mesh data to folder
        vertices_and_faces=rmesh.to_vertices_and_faces()
        #dir="C:/Users/31613/Documents/2021/Graduation/P3/trainingdata_vs/"
        #str_interations=str(interations).zfill(4)
        #dir_vert=dir+str_interations+"_vertices.txt"
        #f =open(dir_vert, "w")
        #for i in vertices_and_faces[0]:
        #   r=str(i)
        #   r=r.replace('[', '{')
        #   r=r.replace(']', '}')
        #   f.write(r+"/")
        #f.close()

        #dir_faces=dir+str_interations+"_faces.txt"
        #f =open(dir_faces, "w")
        #for i in vertices_and_faces[1]:
        #   w=str(i)
        #   w=w.replace('[', '{')
        #   w=w.replace(']', '}')
        #   f.write(w+"/")
        #f.close()

        interations=interations+1

    except IndexError:
        print ("error")
    except AttributeError:
        print ("error")
    except ValueError:
        print ("error")


#Saving the dataset
dataset_array=np.array(dataset)
file="C:/Users/31613/Documents/2021/Graduation/P4/TrainingData/AdjacencyMatrix"
np.save(job_directory, dataset_array, allow_pickle=True, fix_imports=True)
```

DATASET AUGMENTATION

```
# #dictionary that flips the quarter_mesh
row1=[0,1,2,3,4,5,6,7,8,17,26,35,44,53,62,71,80,79,78,77,76,75,74,73,72,63,54,45,3
6,27,18,9,0]
row2=[10,11,12,13,14,15,16,25,34,43,52,61,70,69,68,67,66,65,64,55,46,37,28,19,10]
row3=[20,21,22,23,24,33,42,51,60,59,58,57,56,47,38,29,20]
row4=[30,31,32,41,50,49,48,39,30]
row5=[40]
row1_reverse=reversed(row1)
row2_reverse=reversed(row2)
row3_reverse=reversed(row3)
row4_reverse=reversed(row4)
dict1=dict(zip(row1,row1_reverse))
dict2=dict(zip(row2,row2_reverse))
dict3=dict(zip(row3,row3_reverse))
dict4=dict(zip(row4,row4_reverse))
dict5=dict(zip(row5,row5))
dict_flip = dict1.copy()
dict_flip.update(dict2)
dict_flip.update(dict3)
dict_flip.update(dict4)
dict_flip.update(dict5)

#flip the mesh
def flip_matrix(array,dict_flip,shape):
    array_flipped=np.zeros((shape,shape))
    for i in range(81):
        for e in range(81):
            array_flipped[i,e]=array[dict_flip.get(i),dict_flip.get(e)]
    return array_flipped
```

FLATTEN AND SIMPLIFY ADJACENCY MATRICES

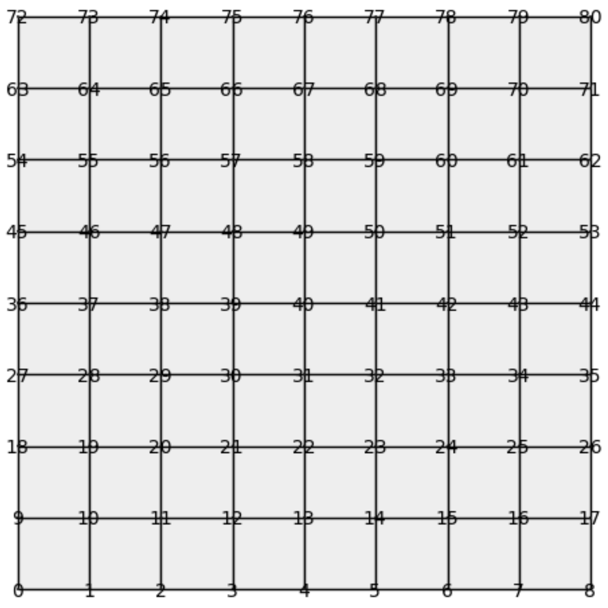
```
#Create the flattemed and simplified arrays
def create_flatten_lst(array):
    flatten_lst=[]
    for i in range(array.shape[1]):
        for e in range(i):
            flatten_lst.append(array[i,e])
    return flatten_lst
#Construct adjacency matrices from flattened and simplified arrays
def flatten_to_matrix(flatten_lst,shape1):
    array_zero=np.zeros((shape1,shape1))
    y=0
    for i in range(shape1):
        for e in range(i):
            array_zero[i,e]=flatten_lst[y]
            array_zero[e,i]=flatten_lst[y]
            y=y+1
    array_new=array_zero
    return array_new
```

AI FRAMEWORK

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
from IPython import display
from sklearn.model_selection import train_test_split
import glob
import imageio
from compas.datastructures import Mesh
from compas_plotters.meshplotter import MeshPlotter
import networkx
from networkx.algorithms.components.connected import connected_components
from keras import layers, activations
import os
import pandas as pd
import tensorflow as tf
```

