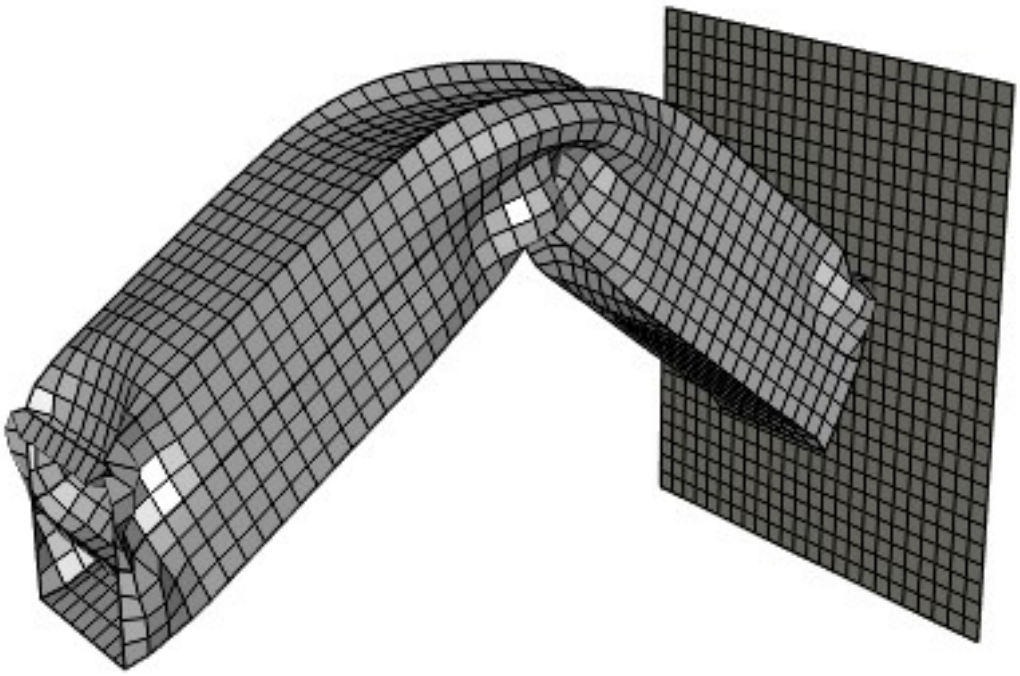


Convolutions on graphs for learning vehicle crash behaviour

Different methods to create graph embeddings

Master's thesis in Computer science and engineering

Daniel Adin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Convolutions on graphs for learning vehicle crash behaviour

Different methods to create graph embeddings

Daniel Adin

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Convolutions on graphs for learning vehicle crash behaviour
Different methods to create graph embeddings
Daniel Adin

Supervisor: Selpi Selpi, Department of Computer Science and Engineering
Advisor: Sandeep Shetty, Volvo Car Corporation
Examiner: Devdatt Dubhashi, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2021

Convolutions on graphs for learning vehicle crash behaviour
Different methods to create graph embeddings
DANIEL ADIN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Convolutional Neural Networks (CNN) have shown successful results in the recent years, especially within the area of image analysis. The idea of learning to predict the result of a crash simulation using machine learning rose from the analogy between images and Finite Element models (FE-models) used in crash simulations. However, the data used when training a machine learning model using CNN needs to be structured in a consistent way, as images are. FE-models however are represented as graphs and do not have the grid-like structure that images have and can therefore not be directly processed using CNN. The purpose of this project was to investigate the possibility to transform FE-models into image-like embeddings and to use CNN to explore these embeddings.

Two graph convolutional methods were investigated for the creation of the embedding. The first one was the Neural Graph Fingerprint (NGF) method suggested in the literature for the original purpose of parsing molecular graphs. The second one was developed during this project, called the FEMBEDDING method, and was to parts inspired by NGF and the Graph Neural Network model that also has been suggested in literature.

Three datasets of crash simulations with varying geometrical complexity were developed during the project. It is shown here that embeddings created by using both methods can successfully be used to train a CNN and predict the outcome of the test sets with a good level of accuracy already with only randomly initialized embedding weights. The FEMBEDDING method made the embeddings richer in information and performed consistently better than the NGF method. For the more geometrical complex dataset it is shown that the value of the FEMBEDDING embeddings increases with an increased neighbourhood depth taken into account while parsing the the FE-graphs.

Keywords: Computer, science, computer science, engineering, graph, convolutions, finite element method, project, thesis.

Acknowledgements

Special thanks to my supervisors Selpi Selpi and Sandeep Shetty for their valuable support during the work of this thesis.

Daniel Adin, Gothenburg, October 2021

Contents

List of Figures

List of Tables

1
Introduction

1.1 Background

Feedforward Neural Networks (NNs) also called Multi Layered Perceptrons (MLPs) have its roots in the perceptron algorithm invented already in 1958 by Frank Rosenblatt [1]. Early ambitions was to emulate the brain, hence the name. Recent years, a special type of NN's called Convolutional Neural Networks (CNN) have shown successful results, especially within the area of image analysis. The idea of learning to predict the result of a crash simulation using machine learning rose from the analogy between images and Finite Element models (FE-models) used in crash simulations. Images consist of pixels with corresponding RGB-values whereas FE-models consist of elements with it's properties. The data used when training a machine learning model using CNN needs to be structured in a consistent way. An image is for example a grid of pixels where each pixel is a feature with a value and have it's specific position in the grid. FE-models however are represented as graphs and do not have this grid-like structure and can therefore not be directly processed using CNN. It has been shown in literature [5] that if graphs are transformed into so called embeddings, meaningful information can successfully be extracted from these embeddings in a downstream MLP, already with only randomly initialized weights, i.e without actually training the weights used in the embeddings. The aim of the work in this thesis project is to show that information can be extracted from FE-model embeddings in a downstream CNN and to investigate the value of different ways to create these embeddings.

1.2 Motivation

The automotive industry is highly competitive, and successful companies are required to present attractive products at the right time. From a customer perspective, "attractive product" could refer to many things: price, design, safety, handling, durability, usability etc.
In the area of vehicle safety, increasingly more of the safety development is done through virtual testing, i.e. simulations. Physical testing is more characterized by verification of a finalized product. However, even if the cost of simulating a complete vehicle crash is a fraction of the cost of a physical crash test, it is still both time consuming and costly. A finished car has gone through several thousand complete vehicle crash simulations and each simulation takes 12-24 hours using several hundreds of CPU-cores. A method that can reduce the response time of a

crash simulation is the motivation behind this work.

1.3 Purpose and goals

The first aim of this project is to investigate the possibility to transform FE-model graphs into image-like embeddings and use CNN to extract meaning from these embeddings. The second aim of this project is to investigate different algorithms to create the embeddings and their ability to express the underlying geometry of FE-models and thereby their ability to predict the outcome of a crash event. The long term goal of the idea at hand is to be able to reduce the time and resources needed to reach the overall safety requirements in a vehicle development program through the use of experience in terms of previously performed simulations. As an example of incentive, optimization of complete vehicles is in practice impossible, due to the needed resources and time to complete such a task.

1.4 Related work

In the bio-mechanical domain there are examples of usage of machine learning to leverage the analysis speed to different Finite Element Analysis problems, for example [23], [17] and [20]. Also in the area of material property predictions machine learning is used, for example [24] and [14]. A few examples exists where the constitutive properties are modeled, [15] and [18]. Articles that aim at predicting mechanical behaviour using ML are [16] and [22]. However, in none of the aforementioned studies, the graph-based nature of FE-models have been addressed. Instead the models have been adjusted to match a pre-defined structures, e.g. the number of input nodes/elements and its order is held constant for each sample and mapped to it's corresponding output variable.

In recent years deep learning on graphs have emerged as a new "hot" topic. With reference to the success of CNNs within computer vision, there are many proposals to capture this success but with graphs as underlying data structure. There are two main tracks when pursuing the convolutional operation on graphs, Spectral methods (for example [3] and [6]) and Spatial methods (for example [5], [4] and [12]). In spectral methods filters are applied to the eigenmodes of the graph Fourier transform. In spatial methods, local features in the graph are extracted through propagation between neighboring vertices, so called message passing. When dealing with 3D-shape analysis it has been suggested in literature that the spatial methods are better choices [11].

The primary application area for graph embeddings is for classification of molecule graphs [7, 5, 10, 9]. To the best of the author's knowledge, there are no examples of studies where the underlying graph data in FE-models have been explored in order to predict the outcome of FE-simulations.

1.5 Scope, limitations and ethical considerations

The datasets used in this project are quite simple and were created specifically for the purpose of this project. The intention is not at this stage to capture all aspects of a crash event but the focus lies in decoding the geometry into embeddings and to extract meaning from these embeddings.

The following stake holders can be identified within the scope of this work:

- The data scientist (the author of this thesis)
- The University of Gothenburg
- Volvo Cars Corporation

It is hard to see any ethical implications in any way at this stage for any of the stakeholders. A question could perhaps be raised asking why customers of cars, or other people in the traffic around a car produced by VCC, are excluded as stakeholders. The reason for this is that the data used in this project is uniquely produced for the purpose of this project. It is not directly connected to the product sold to customers and therefore not an issue for now. Outside the scope this work, considerations could perhaps be made taking the probabilistic nature of machine learning into account and how this could affect the safety of the vehicle that is developed using these kind of methods. However, once the optimum design is found using a probabilistic model, the deterministic model could of course be used for validation purposes.

2
Theory

2.1 Crash simulations

Crash simulations are performed using the Finite Element method (FE-method). In this section the different parts of this process will be briefly described.

2.1.1 Pre-processing

The process of building an FE-model ready to be simulated is called pre-processing. There are many things to consider when building an FE-model for predicting crash events. There are large deformations, the material ruptures and parts that are in contact with each other. All these properties needs to be captured in the model. The model needs also to be prepared for which type of solver that will be used and what type of computer resources that we have at hand.
A physical object can be arbitrary complex in shape. The FE-model discretize this shape into a finite number of *nodes* and *elements*. The process of discretizing the physical object is called *meshing* and usually include some kind of pre-processing of the geometry. In Figure 2.1 an example can be seen, where the middle surface is first extracted from the geometry and the resulting surface is meshed with elements of constant thickness.

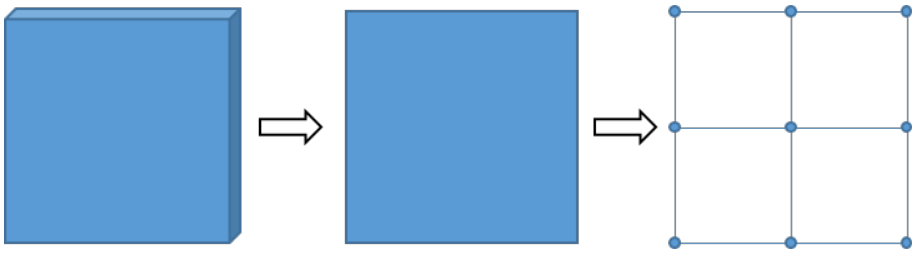


Figure 2.1: A square plate represented by nodes and elements.

The nodes carry the spatial information (x, y and z coordinates) while the elements carry information of constitutive relations, i.e. how the elements respond to forces acting on it. Groups of connected elements with the same constitutive relations are grouped together into *parts*. The parts can in that way be collectively assigned information as material data, thickness, element type etc.

2.1.2 Simulation

When simulating the FE-model that has been built a *solver* is used. A crash event is dynamic (large inertia) and transient (short pulse or initial velocity). The solution to the problem formulated by the FE-model can be obtained through numerical integration of the equations of motion. For crash events this is usually done by a solver using the explicit time integration method where the nodal displacement for the next time step is a function of previous nodal displacement and derivatives thereof.

2.1.3 Post-processing

The process to analyze the results of a FE-simulation is called post-processing. Results in the form of deformations, stresses or accelerations are analyzed and compared with requirements. Models of passengers, so called dummies, are also analysed. Loads, accelerations and deflections of the dummy-body are analysed and compared with requirements thereof. Since an FE-model of a complete vehicle is a very complex model, it can be hard to understand exactly why a certain result is received. Experience plays an important role in terms of trying out efficient countermeasures to improve a result.

2.2 Machine learning

In this section the Machine Learning (ML) methods used in this project is described. There are many ML methods to choose from. The reason behind choosing Neural Networks as the main architectures to work with is because these can mimic non-linear functions. Since crash events normally are non-linear in several aspects, this seems like a natural choice. Convolutional Neural Networks are proven to be efficient and have the ability to capture complex patterns which could be an attractive property when parsing FE-models for information.

2.2.1 Machine learning basics

FE-analysis is a deterministic approach to make predictions about some field of study. We gather all (important) circumstances and use an inferred model to make the prediction. In contrast, ML is a probabilistic approach, i.e. we use prior knowledge of how the circumstances were when a specific outcome was seen and assume that the same outcome will occur if the circumstances are similar. Broadly speaking, machine learning can be categorized into the following:

- Supervised learning: Learning a model how to map input data to an output.
- Unsupervised learning: Trying to find structure or patterns in the input and group the occurrences so that similar samples are grouped together
- Reinforcement learning: The model interacts with a dynamic environment and is adjusted so that it is continuously improved.

This study is in the area of supervised learning, where we have access to both input, which in this case is the FE-model, and output that is the result of the

FEM-simulation. Supervised learning can be further categorized into the following:

- Classification: Learning a model to map the input to a label.
- Regression: Learning a model to map the input to real numbers

The problem at hand is a typical regression problem, where we strive to map a certain input to real numbers. In the following method descriptions only supervised regression will be considered.

