Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Structural Engineering

Elias Kiær
Herman Liabø

Application of Graph Neural Networks to Static and Dynamic Structural Analysis

Master's thesis in Civil and Environmental Engineering
Supervisor: Associate Professor Marcin Luczkowski
Co-supervisor: Dr. techn. Konstantinos Gavriil
June 2023

NTNU
Norwegian University of
Science and Technology

Elias Kiær
Herman Liabø

Application of Graph Neural Networks to Static and Dynamic Structural Analysis

Master's thesis in Civil and Environmental Engineering
Supervisor: Associate Professor Marcin Luczkowski
Co-supervisor: Dr. techn. Konstantinos Gavriil
June 2023

Norwegian University of Science and Technology
Faculty of Engineering
Department of Structural Engineering

Department of Structural Engineering
Faculty of Engineering
NTNU- Norwegian University of Science and Technology

ACCESSIBILITY	OPEN

MASTER THESIS 2023

SUBJECT AREA:	DATE:	NO. OF PAGES:
STRUCTURAL ENGINEERING	08/06/2023	xii + 94

TITLE:

Application of Graph Neural Networks to Static and Dynamic Structural Analysis

Anvendelse av Grafiske Nevrale Nettverk for Statisk og Dynamisk Konstruksjonsanalyse

BY:

Elias Andresen Kiær
Herman Liabø

SUMMARY:
This thesis begins with a focus on the development and implementation of a framework for both static and dynamic Finite Element Analysis (FEA) as plugins to the Rhino Grasshopper parametric environment, leveraging Euler-Bernoulli beam theory and the Newmark average acceleration method. The main part of the work done in this thesis embarks on the potential of using machine learning tools, specifically Graph Neural Networks (GNNs), for traditional structural engineering tasks.

Motivated by the often labour-intensive and resource-demanding nature of creating a Finite Element Analysis (FEA) model, this thesis aims to explore more efficient approaches for situations in structural behaviour analysis where computational efficiency is preferred over absolute precision. In such scenarios, well-designed neural network models, given enough training, can serve as potent solutions to address these challenges.

Showcased in this work is the process of developing the structural analysis framework Velociraptor, including both FEA tools and Artificial Intelligence (AI) tools. It encompasses the development and implementation of an FEA framework for 2D and 3D structures, referred to as Velociraptor2D (V2D) and Velociraptor3D (V3D), and two Neural Network (NN) models - ArchNN and TrussNN - which are implemented utilizing GNNs. Specifically, ArchNN is designed to predict nodal moments in arches, whereas TrussNN is designed to predict nodal time histories for truss bridges.

The V2D and V3D exhibit robust performance when benchmarked against Robot Structural Analysis and Karamba3D. Similarly, the predictive accuracy of ArchNN and TrussNN commendably matches the output from V2D and V3D in many cases. ArchNN and TrussNN demonstrate the potential of using GNNs for structural engineering analysis tasks.

The collaboration between the FEA Grasshopper plugins and the NN models opens the potential for a comprehensive structural analysis framework, incorporating FEA and AI, in the parametric environment. In this setup, NN plugins provide faster computations, whereas FEA plugins furnish benchmark results for verification and reliability. Herein, the FEA trains and tests the NN models, thereby enhancing the AI applicability over time to a more diverse range of structures.

RESPONSIBLE TEACHER: Associate Professor Marcin Luczkowski
SUPERVISOR(S): Associate Professor Marcin Luczkowski and Dr. techn. Konstantinos Gavriil

CARRIED OUT AT: Department of Structural Engineering, Norwegian University of Science and Technology

Preface

Our Master of Science Degree in Structural Engineering at the Norwegian University of Science and Technology comes to a close with the submission of this thesis. Our 5 years of studying have been rewarding and frustrating, fun and boring, inspiring and annoying, and short but at the same time long.

We express our deep gratitude to NTNU for providing a robust education, and opportunities for foreign exchange in Torino and San Diego.