#Loading data

```
#Open the dense mesh that will be used as a base for the adjacency matrix
job_directory=os.getcwd()
data= os.path.join(job_directory, "Trainning_data","dense_meshquarter.obj")
meshdense = Mesh.from_obj(data)
plotter = MeshPlotter(meshdense, figsize=(5, 5))
plotter.draw_edges()
plotter.draw_vertices(text='key', radius=0.01)
plotter.draw_faces()
plotter.show()
```



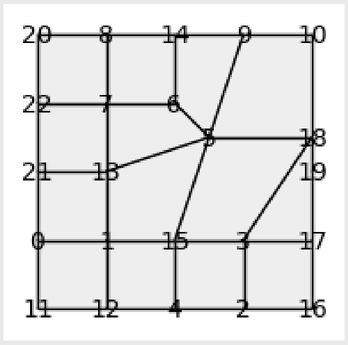
```
# Clear some bad meshes
remove=[437,457,1123,1142,1307,1377,1772,1775,1781,1921,2050,2312,2364,2394,2439,2
639,2709,2717,2734,2754,2968,3128,3196,3283,3309,3359,3522,3524,3579,3597,3604,364
7,3669,3731,3759,3817,3839,4144,4203,4337,4391,4585,4591,4594,4610,4633,4994,5084,
5166,5390,5468,5661]
dataset=np.delete(a, remove, axis=0)
```

```
# Remove best performed meshes from the training dataset
best=[2102,1142,1721,4270,4884,705,3501,562,1367,5277]
dataset_new= np.delete(dataset, best, axis=0)

# Select 1/4 of the adjacency matrices
lst_temp=
[*range(0,9),*range(17,26),*range(34,43),*range(51,60),*range(68,77),*range(85,94)
,*range(102,111),*range(119,128),*range(136,145)]

dataset_quarter=np.take(dataset_new,lst_temp, axis=1)
dataset_quarter=np.take(dataset_quarter,lst_temp, axis=2)
dataset_best=np.take(dataset,best, axis=0)
dataset_quarter_best=np.take(dataset_best,lst_temp, axis=1)
dataset_quarter_best=np.take(dataset_quarter_best,lst_temp, axis=2)

# Get a mesh from an adjacency matrix
def Mesh_from_mtx(mtx):
    vertices_old = list(meshdense.vertices())
    v_old=[]
    for i in vertices_old:
        i_coordinates_old=Mesh.vertex_coordinates(meshdense, i, axes='xyz')
        v_old.append(i_coordinates_old)

    #get the connected edges
    D=networkx.DiGraph(mtx)
    edges = [[u, v] for [u, v] in D.edges()]

    #sort the tuples of edges
    temp=[]
    for i in edges:
        temp.append(tuple(sorted(i)))

    #delete duplicate edges
    temp=set(temp) #first create set
    temp=tuple(temp) #convert set to tuple

    #Convert to list
    edges=[]
    for i in temp:
        z=[]
        for y in i:
            z.append(v_old[y])
        edges.append(z)

    mrebuild=Mesh.from_lines(edges)
    return (mrebuild)
```