2.2.2 Neural Networks

For this section inspiration has been taken from [8]. The idea of a Neural network is to act as a function that maps some input $\mathbf{X}$ to an output y. The training data and training process is used to shape this function to be able to generalize in a way that when given an input it will produce an approximation $\hat{y}$ of the true output y. The approximation that is done during training is that the true input distribution and true output distribution are replaced by a limited number of samples that represents these true distributions. However, the larger (or more representative) this limited number of samples is, the better is the approximation.
The network is built up by layers:

- input layer: $\mathbf{X} = [X_1, X_2, ..., X_N]$
- hidden layers: $\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}, \mathbf{a}^{[l]} = \sigma(\mathbf{z}^{[l]})$, $l \in [1...M]$, and $\mathbf{a}^{[0]} = \mathbf{X}$
- output layer: $\hat{y} = \mathbf{a}^{[M]}$

where $\mathbf{X}$ is a vector containing the input features of a sample, N is the number of features, M is the number of hidden layers and $\hat{y}$ is the predicted output from the network. σ is a non-linear activation function. The most commonly used activation function in modern neural networks is the Rectified Linear Unit, ReLU, that outputs a zero output value for all negative inputs and equal to the input otherwise, i.e. $ReLU(z) = max\{0, z\}$. The trainable parameters in the network are the weights $[W^{[1]}, ..., W^{[M]}]$ and biases $[b^{[1]}, ..., b^{[M]}]$ and is summarized with the notation Θ, hence the output from the network can be written

$$\hat{y} = f(\Theta, \mathbf{X})$$

The error between the predicted output and the true output for a sample is called loss, $L(f(\Theta, \mathbf{X}), y)$. Since we want to achieve a model that performs well for all samples in the training set, we formulate a function that accumulates the loss for all samples, a so called cost function

$$J(\Theta) = \frac{1}{m}\sum_{i=1}^{m} L(f(\Theta, \mathbf{X}), y)$$

where m is the number of sample in the training set.
The most common cost function is probably the mean squared error, MSE:

$$J(\Theta)_{MSE} = \frac{1}{m}\sum_{i=1}^{m}(f(\Theta, \mathbf{X}) - y)^2$$

Other popular cost functions are the mean absolute error, MAE and the root mean squared error, RMSE:

$$J(\Theta)_{MAE} = \frac{1}{m}\sum_{i=1}^{m}|f(\Theta, \mathbf{X}) - y|$$

$$J(\Theta)_{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(f(\Theta, \mathbf{X}) - y)^2}$$

So, the target of the training of our model is to vary the trainable parameters in such way that we minimize the cost:

$$\Theta^* = arg\,\min_{\Theta} J(\Theta)$$

During training, the cost function is optimized using Stochastic Gradient descent or a variant thereof.

2.2.3 Convolutional Neural Networks

For this section inspiration has been taken from [8]. A convolutional neural network is a neural network where one or more layers are convolutional layers. From a more general mathematical point of view a convolution is mathematical operation (see for example [25]) on two functions (f and g) that produces a third function (f*g) that expresses how the shape of one is modified by the other. In deep learning a convolutional operation is interpreted a bit more flexible. It is the sum of element wise multiplication between a limited area of an input matrix and an equally sized kernel matrix. This operation is repeated throughout the input matrix while striding the kernel over the area of the input matrix (left to right, top to bottom). The result from each element is placed in the corresponding place in an output matrix, see Figure 2.2.

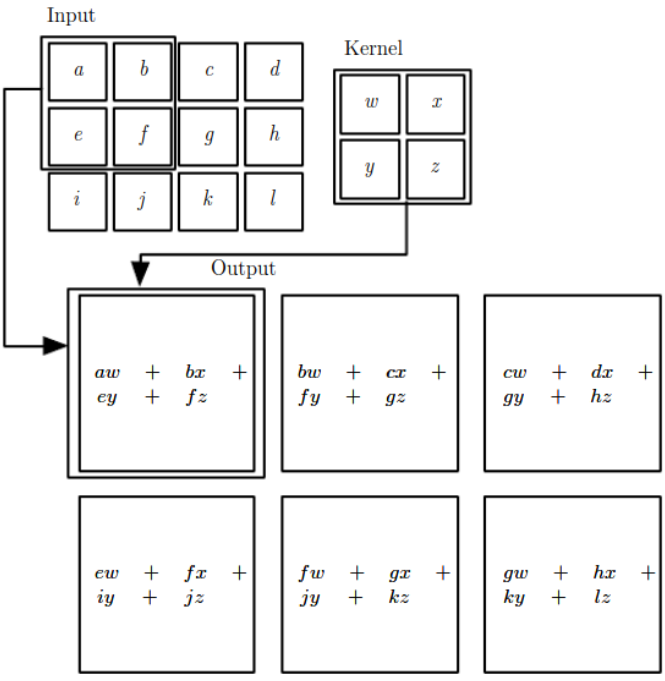


Figure 2.2: Convolutional operation in neural networks, image from [8]

A way to intuitively look upon the convolutional operation is that the convolution compares the kernel with all the different areas in the input matrix. If a certain area resembles the kernel, the response will be magnified in the output for this specific position in the input. It should also be noted how the receptive field grows. One element position in the output now represents 4 element positions in the input. If one more layer is stacked sequentially after this one, one single position of an output in the downstream layer will now represent 9 elements in the original input (the third one 16, 4:th one 25 etc) assuming the same kernel-size. This way detailed patterns of an input matrix can be associated with larger combinations of small patterns and in the end mapped to a class/or regression value. This sense of details is very attractive when it comes to FE-analysis and especially vehicle crash analysis. Often small, subtle variations in the input FE-model can change the outcome quite dramatically.
Almost all convolutional neural networks has also a layer called a pooling layer, where the result from the convolutional layer is down sampled to reduce the number of parameters. This has proven to be a efficiency increase that comes quite cheap from a accuracy perspective. Different types of pooling layers exist, max-pooling, mean-pooling etc.
The pooling is a kind of convolution as well, however this time we simply summarise the elements in the kernel while striding. A popular choice is max-pooling of a certain size and stride with the same size. This means that we will pick the largest element in each kernel position to represent the down-sampled version of the input as output, see example in Figure 2.3.

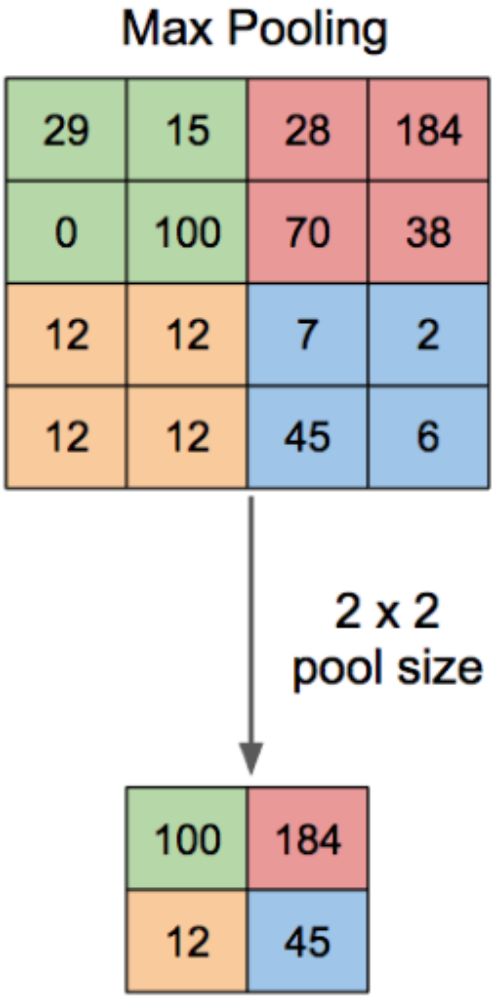


Figure 2.3: Max Pooling operation in neural networks, image from [26]

2.3 Softmax

The softmax function is used in all methods developed in this work and deserves a separate description. It is defined as

$$\sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.1}$$

where $z = [z_1, ..., z_K]$ and $i = 1, ..., K$. The result is vector of same size but where the sum of the elements in $\sigma(z)$ is 1. The elements of $\sigma(z)$ can now be interpreted as the probability of belonging to one of K classes.

2.4 Graph Theory

In this section graph theory will be covered in the context of a FE-model. The simple FE-model with 8 nodes and 3 elements will be used to exemplify the different

concepts.

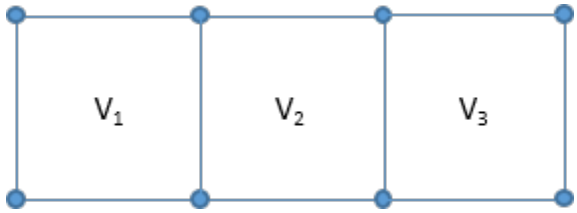


Figure 2.4: A simple FE-model with 8 nodes and 3 elements.

A *graph* is a way to define relations between objects. It is defined as

$$G = (V, E) \tag{2.2}$$

where

$$V = \{v_1, ..., v_N\} \tag{2.3}$$

is the set of N vertices and

$$E = \{e_1, ..., e_M\}, e_m = (v_i, v_j), m \in (1, M), (i, j) \in (1, N) \tag{2.4}$$

is the set of M edges connecting the vertices.
Notice that the order in which we assign the vertices to the set of vertices does not matter and that, if the graph is *undirected*, each edge is defined in the direction of each member vertice.
A convenient way to structure the connectivity that the edges represent is through the *adjacency matrix*. To assemble the adjacency matrix we first decide upon an (arbitrary) order of the vertices and form a vertex feature vector, X. In our sample FE-model for example

$$X = [v_1, v_2, v_3]^T \tag{2.5}$$

Now we can assemble the adjacency matrix, A of size $(N \times N)$, according to this order. We indicate a connection with a 1 in the rows and columns where there is a connection and a 0 otherwise:

$$A = \begin{matrix} Vertices \\ v_1 \\ v_2 \\ v_3 \end{matrix} \begin{matrix} \begin{matrix} v_1 & v_2 & v_3 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \end{matrix} \tag{2.6}$$

In order to explore the surroundings of the vertices in the graph the so called *message pass* can be used. The *sum message pass* is the matrix multiplication of the adjacency matrix and the vertex feature matrix:

$$MP_{sum} = A \times X = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} v_2 \\ v_1 + v_3 \\ v_2 \end{bmatrix} \tag{2.7}$$

Notice how the feature of the explored vertex itself is excluded from the sum. If we want to include the vertex we are exploring, we can add the identity matrix to the adjacency matrix and we get the sum message pass with *self-loops*:

$$MP_{sum} = (I + A) \times X = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} v_1 + v_2 \\ v_1 + v_2 + v_3 \\ v_2 + v_3 \end{bmatrix} \tag{2.8}$$

Another central concept within graph theory is the *degree matrix*. It is defined as the number of neighbours to each vertex positioned along the diagonal. This can easily be derived from the adjacency matrix (with or without self-loops) by counting the non-zero terms on each row and position them along the diagonal. The degree matrix (including self-loops) in our example becomes:

$$D = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{bmatrix} \tag{2.9}$$

Normalizing the adjacency matrix (with self-loops) with the degree matrix we get A_{ns} and we can use that to calculate the *mean message pass*, i.e. the average of the features in the neighborhood of each vertex:

$$A_{ns} = D^{-1} \times (I + A) \tag{2.10}$$

$$MP_{mean} = A_{ns} \times X = \begin{bmatrix} \frac{1}{2}(v_1 + v_2) \\ \frac{1}{3}(v_1 + v_2 + v_3) \\ \frac{1}{2}(v_2 + v_3) \end{bmatrix} \tag{2.11}$$

2.5 Graph Based embeddings

During the literature study of this project two concepts were chosen to be studied more carefully in the project. In this section a brief summary of each of the concepts is presented.

2.5.1 Neural Graph Fingerprint embeddings

Neural Graph Fingerprint (NGF) embeddings is based on the work of Duvenaud et al. in [5]. The origin of this method was the ambition to replace an earlier algorithm, the so called circular molecular fingerprint algorithm with an algorithm that is trainable for specific purposes. The two algorithms can be seen in Figure 2.5.

Algorithm 1 Circular fingerprints

Input: molecule, radius R, fingerprint length S
Initialize: fingerprint vector $\mathbf{f} \leftarrow \mathbf{0}_S$
for each atom a in molecule
  $\mathbf{r}_a \leftarrow g(a)$ ▷ lookup atom features
for $L = 1$ to R ▷ for each layer
  for each atom a in molecule
    $\mathbf{r}_1 \dots \mathbf{r}_N = \text{neighbors}(a)$
    $\mathbf{v} \leftarrow [\mathbf{r}_a, \mathbf{r}_1, \dots, \mathbf{r}_N]$ ▷ concatenate
    $\mathbf{r}_a \leftarrow \text{hash}(\mathbf{v})$ ▷ hash function
    $i \leftarrow \text{mod}(r_a, S)$ ▷ convert to index
    $\mathbf{f}_i \leftarrow 1$ ▷ Write 1 at index
Return: binary vector $\mathbf{f}$

Algorithm 2 Neural graph fingerprints

Input: molecule, radius R, hidden weights $H_1^1 \dots H_R^5$, output weights $W_1 \dots W_R$
Initialize: fingerprint vector $\mathbf{f} \leftarrow \mathbf{0}_S$
for each atom a in molecule
  $\mathbf{r}_a \leftarrow g(a)$ ▷ lookup atom features
for $L = 1$ to R ▷ for each layer
  for each atom a in molecule
    $\mathbf{r}_1 \dots \mathbf{r}_N = \text{neighbors}(a)$
    $\mathbf{v} \leftarrow \mathbf{r}_a + \sum_{i=1}^{N} \mathbf{r}_i$ ▷ sum
    $\mathbf{r}_a \leftarrow \sigma(\mathbf{v} H_L^N)$ ▷ smooth function
    $\mathbf{i} \leftarrow \text{softmax}(\mathbf{r}_a W_L)$ ▷ sparsify
    $\mathbf{f} \leftarrow \mathbf{f} + \mathbf{i}$ ▷ add to fingerprint
Return: real-valued vector $\mathbf{f}$

Figure 2.5: Molecular fingerprint algorithms from [5] reprinted here with permission from David Duvenaud (via email).

The main idea was to use the algorithm for the circular fingerprint and to replace each discrete operation with a differentiable analog. The sorting function is replaced with a permutation invariant function, the hash function is replaced with a single layer neural network and the modulo operation is replaced with a softmax function. With all parts of the algorithm being differentiable, it is possible to do backpropagation, i.e. we can train the weights in the fingerprint algorithm for specific purposes. For the specific implementation of the NGF method in this project, see Section 3.2. NGF was an inspiration in the sense that only a limited number of layers needs to be considered when designing the FEMBEDDING-method developed during the work of this project, see Section 3.3.

2.5.2 Graph Neural Network embeddings

Graph Neural Network (GNN) embeddings is based on the work in [2]. Here a *transition function* F calculates a *vertex state* h_t based on the previous state.

$$h_t = F(h_{t-1}) \tag{2.12}$$

An *output function* G produces an output o_t based on the state.

$$o_t = G(h_t) \tag{2.13}$$

The state is updated in a recurrent way until a fixed value is reached, hence the transition function needs to fulfill certain conditions. According to the paper a sufficient condition is if the transition function is a *contraction map* with respect to the state, i.e. shrinks the distance between two points after applying the transition function to them.
GNN was an inspiration in the sense that a contraction map function can be useful when designing the FEMBEDDING-method developed during the work of this project, see Section 3.3.