Our gratitude is extended to our supervisor Marcin Luczkowski for his guidance, inspiration and involvement in this thesis which was formed over a beer in Tokyo. To our co-supervisor, Konstantinos Gavrill, thank you for sharing your knowledge in machine learning and providing the thought process of a mathematician.

Lastly, thank you to friends and family for your support.

“Innovation is taking two things that already exist and putting them together in a new way.”

- Tom Freston

Abstract

This thesis begins with a focus on the development and implementation of a framework for both static and dynamic Finite Element Analysis (FEA) as plugins to the Rhino Grasshopper parametric environment, leveraging Euler-Bernoulli beam theory and the Newmark average acceleration method. The main part of the work done in this thesis embarks on the potential of using machine learning tools, specifically Graph Neural Networks (GNNs), for traditional structural engineering tasks.

Motivated by the often labour-intensive and resource-demanding nature of creating a Finite Element Analysis (FEA) model, this thesis aims to explore more efficient approaches for situations in structural behaviour analysis where computational efficiency is preferred over absolute precision. In such scenarios, well-designed neural network models, given enough training, can serve as potent solutions to address these challenges.

Showcased in this work is the process of developing the structural analysis framework Velociraptor, including both FEA tools and Artificial Intelligence (AI) tools. It encompasses the development and implementation of an FEA framework for 2D and 3D structures, referred to as Velociraptor2D (V2D) and Velociraptor3D (V3D), and two Neural Network (NN) models - ArchNN and TrussNN - which are implemented utilizing GNNs. Specifically, ArchNN is designed to predict nodal moments in arches, whereas TrussNN is designed to predict nodal time histories for truss bridges.

The V2D and V3D exhibit robust performance when benchmarked against Robot Structural Analysis and Karamba3D. Similarly, the predictive accuracy of ArchNN and TrussNN commendably matches the output from V2D and V3D in many cases. ArchNN and TrussNN demonstrate the potential of using GNNs for structural engineering analysis tasks.

The collaboration between the FEA Grasshopper plugins and the NN models opens the potential for a comprehensive structural analysis framework, incorporating FEA and AI, in the parametric environment. In this setup, NN plugins provide faster computations, whereas FEA plugins furnish benchmark results for verification and reliability. Herein, the FEA trains and tests the NN models, thereby enhancing the AI applicability over time to a more diverse range of structures.

Sammendrag

Denne avhandlingen begynner med et fokus på utvikling og implementering av et rammeverk for statisk og dynamisk *Finite Element Analysis* (FEA) som plugins til Rhino Grasshopper sitt parametriske miljø. Dette er gjort ved utnyttelse av Euler-Bernoulli bjelketeori og dynamisk analyse ved numerisk integrasjon. Den siste delen av arbeidet gjort i denne avhandlingen begir seg ut på potensialet for å bruke maskinlærings-verktøy, spesifikt *Graph Neural Networks* (GNNs), for tradisjonelle oppgaver innen konstruksjonsteknikk.

Motivert av den ofte arbeidsintensive og ressurskrevende naturen ved å lage en FEA modell, har denne avhandlingen som mål å utforske mer effektive tilnærminger for situasjoner i konstruksjonsmessig analyse der beregningseffektivitet foretrekkes fremfor absolutt presisjon. I slike situasjoner kan godt designede *Neural Network* (NN)-modeller, gitt nok trening, fungere som potente løsninger for å håndtere disse utfordringene.

Arbeidet i denne avhandlingen gir en detaljert fremstilling av utviklingen av Velociraptor, et omfattende rammeverk for konstruksjonsanalyse som integrerer både FEA verktøy og *Artificial Intelligence* (AI) -teknologier. Dette omfatter utvikling og implementering av et FEA-rammeverk for 2D- og 3D-konstruksjoner, referert til som Velociraptor2D (V2D) og Velociraptor3D (V3D), og to Neural Network (NN) modeller - ArchNN og TrussNN - som er implementert ved bruk av GNNs. Spesifikt er ArchNN designet for å forutsi nodemomenter i buer, mens TrussNN er designet for å forutsi nodale tidshistorier for fagverksbroer.