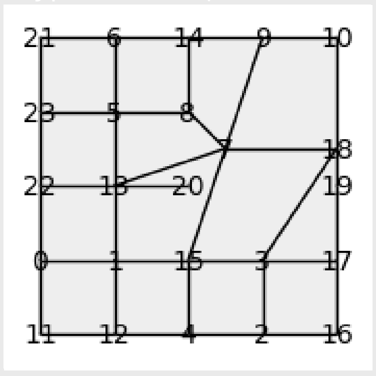
#VAE

```
#Sampling Layer
class Sampling(layers.Layer):
    """Uses (z_mean, z_log_var) to sample z, the vector encoding a digit."""

    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

#Encoder
latent_dim = 10

encoder_inputs = keras.Input(shape=(3240))
x = layers.Dense(1000, activation="relu")(encoder_inputs)
x = layers.Dense(100, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z], name="encoder")
encoder.summary()

#Decoder

latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(100, activation="relu")(latent_inputs)
x = layers.Dense(1000, activation="relu")(x)
decoder_outputs= layers.Dense(3240, activation="sigmoid")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
decoder.summary()

#VAE customize
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def train_step(self, data):
        if isinstance(data, tuple):
            data = data[0]
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            #reconstruction =tf.round(self.decoder(z))
            reconstruction = self.decoder(z)
            reconstruction_loss = tf.reduce_mean(
                keras.losses.mean_squared_error(data, reconstruction)
            )
            reconstruction_loss *= 100 * 100

            kl_loss = 1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)
            kl_loss = tf.reduce_mean(kl_loss)
            kl_loss *= -0.5

            total_loss = reconstruction_loss + kl_loss
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))

        return {
            "loss": total_loss,
            "reconstruction_loss": reconstruction_loss,
            "kl_loss": kl_loss,
        }

    def test_step(self, data):
      if isinstance(data, tuple):
        data = data[0]

      z_mean, z_log_var, z = self.encoder(data)
      reconstruction = self.decoder(z)
      reconstruction_loss = tf.reduce_mean(
            keras.losses.mean_squared_error(data, reconstruction)
      )
      reconstruction_loss *= 100 * 100

      kl_loss = 1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)
      kl_loss = tf.reduce_mean(kl_loss)
      kl_loss *= -0.5
      total_loss = reconstruction_loss + kl_loss

      return {
          "loss": total_loss,
          "reconstruction_loss": reconstruction_loss,
          "kl_loss": kl_loss,
      }

#dataset
x_train = dataset_quarter.astype('float32')

#train
vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam(),metrics=['accuracy'])
history = vae.fit(x_train, epochs=500, batch_size=64,validation_split=0.2)


Check how the data are decoded
test_mesh=dataset_quarter_best[1]
testmesh= Mesh_from_mtx(test_mesh)
plotter = MeshPlotter(testmesh, figsize=(2, 2))
plotter.draw_edges()
plotter.draw_vertices(text='key', radius=0.01)
plotter.draw_faces()
plotter.show()
test_mesh= test_mesh.astype('float32')
test_mesh=test_mesh.reshape((1,81,81,1))
#tensor = tf.convert_to_tensor(test_mesh)
x= vae.encoder(test_mesh)
```

```
mean=x[0]
logvar=x[1]
z=x[2]
print(z)
decodedtensor=vae.decoder(z)
decodesarray=decodedtensor.numpy()
mtx=decodesarray.reshape(81,81)
array =np.zeros((81,81))
thr=0.4
for i in range(mtx.shape[0]):
    for y in range(mtx.shape[1]):
        if mtx[i,y]>thr:
            array[i,y]=1

testmesh= Mesh_from_mtx(array)
plotter = MeshPlotter(testmesh, figsize=(2, 2))
plotter.draw_edges()
plotter.draw_vertices(text='key', radius=0.01)
plotter.draw_faces()
plotter.show()
```

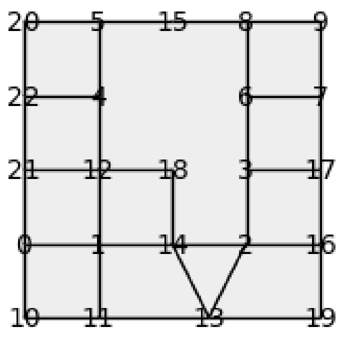


```
tf.Tensor( [[-0.8392136 -0.01628723 1.4019599 0.27650622 -2.1292992
0.33380827 0.18326117 1.1488624 -1.7288721 -0.04750239]], shape=(1, 10),
dtype=float32)
```