3
Methods and experiments

In this chapter the different methods and related experiments are described. The Neural Graph Fingerprint method was specifically designed to parse information in molecular graphs and can be trained for specific purposes but was shown to perform already with randomly initialized weights in [5], i.e. a downstream Neural Network could be trained based on the embeddings without actually modifying the weights of the embeddings themselves. The Graph Neural Network (GNN) method is a generic method and in some sense the origin of neural network on graph data. Both the NGF method and the GNN method were used as inspiration when designing the FEMBEDDING method described in this section. To set the methods into the context of FE-models, the vertex feature vector X is the center of gravity for the N elements in the FE-model, A is the corresponding adjacency matrix and A_{ns} is the normalised adjacency matrix as described in Section 2.4.

3.1 Properties of embeddings

During the development of the algorithms that create the embeddings a set of requirements have been considered. These are

1. The embeddings from graphs of different sizes need to have fixed size in order to be able to use them in a neural network.
2. The embedding needs to be invariant to the order of the input of the features, i.e. how we order/renumber elements/nodes in the FE-model.

As a complement to these requirements a set of desired properties have also been considered. These are

1. The embedding should be invariant to translations and rotations, i.e. moving or rotating an FE-model in space should still give the same geometry encoding. The motivation behind this property is that we strive to separate the geometry encoding from the response to loads acting on the geometry. This would give an increased generalization of the embedding.
2. Two embeddings should be similar if the FE-models represent similar physical objects, i.e. element size should not matter.
3. The embeddings should pay attention to details, i.e. small variations that are important to identify important events, should be present in the embedding.

3.2 Algorithm for embeddings based on NGF

The initial state can be written as

$$r_0 = X \in \mathbb{R}^{N,3} \tag{3.1}$$

For each neighbourhood layer a message pass is performed to gather information from the neighbours into a diffusion vector v_l.

$$v_l = A \times r_{l-1} \in \mathbb{R}^{N,3}, l = (1, ..., L) \tag{3.2}$$

where L is the number of neighbourhood layers considered. Here we can choose whether to use the mean message pass or the sum message pass, i.e. whether the adjacency matrix is normalised with the degree matrix or not, see Section 2.4. Both variants are studied in this project.
The updated state is received by convolving the diffusion vector with a randomly initialized hidden weight matrix H_l of size 3×3 and applying an activation function σ to remove symmetries.

$$r_l = \sigma(v_l \times H_l) \in \mathbb{R}^{N,3} \tag{3.3}$$

The contribution from each neighbourhood layer is derived by convolving r_l with a randomly initialized output weight matrix W_l of size $3 \times S$ and apply the softmax function to the rows.

$$i_l = softmax(r_l \times W_l) \in \mathbb{R}^{N,S} \tag{3.4}$$

Here S is the length of the final embedding vector. The contribution from each layer is calculated through a graph level pooling operation where we use a permutation invariant function to summarise the output from all the vertices. This can be done in several ways, but for simplicity and efficiency reasons summation is used in this project.

$$i_{l,pool} = \sum_{n=1}^{N} i_l \in \mathbb{R}^{1,S} \tag{3.5}$$

Finally we sum the contributions from all the layers to get the final embedding

$$embedding = \sum_{l=1}^{L} i_{l,pool} \in \mathbb{R}^{1,S} \tag{3.6}$$

Algorithm 1 shows the pseudo code of the NGF method used in this project, based on interpretation from [5].

Algorithm 1 Creation of NGF embeddings.

Input: Adjacency matrix A or A_{ns}, coordinate vector X, hidden weight matrices $H_1, ..., H_L$, output weight matrices $W_1, ..., W_L$
Initialize: embedding vector $embedding = 0$, $r_0 = X$
for $l = 1...L$ **do**
    $v_l = A \times r_{l-1}$
    $r_l = (v_l \times H_l).relu()$
    $i_l = (r_l \times W_l).softmax(rows)$
    $embedding = embedding + i.sum(columns)$
end for

3.3 The FEMBEDDING algorithm

The FEMBEDDING method was developed during this project as an alternative to the NGF method. Inspiration was found in the GNN-method in the sense that it uses a transition function that is a contraction map. The calculation of center of gravity is a function that fulfills the requirement as contraction map since an expanding neighbourhood of an element will eventually contain the information from all elements of the FE-model and consequently result in the center of gravity for all elements in the FE-model. This means that the mean message pass described in Section 2.4 is a candidate to be used in the creation of the embedding. Inspiration was also found in the NGF-method since it seems we do not need to exhaust the iterations to convergence in order to get a valuable embedding.
The initial state can be written as

$$h_0 = X \in \mathbb{R}^{N,3} \tag{3.7}$$

and the update of the state can be written as

$$h_t = A_{ns} \times h_{t-1} \in \mathbb{R}^{N,3} \tag{3.8}$$

where the mean message pass is applied as described in Section 2.4. The updated state is simply the center of gravity for the increased neighbourhood.
To get an output that eventually will converge towards zero we can calculate the difference between the current state and the last state,

$$diff_t = h_t - h_{t-1} \in \mathbb{R}^{N,3} \tag{3.9}$$

where $diff_t$ is the vector between the center of gravity between two differently sized neighbourhoods.
The output from this state can now be formulated as

$$o_t = \sigma(diff_t \times W) \in \mathbb{R}^{N,S} \tag{3.10}$$

where σ is an activation function and W is the output weight matrix of size $3 \times S$. We sum the contributions from each iteration up to $t = T$

$$em = \sum_{t=0}^{T} o_t \in \mathbb{R}^{N,S} \tag{3.11}$$

The softmax function is applied to this sum in order to calculate the probability of each vertex belonging to any of the S classes indicated by the output weight matrix.

$$classification_{matrix} = softmax(em) \in \mathbb{R}^{N,S} \tag{3.12}$$

The classification matrix now consists of N rows where the elements of the row sum to 1.
The final embedding is calculated through a graph level pooling operation in the same manner as for the NGF method.

$$embedding = \sum_{i=1}^{N} em_i \in \mathbb{R}^{1,S} \tag{3.13}$$

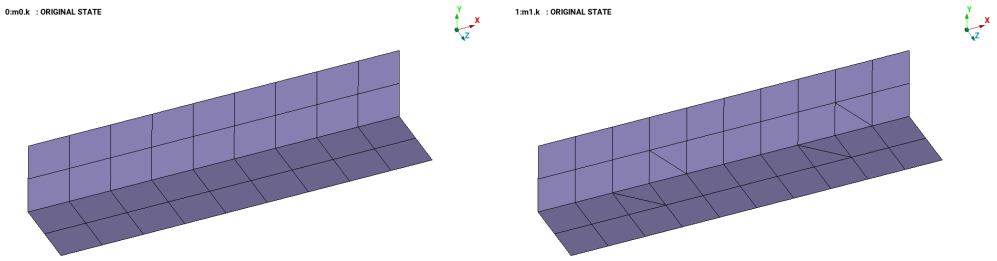
Algorithm 2 shows the pseudo code of the FEMBEDDING method developed in this project, based on inspiration both from [2] and [5].

Algorithm 2 Creation of FEMBEDDINGs.

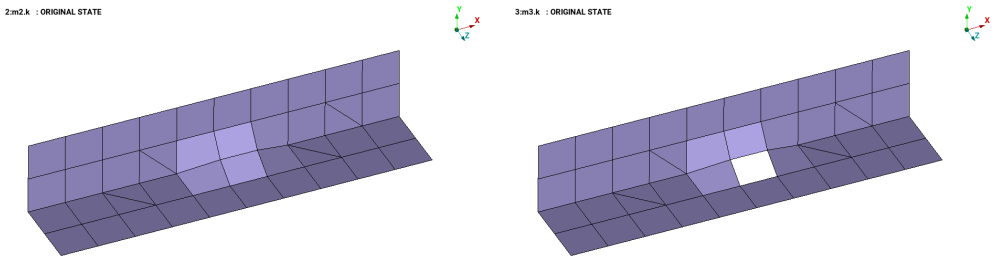
Input: Normalized adjacency matrix A_{ns}, coordinate vector X, output weight matrix W, number of iterations T
Initialize: embedding matrix $em = 0$, $h_0 = X$
for $t = 1...K$ **do**
    $h_t = A_{ns} \times h_{t-1}$
    $diff_t = h_t - h_{t-1}$
    $o_t = (diff_t \times W).relu()$
    $em = em + o_t$
end for
$classification_{matrix} = em.softmax(\text{rows})$
$embedding = classification_{matrix}.sum(\text{columns})$

3.4 Embedding analysis

Both Algorithm 1 and Algorithm 2 fulfill requirement 1 due to that the algorithms output a fixed size embedding vector independent of the size of the FE-model. Requirement 2 is also fulfilled for both algorithms since all vertex features are combined with all columns in the output weight matrices, the order in which this is done is not of importance and the final embedding is the sum of all rows, hence permutation invariant. In terms of the desired property 1 (invariant to translations and rotations) the FEMBEDDING-method fulfill this property because in the algorithm we convolve the difference between the center of gravity of 2 layers while it is not fulfilled by the NGF-method since we convolve the absolute value of each layers center of gravity. In order to judge how the desired properties 2 (insensitivity to mesh) and 3 (attention to details) is fulfilled by the two algorithms, a set of small FE-models was constructed to showcase these properties. In Figure 3.1 the models are shown.



(a) Model0 **(b)** Model1



(c) Model2 **(d)** Model3

Figure 3.1: Models for conceptual comparison.

Model0 is a reference model. Model1 is geometrically identical as Model0 but with a different mesh. Model2 has exactly the same mesh as Model1 except that a few nodes have been moved a small distance. Model3 has exactly the same mesh as Model2 except that one element is deleted.
Embeddings were created using Algorithms 1 and 2. In Figure 3.2 the embedding value has been plotted in the order it appears in the embedding vector for the different models and different number of layers taken into account in the embeddings.

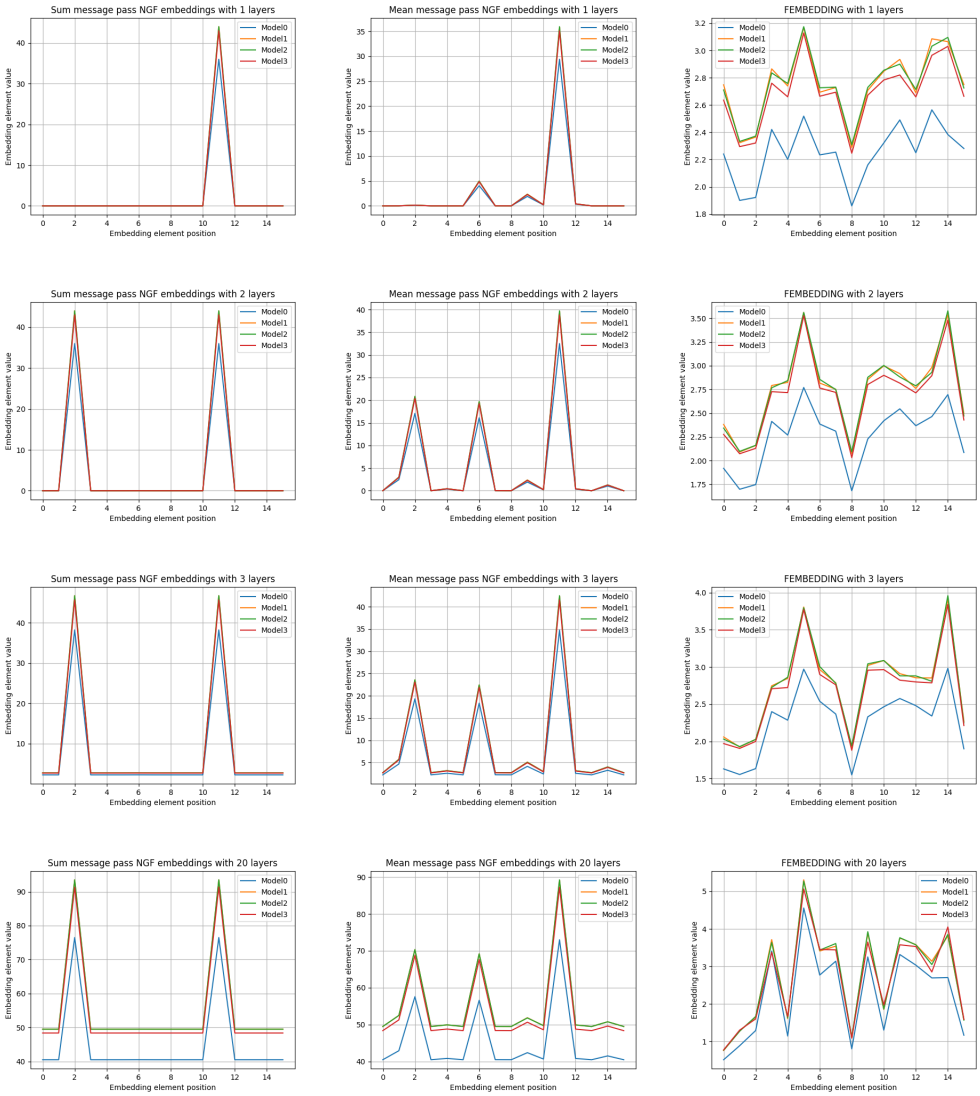


Figure 3.2: Plots of embeddings for the conceptual models. The columns represent the three embedding creation methods that is studied, NGF with sum message pass, NGF with mean message pass and the FEMBEDDING method. The rows show different number of layers included when creating the embeddings, 1, 2, 3, and 20 layers respectively.

From the plots in Figure 3.2 one conclusion is that both methods have problems to create embeddings that are similar for FE-models that just have different meshes. However it appears that in the FEMBEDDING method the error is reduced as more layers are included when creating the embedding. Another conclusion is that the NGF method does not seem to capture the small geometry change between Model1 and Model2, the curves are placed on top of each other, while there is a difference between Model2 and Model3. The FEMBEDDING method seems to capture both these differences, i.e. it appears it gives attention to details as geometrical changes and holes.

3.5 Creating embedding images

In order to use convolutional neural network architectures, the embedding vectors needs to be transformed to image-like structures, which is simply done by changing the dimensionality of the embedding vectors. The dimensionality of the embeddings are initially $[1, S]$ where S is the length of the embedding vectors. The dimensionality for an image-like structure is $[C, W, H]$ where $C = 1$ is the number of channels and W, H is the width and height of the image respectively. For the sake of simplicity a design choice was to set $W = H = \sqrt{S}$. The notation of the width/height is *embedding image size*.

3.6 Experiments

The experiments conducted in this project is done by first creating the embeddings based on the methods described earlier. These embeddings are used when training Convolutional Neural Networks to predict the target values. This is done for three different datasets described below.

3.6.1 Dataset 1

A simple loadcase was setup in order to compare the different ways to create embeddings. The way this loadcase is defined can be seen in Figure 3.3. It consists of a steel beam with wall thickness of 1.5mm with a square cross-section. The beam is constrained in all directions in one end. The other end is also constrained except for the axial direction. A rigid wall with a weight of 75 kg hits the beam axially in this end with a velocity of 7.5m/s. The loadcase is simulated for 30 ms while the traveling distance of the rigid wall and the load that acts upon the wall is registered.

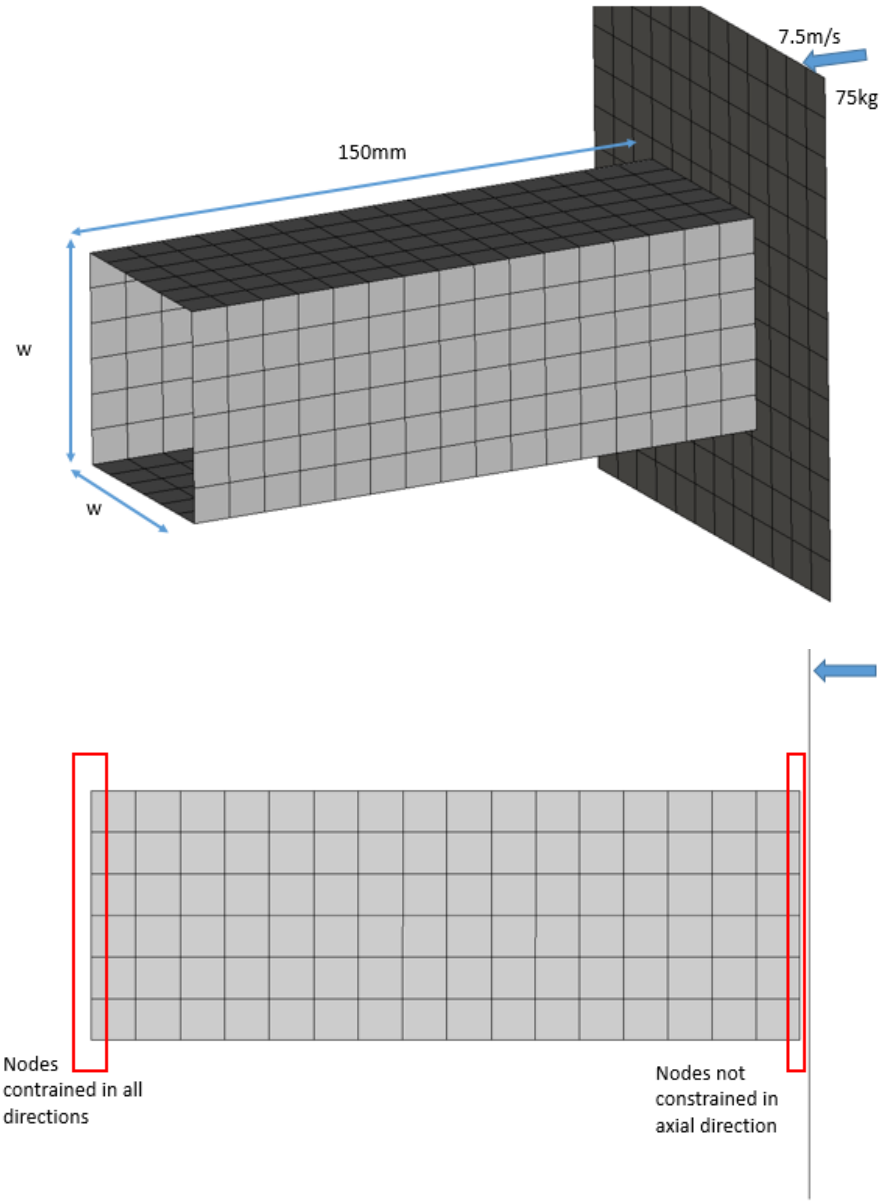


Figure 3.3: Loadcase definition of dataset 1.

The width of the side was linearly varied between 20 and 80 mm. In total 1998 simulations was executed. In Figure 3.4 the deformed state of a sample from dataset 1 is shown.

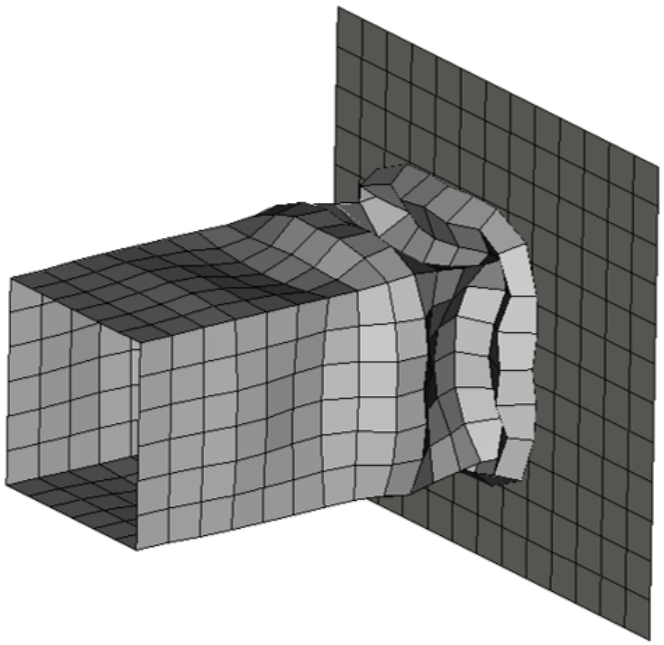


Figure 3.4: Deformed state of a sample from dataset 1.

In Figure 3.5 the targets are plotted as a function of the section side width.

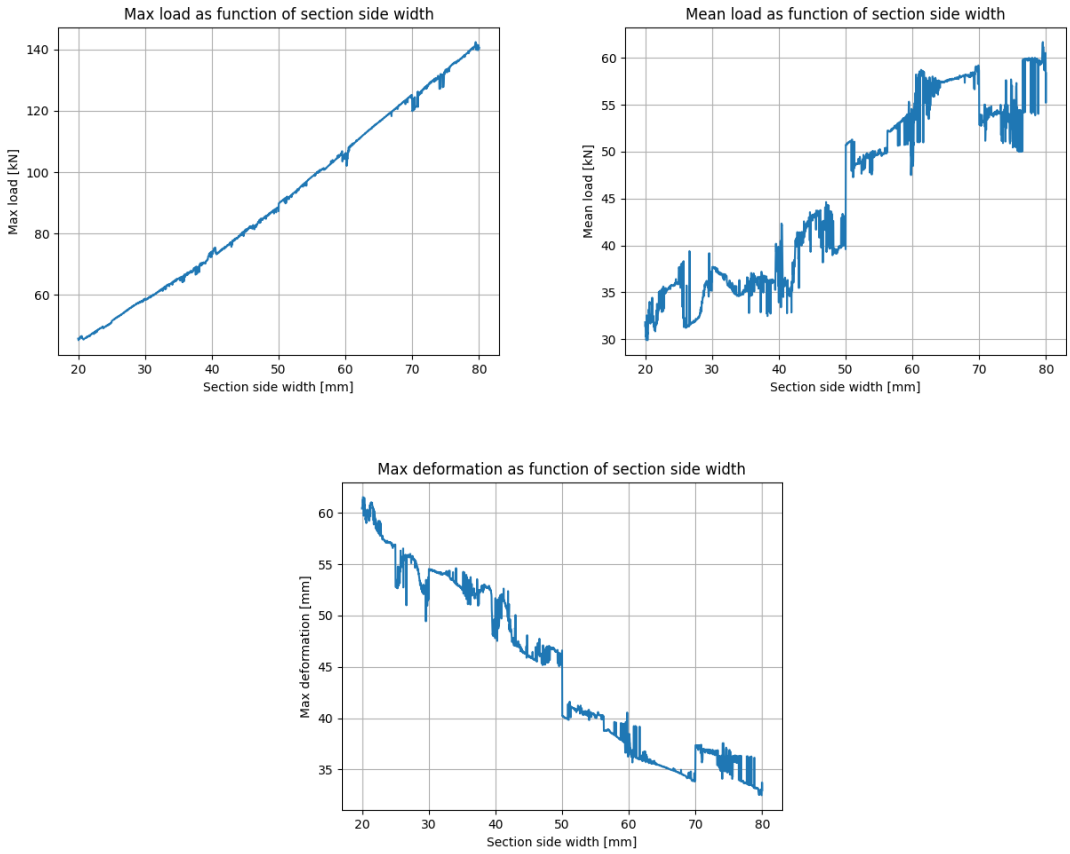


Figure 3.5: The studied target data maximum load, mean load and maximum deformation respectively are plotted as function of cross-section width.

3.6.2 Dataset 2

The second dataset was setup to have a slightly increased complexity. The definition of this dataset can be seen in Figure 3.6. It consists of a steel beam with wall thickness of 1.5mm with a square or circular cross-section. The beam has a variable curvature along its axis. The beam is constrained in all directions in one end. The other end is also constrained except for the axial direction. A rigid wall with a weight of 75 kg hits the beam axially in this end with a velocity of 10.0m/s. The loadcase is simulated for 100 ms while the traveling distance of the rigid wall and the load that acts upon the wall is registered.

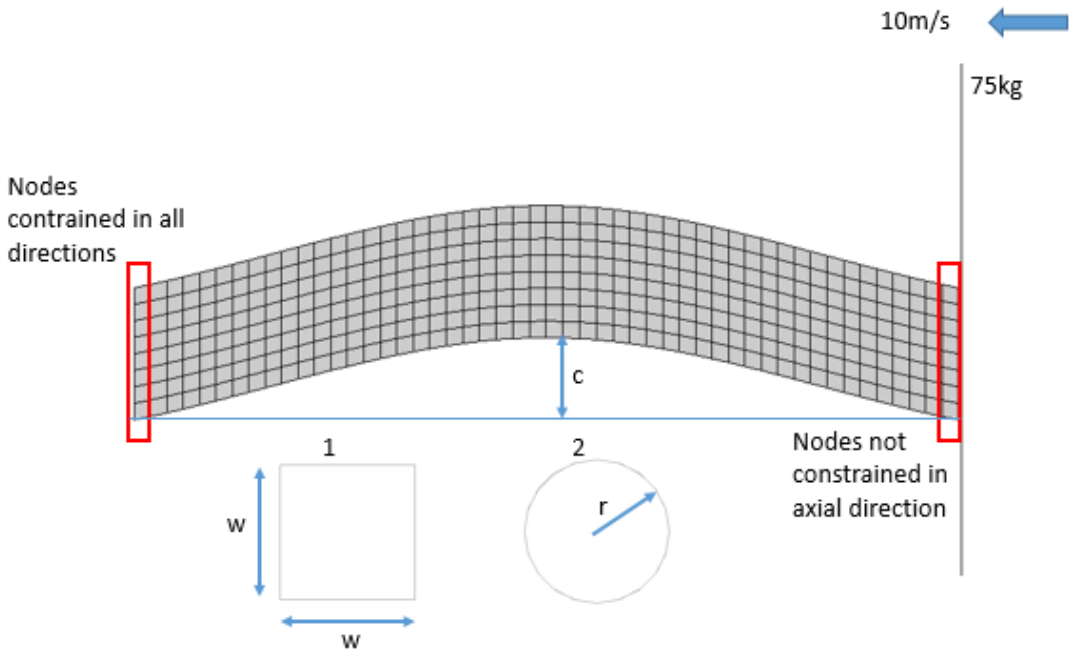


Figure 3.6: Loadcase definition of dataset 2.

The width w of the side of the beam with a square cross-section was varied between 60 and 100 mm. The radius r of the beam with a circular cross-section was varied between 30 and 50 mm. The curvature c of the beams was varied between 0 and 50 mm. In total 5000 simulations was executed. The resulting dataset consisted of 4958 simulations because in 42 simulations, the kinetic energy had not reached 0 when the simulation reached the final time-step, hence these 42 simulations were removed from the dataset. In Figure 3.7 the deformed state of two samples from dataset 2 is shown.

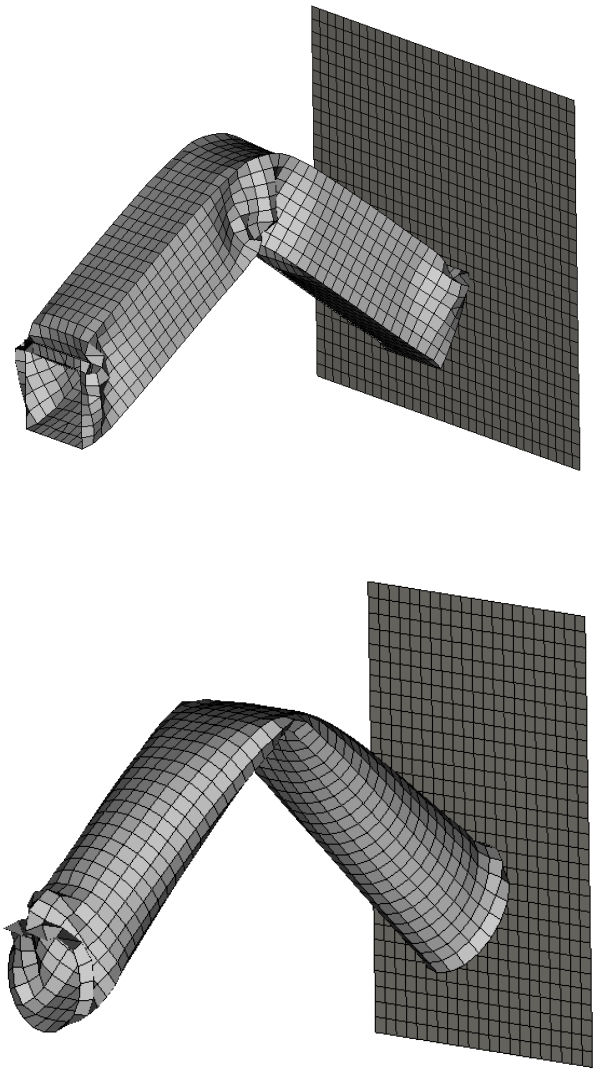


Figure 3.7: Deformed state of samples from dataset 2.