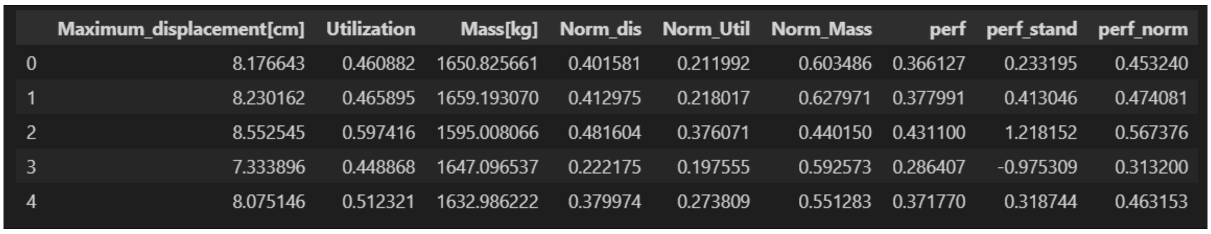
```
#Generate a mesh randomly from latent space
tf_random=tf.random.uniform(shape
    =[1, 10],
    minval=-1,
    maxval=1,
    dtype=tf.dtypes.float32,
    seed=None,
    name=None
)
```

```
decodedtensor=vae.decoder(tf_random)
decodedtensor=tf.round(decodedtensor)
print (decodedtensor.shape)
decodesarray=decodedtensor.numpy()
mtx=decodesarray.reshape(81,81)
testmesh= Mesh_from_mtx(mtx)
plotter = MeshPlotter(testmesh, figsize=(2, 2))
plotter.draw_edges()
plotter.draw_vertices(text='key', radius=0.01)
plotter.draw_faces()
plotter.show()
```



```
#Get the loss history
history.history.keys()
history.history['loss']


#Save VAE
Save_directory= os.getcwd()
vae.get_layer('encoder').save_weights(Save_directory + "encoder_weights.h5")
vae.get_layer('decoder').save_weights(Save_directory +"decoder_weights.h5")
vae.get_layer('encoder').save(Save_directory + "encoder_arch")
vae.get_layer('decoder').save(Save_directory +"decoder_arch")

#Load VAE
Load_directory= os.getcwd()
encoder= keras.models.load_model(Load_directory + "encoder_arch")
decoder= keras.models.load_model(Load_directory +"decoder_arch
vae = VAE(encoder, decoder) #You need to have VAE class defined for this to works
vae.get_layer('encoder').load_weights(location + "encoder_weights.h5")
vae.get_layer('decoder').load_weights(location +"decoder_weights.h5")
vae.compile(optimizer=keras.optimizers.Adam())
```

```
#Surrogate
location_labels= os.path.join(job_directory,LABELS.csv")
column_names=["Maximum_displacement[cm]","Utilization","Mass[kg]","Norm_dis","Norm
_Util","Norm_Mass","perf","perf_stand","perf_norm"]
labels_temp = pd.read_csv(location_labels, names=column_names)
labels_temp.drop(labels=None, axis=None, index=best, columns=None, level=None,
inplace=True, errors='raise')
labels_temp.reset_index(drop=True, inplace=True)
labels_temp.head()
```



	Maximum_displacement[cm]	Utilization	Mass[kg]	Norm_dis	Norm_Util	Norm_Mass	perf	perf_stand	perf_norm
0	8.176643	0.460882	1650.825661	0.401581	0.211992	0.603486	0.366127	0.233195	0.453240
1	8.230162	0.465895	1659.193070	0.412975	0.218017	0.627971	0.377991	0.413046	0.474081
2	8.552545	0.597416	1595.008066	0.481604	0.376071	0.440150	0.431100	1.218152	0.567376
3	7.333896	0.448868	1647.096537	0.222175	0.197555	0.592573	0.286407	-0.975309	0.313200
4	8.075146	0.512321	1632.986222	0.379974	0.273809	0.551283	0.371770	0.318744	0.463153