In Figure 3.8 the targets are plotted for a quadratic section of width 80mm and a circular section with radius 40mm as a function of the curvature.

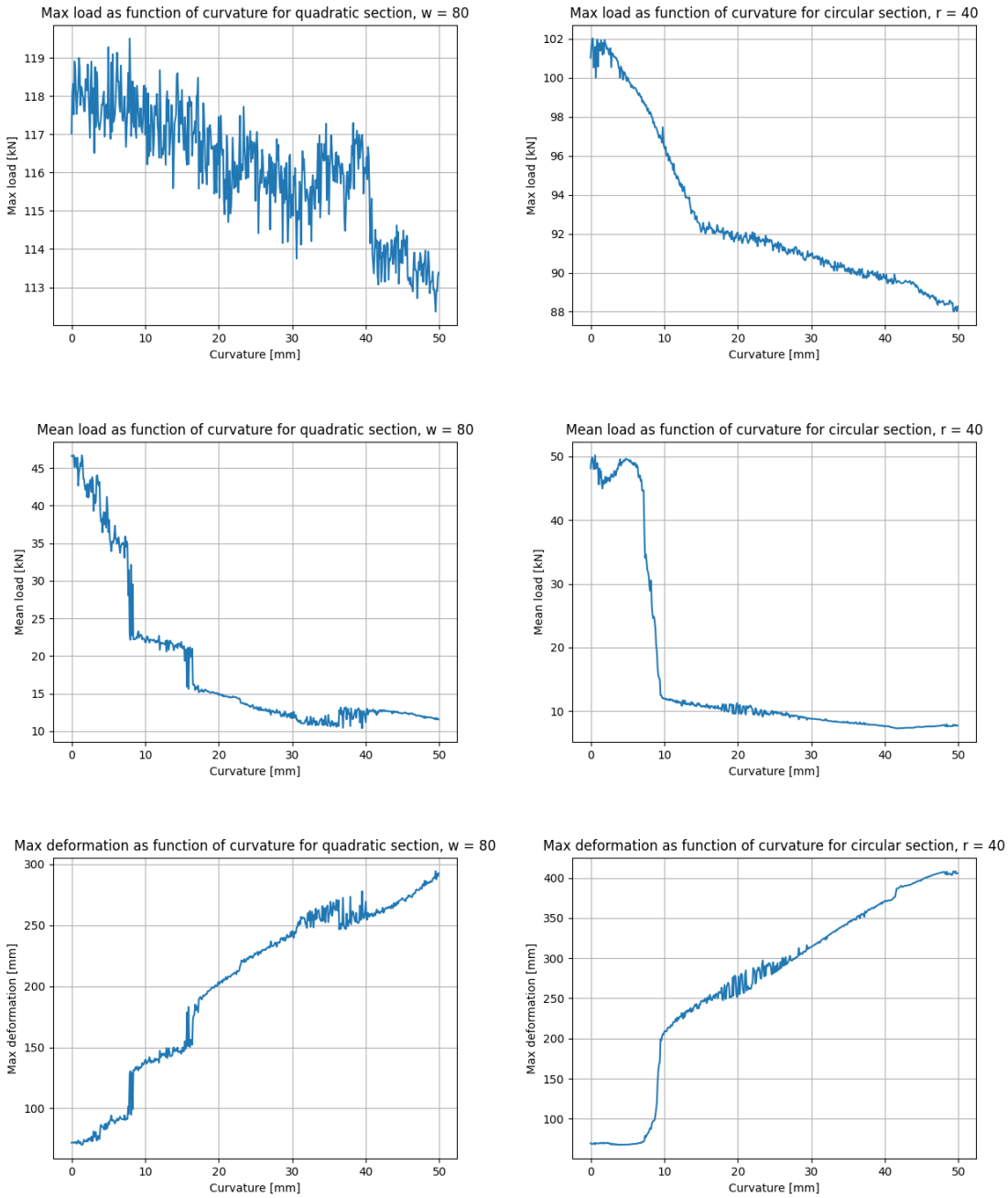


Figure 3.8: Target data as function of curvature in dataset 2.

3.6.3 Dataset 3

The third dataset was setup to have an increased geometrical complexity. The definition of this dataset can be seen in Figure 3.9. As in dataset 1 and dataset 2 it consists of a steel beam with wall thickness of 1.5mm, 4 different section geometries and a curvature along its axial direction. The beam is constrained in all directions in one end. The other end is also constrained except for the axial direction. A rigid wall with a weight of 75 kg hits the beam axially in this end with a velocity

of 10.0m/s. The loadcase is simulated for 100 ms while the traveling distance of the rigid wall and the load that acts upon the wall is registered.

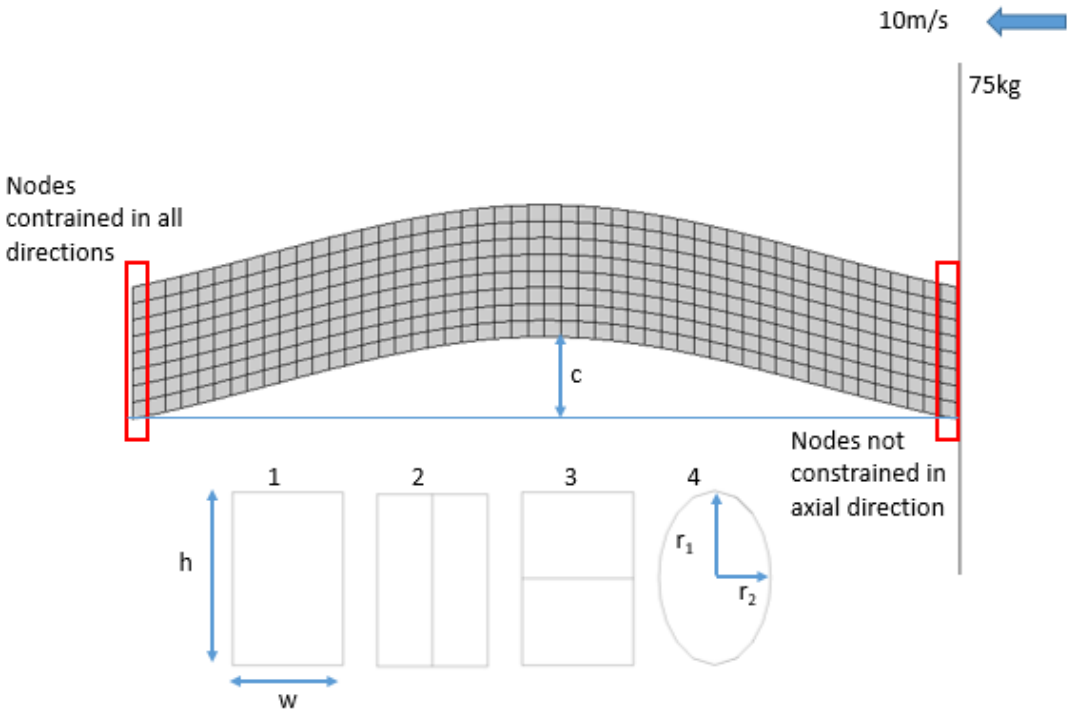


Figure 3.9: Loadcase definition of dataset 3.

The height h and width w of the side of the beams with rectangular sections were varied between 50 and 100 mm. The radius r_1 and r_2 of the beam with an elliptic cross-section was varied between 25 and 50 mm. The curvature c of the beams was varied between 0 and 50 mm. In total 4000 simulations was executed. The resulting dataset consisted of 3972 simulations because in 28 simulations, the kinetic energy had not reached 0 when the simulation reached the final time-step, hence they were removed from the dataset. In Figure 3.10 the deformed state of 4 samples from dataset 3 is shown, one from each main section geometry layout.

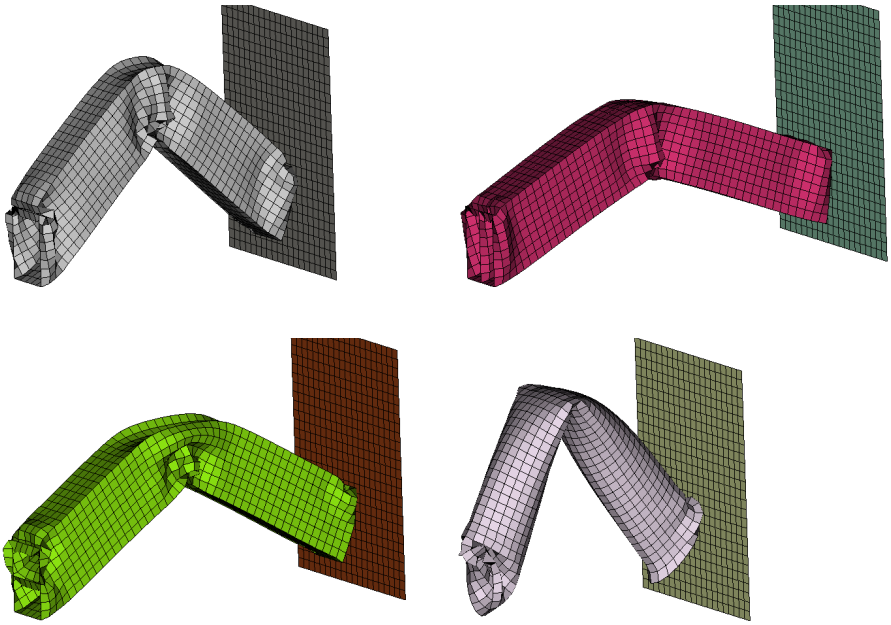


Figure 3.10: Deformed state of samples from dataset 3.

In Figure 3.11 samples of the targets are plotted as function of the curvature for rectangular sections with $h = 80mm$ and $w = 50mm$. The corresponding elliptic section had $r_1 = 40mm$ and $r_2 = 25mm$.

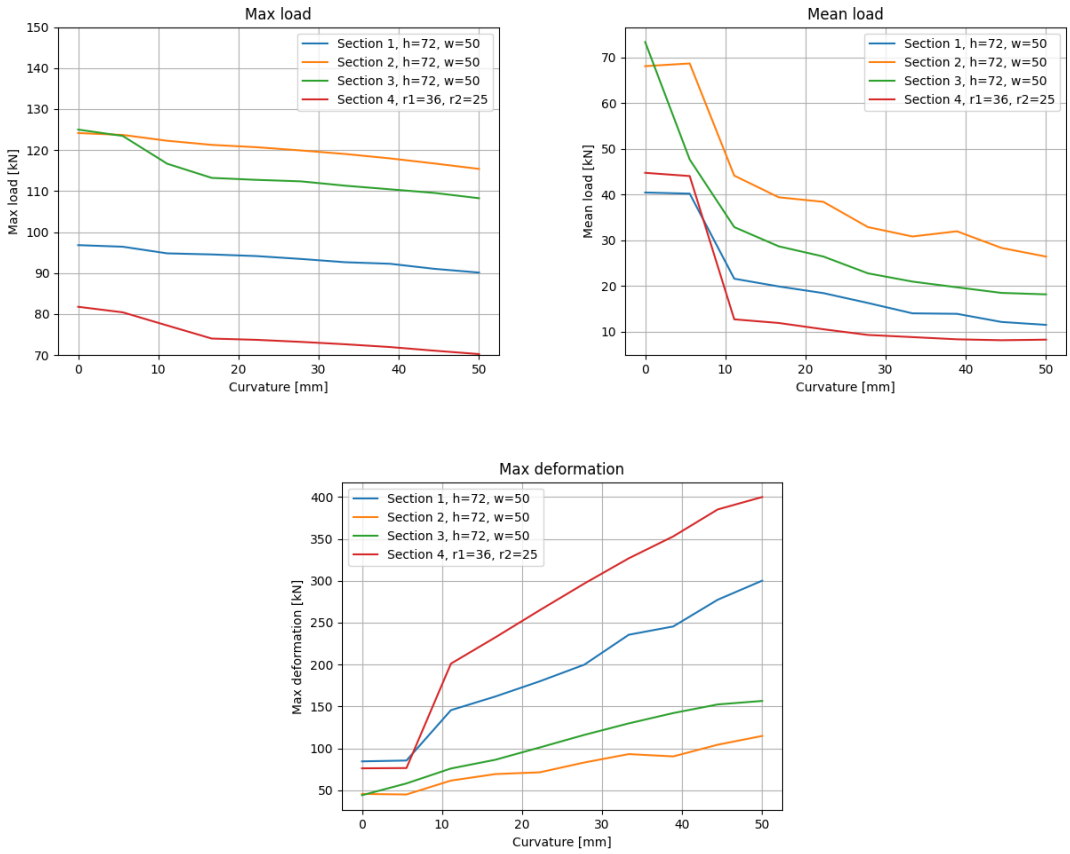


Figure 3.11: Target data as function of curvature in dataset 3.

3.6.4 Data creation, extraction and pre-processing

Normally when working with FE-analyses a lot of the work is done manually or partly with scripts. Since there was a need for many samples to train the models, this was fully done through scripts. In total over 10000 models were built, simulated and post-processed. To form the datasets, the geometry file for each simulation is considered as the source of input to extract the vertex feature vector X from the graph data. From the results of the simulations the values of the maximum load $y_{max-load}$, mean load $y_{mean-load}$ and maximum deformation $y_{max-def}$ were extracted and chosen as target data during training and validation of the CNN models.

3.6.5 Training procedure and Network architecture

The general training procedure consists of the following steps:

- Split the data-set in training and test set. These sets are saved to disc and reused for all embedding types. The training set is used to train the parameters of the CNN model and the test set is used to validate the model performance after training.
- The geometry files are pre-processed so that graph data and target data is available for each sample.

- Embeddings are created from the graphs and saved to disc.
- The embeddings are normalized so the mean is 0 and the variance is 1.
- The CNN model is trained on the training data using a cost function that is the sum of the cost functions related to each target value, i.e.

$$J(\Theta)_{max-load} = \frac{1}{m}\sum_{i=1}^{m}(f(\Theta, \mathbf{X})_{max-load} - y_{max-load})^2 \tag{3.14}$$

$$J(\Theta)_{mean-load} = \frac{1}{m}\sum_{i=1}^{m}(f(\Theta, \mathbf{X})_{mean-load} - y_{mean-load})^2 \tag{3.15}$$

$$J(\Theta)_{max-def} = \frac{1}{m}\sum_{i=1}^{m}(f(\Theta, \mathbf{X})_{max-def} - y_{max-def})^2 \tag{3.16}$$

$$J(\Theta) = J(\Theta)_{max-load} + J(\Theta)_{mean-load} + J(\Theta)_{max-def} \tag{3.17}$$

  where m is the number of samples in the training set. The training results in a model where the parameters have values such that total cost is minimized, i.e.

$$\Theta^* = arg\,\min_{\Theta} J(\Theta) \tag{3.18}$$

- The trained model is evaluated using the test set, i.e. the total cost for the samples in the test set is calculated.

For dataset 1 the embedding image size was chosen to be 8 and the corresponding CNN architecture can be seen in Table 3.1.

Table 3.1: CNN architecture for dataset 1.

Layer no	Layer type	in channels	out channels	kernel size	stride
1	2D CNN	1	16	(3,3)	(1,1)
2	Batch Norm	16			
3	ReLU activation				
4	2D CNN	16	32	(3,3)	(1,1)
5	Batch Norm	32			
6	ReLU activation				
7	2D CNN	32	64	(3,3)	(1,1)
8	Batch Norm	64			
9	ReLU activation				
10	Flattening				
11	Linear	256	128		
12	Batch Norm	128			
13	ReLU activation				
11	Linear	128	3		

For dataset 2 and 3 the embedding image size was chosen to be 16 and the corresponding CNN architecture can be seen in Table 3.2.

Table 3.2: CNN architecture for dataset 2 and 3.

Layer no	Layer type	in channels	out channels	kernel size	stride
1	2D CNN	1	16	(3,3)	(1,1)
2	Batch Norm	16			
3	ReLU activation				
4	2D CNN	16	32	(3,3)	(1,1)
5	Batch Norm	32			
6	ReLU activation				
7	Maxpool			(2,2)	
8	2D CNN	32	64	(3,3)	(1,1)
9	Batch Norm	64			
10	ReLU activation				
11	2D CNN	64	128	(3,3)	(1,1)
12	Batch Norm	128			
13	ReLU activation				
14	Flattening				
15	Linear	512	256		
16	Batch Norm	256			
17	ReLU activation				
18	Linear	256	128		
19	Batch Norm	128			
20	ReLU activation				
21	Linear	128	3		

3.6.6 Hyperparameters for performed experiments

During the experiments, the Adams optimizer was used. The initial learning rate was $1e-2$. It turned out that the training could be performed efficiently with the whole training set read into memory and that the convergence was the most effective with the complete training set as batch size, i.e. so called batch gradient descent was used. In Table 3.3 the different experiments and hyperparameters are shown.

Table 3.3: Experiments and hyperparameters (lr df/int = learning rate decay factor and update interval, i.e. the learning rate is updated by multiplying with the decay factor every stated interval, tlf = target loss function)

Dataset	Layers	lr df/int	Epochs	Trials	Method	tlf
1	1-10	0.9/50	2000	10	NGF smp	MSE
1	1-10	0.9/50	2000	10	NGF mmp	MSE
1	1-10	0.9/50	2000	10	FEMBEDDING	MSE
2	1-10	0.9/50	2000	10	NGF mmp	MSE
2	1-10	0.9/50	2000	10	FEMBEDDING	MSE
3	1-10	0.95/50	3000	10	NGF mmp	MSE
3	1-10	0.95/50	3000	10	FEMBEDDING	MSE
3	10	0.95/50	3000	10	FEMBEDDING	RMSE
3	10	0.95/50	3000	10	FEMBEDDING	MAE

3.6.7 Software

PyTorch [13] was used as the main platform to train the CNN-models and some of the functionality in PyTorch Geometric [19] was used for handling graph data.

4
Results

4.1 Results using dataset 1

Dataset 1 consisted of in total 1998 samples. The training set had 1598 samples and the test set had 400 samples. In Figure 4.1 the average test loss of 10 trials/embedding type has been plotted as a function of increasingly larger neighbourhood when creating the embeddings.

Figure 4.1: Test loss as a function of neighbourhood size for dataset 1.

As can be seen in the figure, the FEMBEDDING method has in general lower test loss then the NGF method with sum message pass (smp) and NGF method with mean message pass (mmp). It was indicated in [5] that the scale of the randomly initialized weights had influence on the possibility to take advantage of the neighbourhood size. The embeddings used in the results in Figure 4.1 had weights that was initialized in the range $[-0.5, 0.5]$. In Figure 4.2 the weights were instead initialized in the range $[-0.25, 0.25]$ and $[-1.0, 1.0]$ respectively.

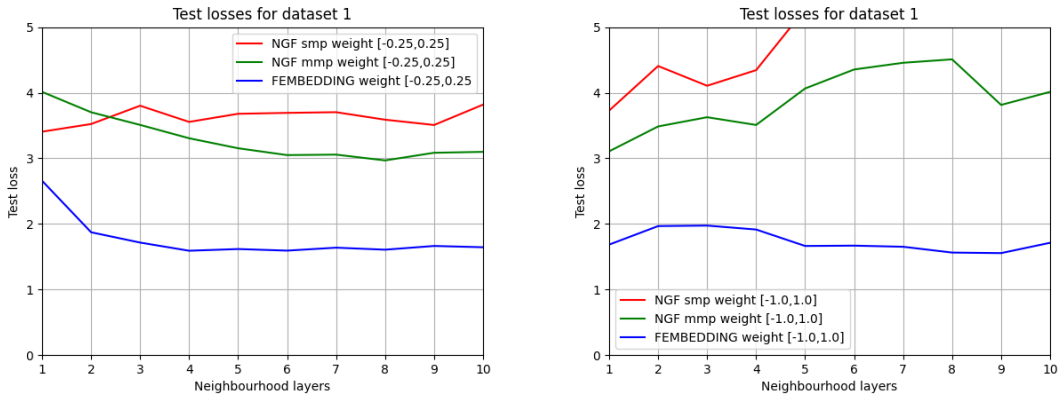


Figure 4.2: Test loss as a function of neighbourhood size for dataset 1 with different weight scales (left: [-0.25, 0.25], right: [-1.0, 1.0]).

From 4.2 we can see that the NGF smp consistently performed worse then the NGF mean message pass, hence the other studies will be limited to NGF mmp. We can also conclude that the FEMBEDDING method consistently performed better than the NGF method and that the tendency to take advantage of the neighbourhood is clearer for somewhat smaller weight scales. However, so far we can not draw the conclusion that this overall leads to smaller average loss.

4.2 Results using dataset 2

Dataset 2 consisted of in total 4957 samples. The training set includes 3965 samples and the test set had 992 samples. In Figure 4.3 the average test loss of 10 trials/embedding type has been plotted as a function of increasingly larger neighbourhood when creating the embeddings.

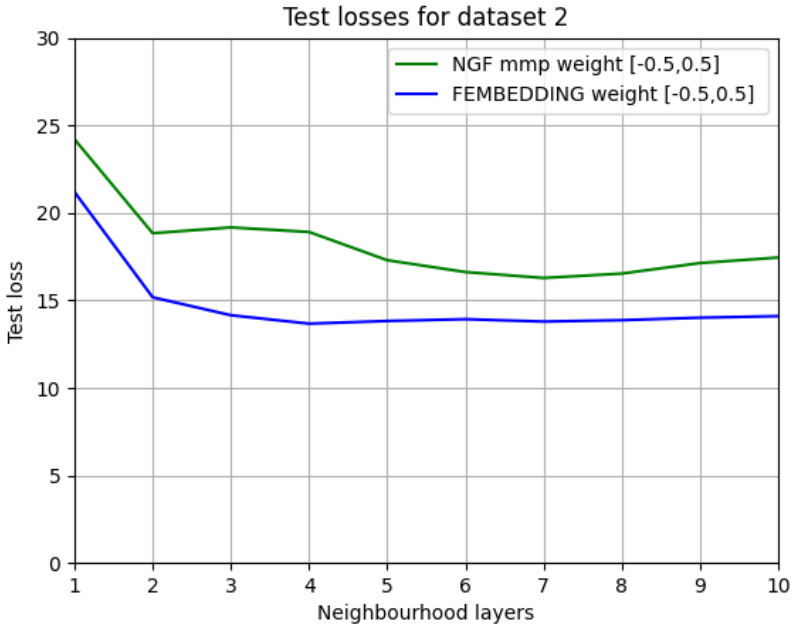


Figure 4.3: Test loss as a function of neighbourhood size for dataset 2.

In Figure 4.4 the weights were initialized in the range $[-0.25, 0.25]$ and $[-1.0, 1.0]$ respectively.

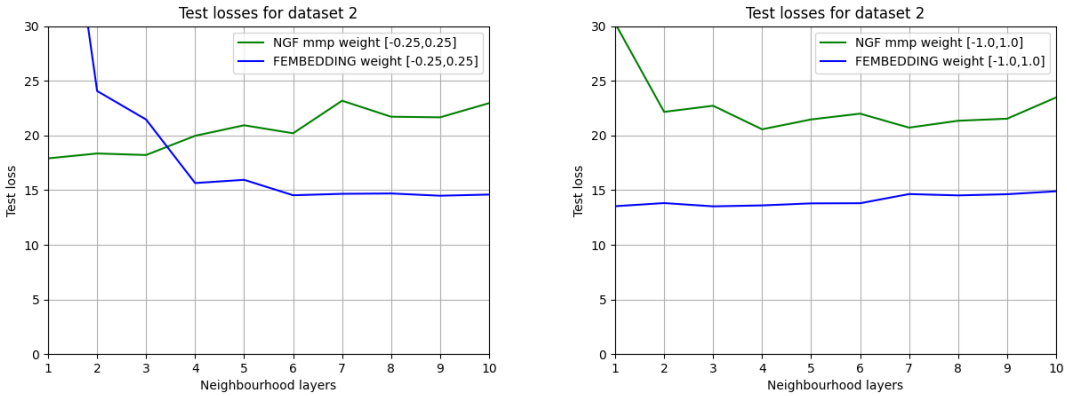


Figure 4.4: Test loss as a function of neighbourhood size for dataset 2 with different weight scales (left: [-0.25, 0.25], right: [-1.0, 1.0]).

As can be seen in the figures, the loss levels are in general higher than for dataset 1. Also for this slightly more complex dataset, embeddings from the FEMBEDDING method showed to be more valuable and reach in general a lower loss level than the NGF method. Furthermore it seems to take advantage of an increased neighbourhood depth for smaller weight scales as was the same for dataset 1. However the lowest loss levels are roughly the same independent of neighbourhood depth. The NGF method seems to show a value of increasing the neighbourhood depth for this

dataset, but for a bit higher weight scales, i.e. not in line with the FEMBEDDING method.

4.3 Results using dataset 3

Dataset 3 consisted of in total 3972 samples. The training set includes 3177 samples and the test set had 795 samples. In Figure 4.5 the average test loss of 10 trials/embedding type has been plotted as a function of increasingly larger neighbourhood when creating the embeddings.

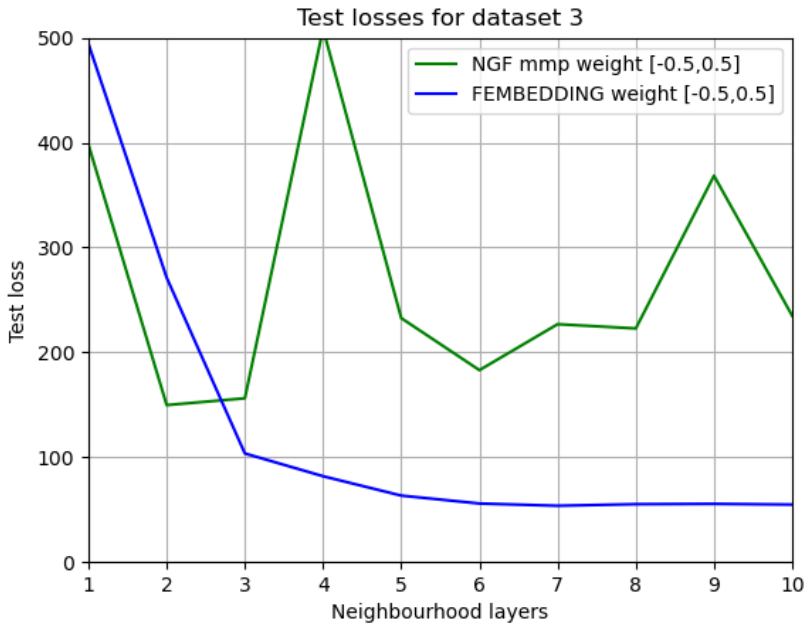


Figure 4.5: Test loss as a function of neighbourhood size for dataset 3.

In Figure 4.6 the weights were initialized in the range $[-0.25, 0.25]$ and $[-1.0, 1.0]$ respectively.

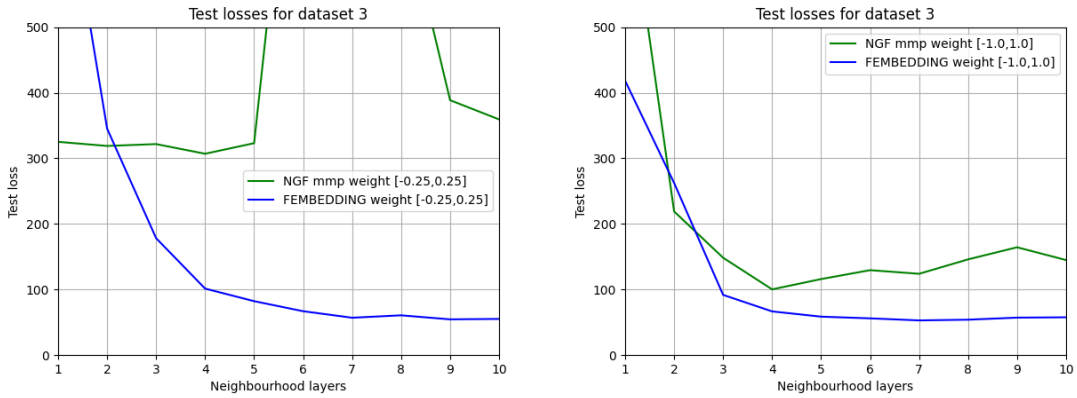


Figure 4.6: Test loss as a function of neighbourhood size for dataset 3 with different weight scales (left: [-0.25, 0.25], right: [-1.0, 1.0]).