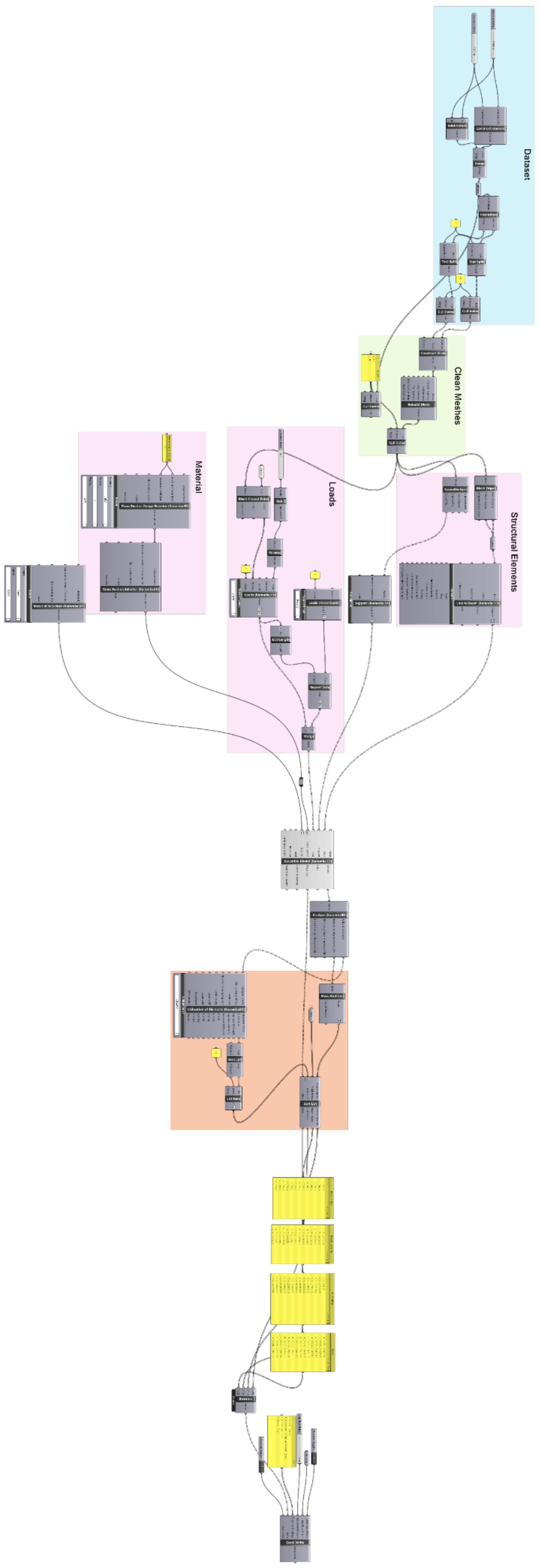
```
#exclude 50 samples from training the surrogate
lst=[]
for i in range(50):
    lst.append(i)

train_labels=labels_temp["perf_norm"]
labels_np=np.array(train_labels)
labels_np=labels_np.astype('float32')
labels_np=labels_np.reshape(5879,1)
y_train=np.delete(labels_np, lst, axis=0)

#Trainning data pre-process
features_np = np.asarray(out)
x_train=np.delete(features_np, lst, axis=0)
x_test=np.take(features_np,lst,axis=0)
print (x_test.shape)
y_test=np.take(labels_np,lst,axis=0)
print (y_test.shape)

#Surrogate-Linear Regression
normalizer = tf.keras.layers.Normalization(axis=1)

def build_and_compile_model(normalizer):
  model = keras.Sequential([
      normalizer,
      keras.Input(shape=(1,81,81)),
      layers.Flatten(),
      layers.Dense(32, activation='relu'),
      layers.Dense(32, activation='relu'),
      layers.Dense(1)
  ])

  model.compile(loss='mean_absolute_error',
                optimizer=tf.keras.optimizers.Adam(5e-4))
  return model
```

```
dnn_model = build_and_compile_model(normalizer)

#Train
history = dnn_model.fit(
    x_train,
    y_train,
    validation_split=0.2,
    verbose=1, epochs=100,batch_size=32)


#Get loss history
loss=history.history['loss']

#Save the surrogate model
save_path= os.getcwd()
dnn_model.save(save_path)

loc_history_loss=os.path.join(job_directory,save_his,"loss")
loc_history_val_loss=os.path.join(job_directory,save_his,"val_loss")
```