As can be seen in the figure, the loss levels are in general higher than for datasets 1 and 2. It seems that an increased complexity is making it harder for the models to separate between different geometries. This might also be dependent on that the embedding length is not large enough. The embedding length used for dataset 3 was 256 while the largest model to encode contained more than 6000 elements. Again the FEMBEDDING method performs better than the NGF method and for this dataset the FEMBEDDING method consistently show a benefit of including an increased neighbourhood depth when creating the embeddings.
In Table 4.1 the three samples from the test sets in dataset 3 having the largest and smallest error compared with the true value for each target are listed for the FEMBEDDING method where 10 neighbourhood hops was considered when creating the embedding.

Table 4.1: Sample of the three most and least correctly predicted targets using the FEMBEDDING method.

Target	Id	True	Predicted	Error [%]
Max load	section2_555	142.850998	142.849854	0.000801
Max load	section2_990	170.145004	170.147644	0.001551
Max load	section2_299	149.714996	149.712051	0.001967
Max load	section4_102	69.113098	66.846153	3.280051
Max load	section1_192	119.416000	115.011078	3.688720
Max load	section1_41	92.486099	96.404068	4.236279
Mean load	section1_709	9.839859	9.839802	0.000582
Mean load	section1_354	20.143547	20.144085	0.002670
Mean load	section4_720	9.771930	9.772328	0.004079
Mean load	section1_632	23.376562	31.898949	36.456971
Mean load	section1_812	17.404263	24.252714	39.349278
Mean load	section4_102	7.403901	12.868861	73.811911
Max def	section4_107	465.670013	465.677277	0.001560
Max def	section4_329	456.589996	456.610687	0.004532
Max def	section4_565	251.339996	251.366394	0.010503
Max def	section4_94	85.945	114.159622	32.828696
Max def	section4_373	75.209	101.327560	34.727973
Max def	section1_912	94.777	140.206757	47.933313

From Table 4.1 we can see that the prediction of the maximum load seems to be the easiest and that the prediction of the mean load and max deformation seems harder.
In order to find out if there are other metrics then the MSE that could be used as target cost function when training the models and possibly reduce the errors in the test set, two other metrics were evaluated, namely the Mean Absolute Error (MAE) and the Root Mean Squared Error (RMSE). 10 trials was done on dataset 3 using the FEMBEDDING method with 10 neighbourhood hops included when creating the embeddings. In Table 4.2 the result from this study can be seen.

Table 4.2: Test loss when evaluating models trained using different cost function metrics.

Trial	MSE-loss	MAE-loss	RMSE-loss
1	48.26	5.38	10.52
2	54.73	5.66	10.32
3	51.97	5.28	10.22
4	57.53	5.49	10.35
5	65.60	5.56	10.35
6	51.43	6.05	10.35
7	48.32	5.50	10.02
8	54.89	5.73	10.59
9	56.73	5.69	10.15
10	55.18	5.35	9.85
Average:	54.46	5.57	10.27

From Table 4.2 we can see that in comparison with the square-root of the MSE metric (to get the same units), the MAE metric in general gives a lower loss and that the RMSE in general gives a higher loss. To get a better comparison the mean and standard deviation of the sample error was calculated for the 10 trials and the result can be seen in Table XX.

Table 4.3: Mean and standard deviation of target sample errors for different cost function metrics.

Cost function metric	Target	Sample error mean [%]	Sample error standard deviation
MSE	Max load	0.60	0.56
MAE	Max load	0.42	0.50
RMSE	Max load	0.41	0.42
MSE	Mean load	5.38	6.17
MAE	Mean load	4.25	5.57
RMSE	Mean load	4.68	5.74
MSE	Max def	3.02	4.20
MAE	Max def	2.75	4.02
RMSE	Max def	3.00	4.53

As can be seen in Table 4.3 the MAE metric perform better and has both lower mean error and lower error standard deviation for the two targets having the largest levels of error.
To get an idea of the error distribution for the three metrics sample histograms of the sample error is shown in Figure 4.7 for the three targets.

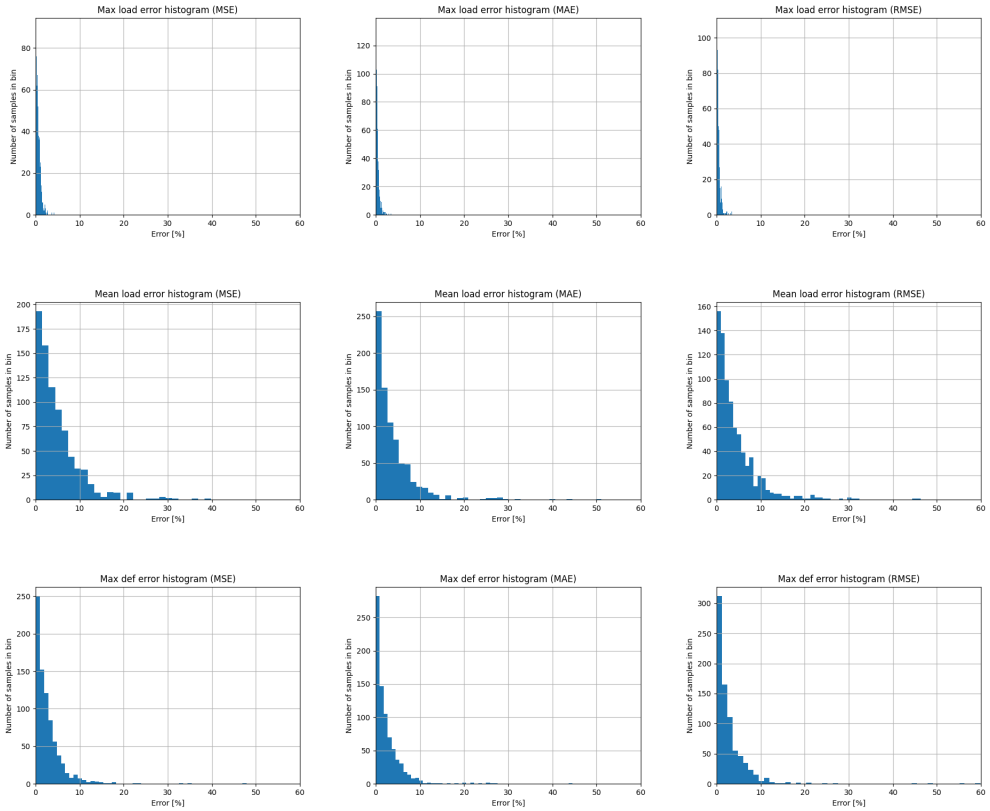
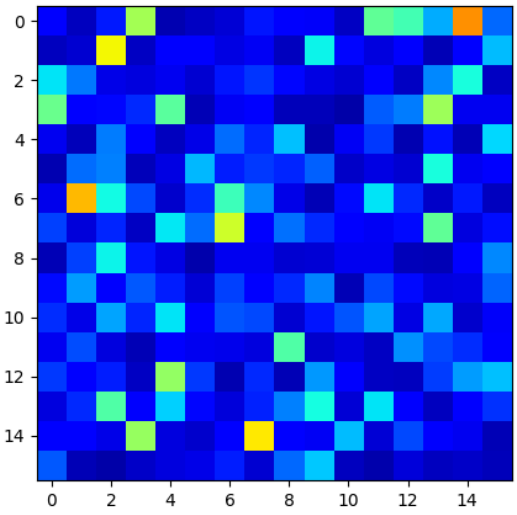
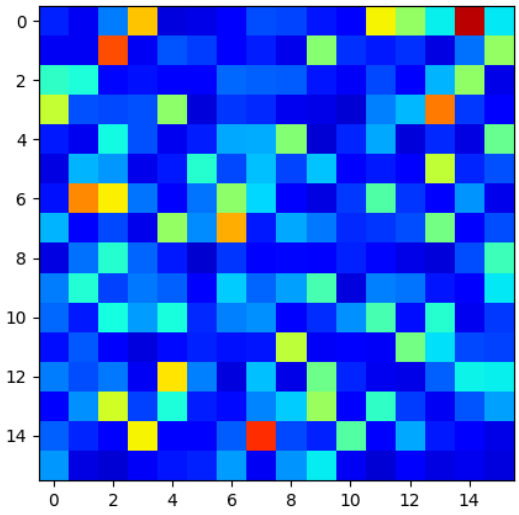


Figure 4.7: Histograms of target errors. The top row shows sample error distribution of the maximum load predictions while the middle and last row show the corresponding error distributions for the mean load and maximum deformation targets respectively.

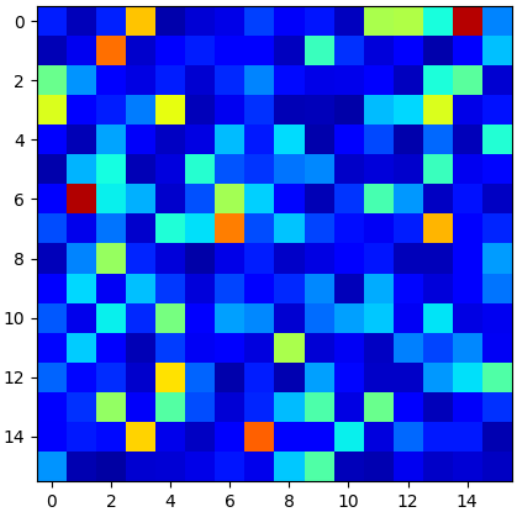
From Figure 4.7 we can see that the vast majority of the samples in the test set have errors less than 10% for any of the targets.
Since the embeddings has been treated as images in this project, a few different embeddings using the FEMBEDDING method from dataset 3 are shown in Figure 4.8.



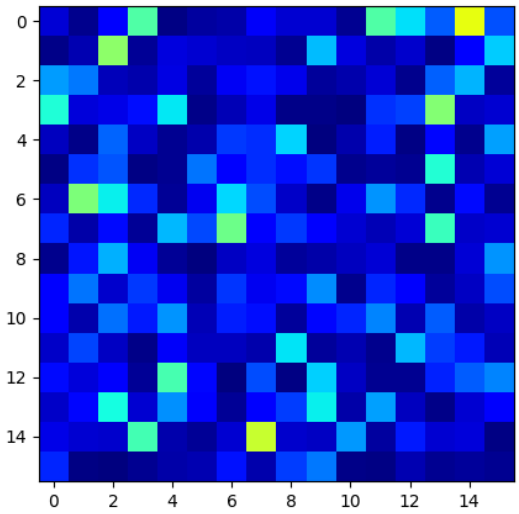
(a) Section 1, sample 55.



(b) Section 2, sample 55.



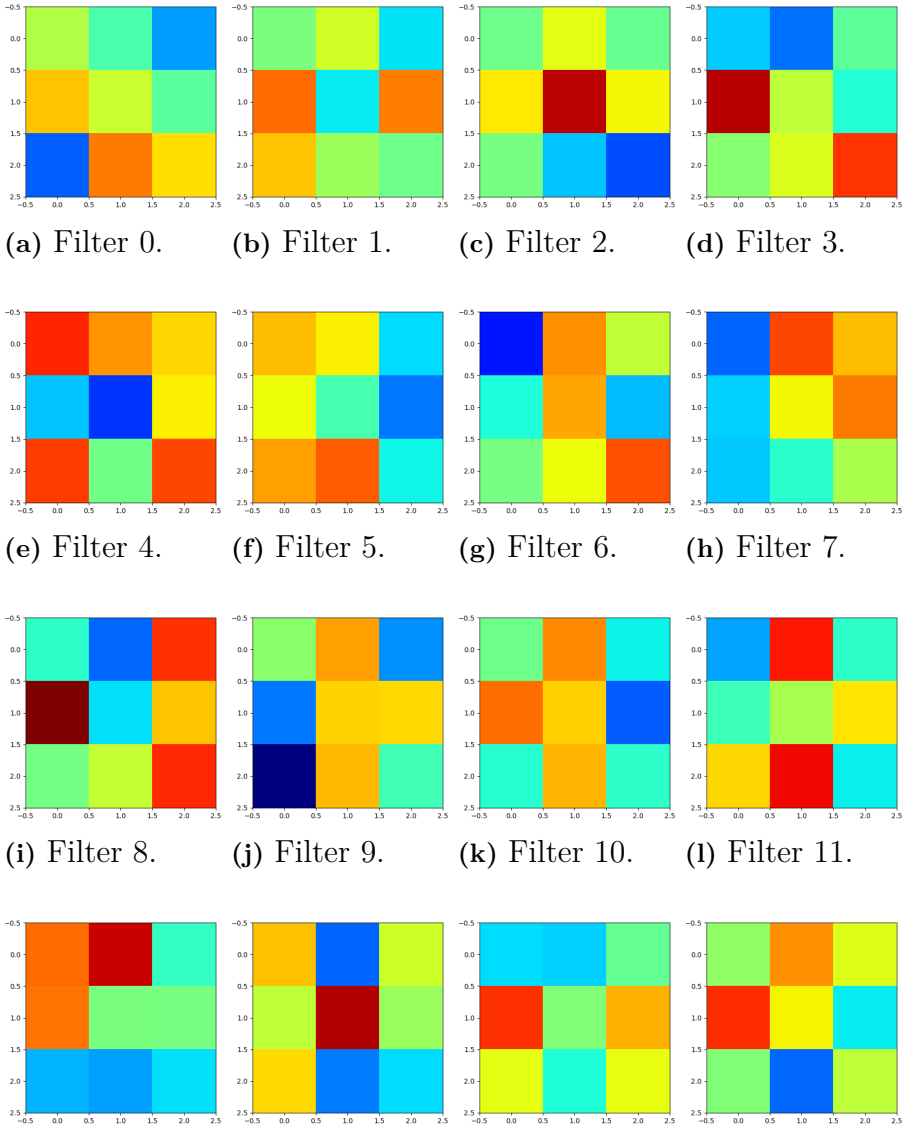
(c) Section 3, sample 55.



(d) Section 4, sample 55.

Figure 4.8: FEMBEDDING images from dataset 3.

A closer look at these images reveals that there are patterns, however subtle. Notice for example that the embedding image for Section 4 is in general darker then for the rectangular section geometries. Also it seems that the images for section 2 and 3 have in general a few red/brown pixels that positioned more towards the yellow/green colours. But what patterns are captured in the CNN for these images? As an attempt to answer that question the trained filters from the first CNN-layer has been plotted as images in Figure 4.9



(a) Filter 0. **(b)** Filter 1. **(c)** Filter 2. **(d)** Filter 3.

(e) Filter 4. **(f)** Filter 5. **(g)** Filter 6. **(h)** Filter 7.

(i) Filter 8. **(j)** Filter 9. **(k)** Filter 10. **(l)** Filter 11.

(m) Filter 12. **(n)** Filter 13. **(o)** Filter 14. **(p)** Filter 15.

Figure 4.9: CNN filter images from first CNN-layer with FEMBEDDING method from dataset 3.

What does the feature maps look like when applying these filters on an embedding image? In Figure 4.10 the feature maps are plotted as images when the embedding image from Section 1 sample 55 is passed through the first layer of the trained CNN.

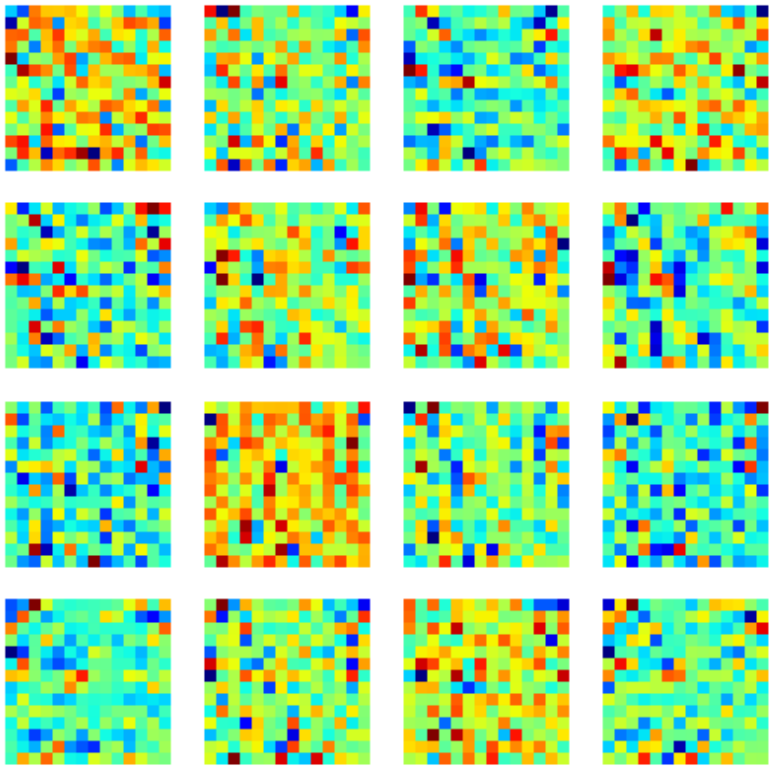
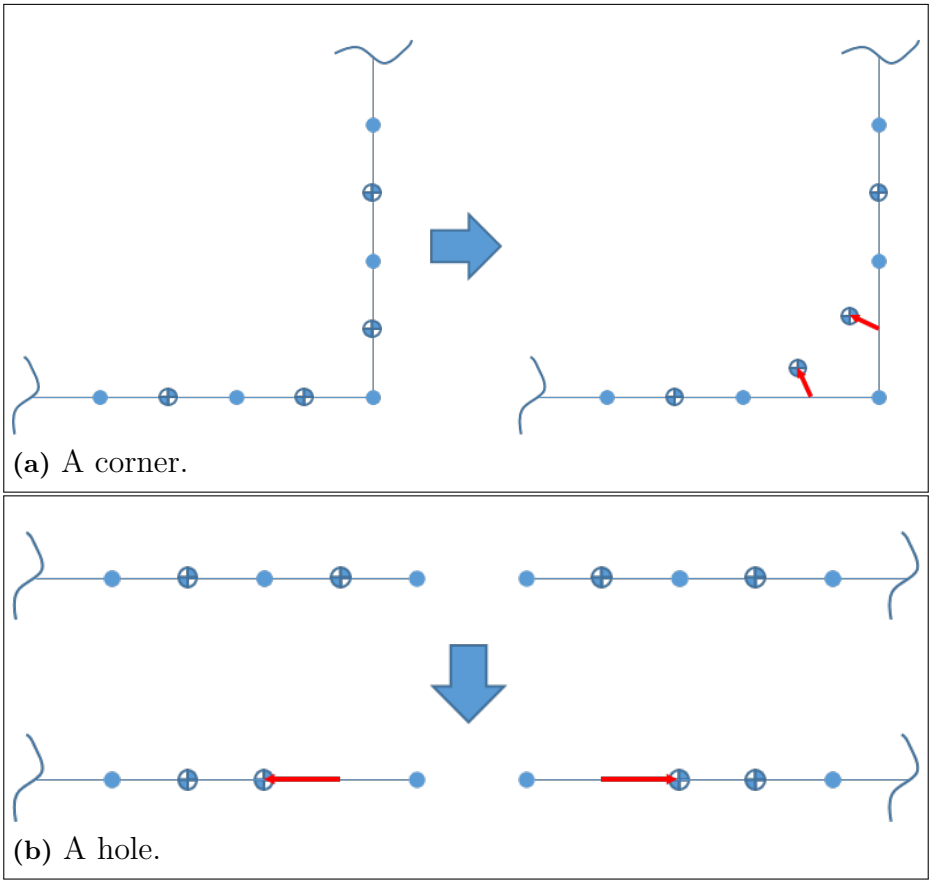


Figure 4.10: CNN feature map images from first CNN-layer with FEMBEDDING method from dataset 3.

From Figure 4.9 and Figure 4.10 we can see that it is very hard for the human eye to comprehend the complex patterns that the trained CNN use to predict result from unseen embeddings.
For the FEMBEDDING method, the initial motivator to convolve the difference between a neighbourhood and the next larger neighbourhood was that this will eventually converge to a vector of zeros. However, it turns out that this seems to be an efficient way of encoding geometry. Why is that the case? In Figure 4.11 an attempt to explain this in a 2D geometry is shown.



(a) A corner.

(b) A hole.

Figure 4.11: A 2D mesh and corresponding vertex feature values before and after a mean message pass. The red arrows indicate how the vertex feature values have changed between the states.

Notice how the vertex feature values change between states only for elements having neighbours where the mean of the coordinate values is different from the coordinate value of the element itself. Hence important details as holes and corners can be detected using the FEMBEDDING method.

5
Conclusion

The purpose of this project was to investigate the possibility to transform FE-graphs into image-like embeddings using only randomly initialized weights and to use Convolutional Neural Networks to explore these embeddings. Two graph convolutional methods were investigated for the creation of embedding images from FE-model graphs. The first one was the Neural Graph Fingerprint (NGF) method suggested in [5] and the second one was developed during this project, called the FEMBEDDING method, with inspiration from both [2] and [5].
From for example Figure 4.1, 4.3 and 4.5 we can see that the embeddings as created in this project are valuable already with randomly initialized weights and that it is possible to treat the embeddings as images and successfully train CNN's using those embedding images, i.e. with low test losses. From these Figures we can also draw the conclusion that FEMBEDDING-method, as it was designed in this project, is a better choice in comparison with the NGF-method when creating the embedding images, since it consistently performed better than the NGF method. From Figure 4.5 and Figure 4.6 we can see that for the more geometrically complex dataset 3, the FEMBEDDING method consistently show an increased value with increased neighbourhood depth taken into account when creating the embeddings.

5.1 Discussion

The three datasets of crash simulations developed during the project had a varying level of geometric complexity. As can be seen in Figure 4.1, 4.3 and 4.5 the general loss levels for the test sets increase for datasets with increased geometric complexity. This might indicate that for even larger geometric complexity it could be hard to use embeddings of complete FE-models (as was the case in this project) without training the weights of the embeddings.
An attempt was done to present the embeddings, filters and feature maps as images, but it seems those images are very abstract and it is not meaningful to think of these image-like embeddings as more than means to the usage of 2D CNN architecture which in general has proven to be an efficient way of training a Neural Network.
In [5] it was shown that the value of the embeddings increase with increasingly large neighbourhood taken into account when creating the embeddings. For the datasets used in this project, this could not be clearly shown when applying the NGF method, while there was a clearer tendency for the FEMBEDDING method. For the less geometrically complex datasets 1 and 2 it seems that if the weights are initialized with larger random values, less neighbourhood layers need to be included

when creating the embeddings. However, the lowest loss level seems to be roughly in the same range independent of weight scale, hence a certain neighbourhood depth is recommended. In general it seems that neighbourhood depth above 7-8 layers seems to give very little improvements for the studied datasets.
In Table 4.3 the sample error mean and standard deviation were compared when training using dataset 3 but with different metrics. It seems that the Mean Absolute Error (also known as L1 Loss) is a better choice for a multi target cost function. It appears that the most difficult targets to predict (mean load and max deformation in this project) are handled in a better way and both the mean error and the error standard deviation are improved when using MAE as metric.

5.2 Future work

In this project the embeddings were built up by summing layer wise contributions from increasingly larger neighbourhood. It has been suggested in literature [21] that these layer wise contributions instead could be concatenated. An idea worth investigating could be to evaluate this method in the setting of FE-models.
A natural next step on a principal level would probably be to include time dependency but to predict the true dynamics of a crash, contact between parts would also need to be predicted. Attempts were done on an early stage in this project to split the FE-models in a predefined number of sub-models where the geometry of the sub-models were encoded into an embedding while tracking the position of each sub-model. Something similar could be of interest for a future study. In the work at hand the purpose was to use only randomly initialized weights when creating the embeddings, but the result indicated that for more complex geometries, the loss levels increase. In future work it could therefore be of interest to investigate if training also the embedding weights would lead to improvements for these more complex datasets.

Bibliography

[1] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957. URL: https://books.google.se/books?id=P%5C_XGPgAACAAJ.

[2] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

[3] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. "Spectral Networks and Locally Connected Networks on Graphs". In: (2014). arXiv: 1312.6203 [cs.LG].

[4] James Atwood and Don Towsley. "Diffusion-convolutional neural networks". In: *arXiv preprint arXiv:1511.02136* (2015).

[5] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P Adams. "Convolutional Networks on Graphs for Learning Molecular Fingerprints". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/f9be311e65d81a9ad8150a60844bb94c-Paper.pdf.

[6] Mikael Henaff, Joan Bruna, and Yann LeCun. "Deep Convolutional Networks on Graph-Structured Data". In: (2015). arXiv: 1506.05163 [cs.LG].

[7] Hanjun Dai, Bo Dai, and Le Song. "Discriminative Embeddings of Latent Variable Models for Structured Data". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, June 2016, pp. 2702–2711. URL: http://proceedings.mlr.press/v48/daib16.html.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org/.

[9] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. "Molecular graph convolutions: moving beyond fingerprints". In: *Journal of computer-aided molecular design* 30.8 (2016), pp. 595–608.

[10] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. "Learning Convolutional Neural Networks for Graphs". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, June 2016, pp. 2014–2023. URL: http://proceedings.mlr.press/v48/niepert16.html.

[11] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. "Geometric Deep Learning: Going beyond Euclidean data". In: *IEEE Signal Processing Magazine* 34.4 (2017), pp. 18–42. DOI: 10.1109/MSP.2017.2693418.

[12] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: (2017). arXiv: 1609.02907 [cs.LG].

[13] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch". In: *NIPS-W*. 2017.

[14] Ivanna Baturynska, Oleksandr Semeniuta, and Kristian Martinsen. "Optimization of Process Parameters for Powder Bed Fusion Additive Manufacturing by Combination of Machine Learning and Finite Element Method: A Conceptual Framework". In: *Procedia CIRP* 67 (2018). 11th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 19-21 July 2017, Gulf of Naples, Italy, pp. 227–232. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2017.12.204. URL: http://www.sciencedirect.com/science/article/pii/S2212827117311484.

[15] Tom Gulikers. *An Integrated Machine Learning and Finite Element Analysis Framework, Applied to Composite Substructures including Damage*. Dec. 2018. URL: http://resolver.tudelft.nl/uuid:615f2151-bcae-4e78-a2cb-3f1891a28275.

[16] Oleksiy Kononenko and Iryna Kononenko. "Machine learning and finite element method for physical systems modeling". In: *arXiv preprint arXiv:1801.07337* (2018).

[17] Liang Liang, Minliang Liu, Caitlin Martin, and Wei Sun. "A deep learning approach to estimate stress distribution: a fast and accurate surrogate of finite-element analysis". In: *Journal of The Royal Society Interface* 15.138 (2018), p. 20170844. DOI: 10.1098/rsif.2017.0844. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsif.2017.0844. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsif.2017.0844.

[18] German Capuano. "Smart Finite Elements: An application of Machine Learning to Reduced-Order Modeling of Multi-Scale Problems". PhD thesis. Georgia Institute of Technology, 2019.

[19] Matthias Fey and Jan E. Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[20] Ali Madani, Ahmed Bakhaty, Jiwon Kim, Yara Mubarak, and Mohammad R. K. Mofrad. *Bridging Finite Element and Machine Learning Modeling: Stress Prediction of Arterial Walls in Atherosclerosis*. 084502. May 2019. DOI: 10.1115/1.4043290. eprint: https://asmedigitalcollection.asme.org/biomechanical/article-pdf/141/8/084502/6390325/bio_141_08_084502.pdf. URL: https://doi.org/10.1115/1.4043290.

[21] Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Ben Chamberlain, Michael Bronstein, and Federico Monti. "SIGN: Scalable Inception Graph Neural Networks". In: (2020). arXiv: 2004.11198 [cs.LG].

[22] Zhenguo Nie, Haoliang Jiang, and Levent Burak Kara. "Stress field prediction in cantilevered structures using convolutional neural networks". In: *Journal of Computing and Information Science in Engineering* 20.1 (2020).

[23] Oscar J. Pellicer-Valero, María José Rupérez, Sandra Martínez-Sanchis, and José D. Martín-Guerrero. *Real-time biomechanical modeling of the liver using Machine Learning models trained on Finite Element Method simulations*. 2020. DOI: https://doi.org/10.1016/j.eswa.2019.113083. URL: http://www.sciencedirect.com/science/article/pii/S0957417419308000.

[24] Biaojie Yan, Rui Gao, Pengchuang Liu, Pengcheng Zhang, and Liang Cheng. "Optimization of thermal conductivity of UO2–Mo composite with continuous Mo channel based on finite element method and machine learning". In: *International Journal of Heat and Mass Transfer* 159 (2020), p. 120067. ISSN: 0017-9310. DOI: https://doi.org/10.1016/j.ijheatmasstransfer.2020.120067. URL: http://www.sciencedirect.com/science/article/pii/S0017931020330039.

[25] *Convolution*. https://en.wikipedia.org/wiki/Convolution. Accessed: 2020-11-05.

[26] Mohit Deshpande. *Introduction to Convolutional Neural Networks for Vision Tasks*. Accessed: 2020-11-05. URL: https://pythonmachinelearning.pro/introduction-to-convolutional-neural-networks-for-vision-tasks/.