#Gradient Descent

```
latent_dim = 10
gradient_lst=[]#this is the root mean square gradients list
perf_lst=[]#this is the performance list
mesh_lst=[]#this is the adjacency matrices list

test_mesh=dataset_eval[4775]
test_mesh= test_mesh.astype('float32')
test_mesh=test_mesh.reshape((1,81, 81))
x= vae.encoder(test_mesh)
z=x[2]

#Gradient descent for a single sample:
for i in range(100):
    with tf.GradientTape() as tape:
        tape.watch(z)
        decodedtensor=vae.decoder(z)
        decodedtensor= layers.Reshape((1,81,81))(decodedtensor)
        y = dnn_model_loaded(decodedtensor)
        gradient = tape.gradient(y,z)

    gr_arr=gradient.numpy().reshape(latent_dim)
    rms=np.sqrt(np.mean(gr_arr**2))
    gradient_lst.append(rms)
    yarr=y.numpy()
    yarr=yarr.reshape(1)
    perf_lst.append(yarr[0])
    decodedtensor=vae.decoder(z)
    decodedtensor=tf.round(decodedtensor)
    decodesarray=decodedtensor.numpy()
    mtx=decodesarray.reshape(81,81)
    mesh_lst.append(mtx)
    z=z-(lr*gradient)
    if yarr[0]<0.19:
        break

#VAE for option 1: Using the structure of vertices as training data
#Load data
File= os.getcwd()
dataset=np.load(file, mmap_mode=None, allow_pickle=False, fix_imports=True,
encoding='ASCII')

#Normalize
dataset = dataset.astype('float32') / 15.

#Dataset pre-processing
dataset = dataset.reshape((dataset.shape[0], 81, 3))
train_dataset,test_dataset=train_test_split(dataset, test_size=0.2,
train_size=None, random_state=None, shuffle=True, stratify=None)

train_size = (train_dataset.shape)[0]
batch_size = 32
test_size = (train_dataset.shape)[1]

train_dataset = (tf.data.Dataset.from_tensor_slices(train_dataset)
                 .shuffle(train_size).batch(batch_size))
test_dataset = (tf.data.Dataset.from_tensor_slices(test_dataset)
                .shuffle(test_size).batch(batch_size))

#Sampling later is the same as option 2
#Class VAE is the same as option 2

#Encoder
latent_dim = 9
encoder_inputs = keras.Input(shape=(81, 3, 1))
x = layers.Flatten()(encoder_inputs)
x = layers.Dense(81, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z], name="encoder")
encoder.summary()

#Decoder
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(81, activation="relu")(latent_inputs)
x = layers.Dense(81*3, activation="sigmoid")(x)
decoder_outputs = layers.Reshape((81, 3))(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
decoder.summary()


#Trainning
vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam())
history = vae.fit(dataset, epochs=200,
batch_size=64,validation_split=0.2,verbose=2)
```

FEM SIMULATION AT GRASSHOPPER WITH KARAMBA3D



TIMELINE:

	Choose Topic
16th November	**P1 Presentation**
	Study Literature related to AI Generative Models
	Study Literature related to possible study cases
	Experiment with Generative Models and existed Tutorials
	Study Python libraries (COMPAS)
	Write report
4th April	**P2 Presentation**
8th May	Finish the generation of the training dataset
15th May	Finish FEM simulations and label the data
	Start Building a Variational Autoencoder Model
	Finish training a Variational Autoencoder and check if more training data are needed
1st June `	**P3 Presentation**
30th June	Build a basic Surrogate Model
30th June	Build a basic Gradient Descent Optimizer
	Train a Surrogate Model
30th July	Check if the Surrogate Model is trained successfully
	Run a Gradient Descent Algorithm
1st -30th August	Vacation
20th September	Gather results and check if better results are being produced
	Make any needed improvements
	Gather conclusions
	Reflect on possible improvements
	Finish the Report
15th October	**P4 Presentation**
	Make corrections based on remarks and Improve the Report
October final week	**P5 Presentation**