V2D og V3D viser solid ytelse når sammenlignet med Robot Structural Analysis og Karamba3D. På samme måte er den prediktive nøyaktigheten til ArchNN og TrussNN anerkjennelsesverdig sammenlignbar med resultatene fra V2D og V3D i mange tilfeller. ArchNN og TrussNN demonstrerer potensialet for å bruke GNNs for oppgaver innen konstruksjonsteknikk.

Samarbeidet mellom FEA Grasshopper-komponentene og NN-modellene som er utviklet åpner muligheten for et omfattende konstruksjonsanalyserammeverk, som inkluderer FEA og AI, i det parametriske miljøet. I dette oppsettet gir NN-komponenter raskere beregninger, mens FEA-komponenter leverer referanseresultater for verifisering og pålitelighet. Her trener og tester FEA NN-modellene, og øker dermed deres anvendelighet over tid til et mer variert spekter av strukturer.

Glossary

AAD	Algorithms Aided Design, creating digital models based on algorithms.
Activation function	Function inside each neuron that processes the input to create output.
AI	Artificial Intelligence, the simulation of human intelligence processes by machines, especially computer systems.
Backpropagation	Minimizing the loss function propagating backwards through the model
Batch Size	Number of training examples fed to the network at a time.
Epoch	One training iteration over the available data for the machine learning network.
FEA	Finite Element Analysis, using the Finite Element Method for engineering purposes.
FEM	Finite Element Method, discretizing problems into manageable parts for calculations.
Forward propagation	The process of a model prediction. Data propagates through the NN model from input to output.
GNN	Graph Neural Network, a neural network using graph representation.
Hidden layer	Columns of neurons between the input and output layer.
Hidden units	Hidden layer neurons.
Hyperparameters	Variables which determine the NN structure
Input layer	Column of neurons at the start of the network, receiving input data.
Label	Value predicted by the machine learning process.
Learning rate	Parameter controlling the length of the training steps.
Loss Function	Function calculating deviation between predicted value and label.
ML	Machine learning, a branch of AI which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy.
NN	Neural Network, a machine learning structure using neurons to learn patterns and relationships in data.
Output layer	Last column of neurons delivering predictions.
Weight	Parameter controlling the strength of connections between neurons.

Table of Contents

Preface iii

Abstract v

Sammendrag vi

Glossary vii

List of Figures x

List of Tables xi

List of Procedures xii

1 Introduction 1

1.1 Background 1

1.2 Research Goal 2

1.3 Structure of the Thesis 2

1.4 Software 3

2 FEM Theory 4

2.1 Introduction 4

2.2 Euler-Bernoulli Beam Theory 4

2.3 FEM Static 5

2.4 FEM Dynamic 12

3 Neural Networks Theory 17

3.1 Introduction 17

3.2 Fundamentals of Neural Networks 17

3.3 Graph Neural Networks 24

4 VelociraptorFEA: Static and Dynamic Beam Calculation 28

4.1 Velociraptor2D Method 28

4.2 Velociraptor3D Method 34

4.3 V3D and V2D Benchmark Tests 40

4.4 VelociraptorFEA Discussion 49

5 VelociraptorAI **50**

5.1 ArchNN Method 50

5.2 ArchNN Results 65

5.3 ArchNN Discussion 67

5.4 TrussNN Method 68

5.5 TrussNN Results 80

5.6 TrussNN Discussion 84

6 Discussion **86**

6.1 VelociraptorFEA 86

6.2 Neural Networks 87

6.3 Velociraptor Framework 88

6.4 Real Time Health Monitoring 89

7 Conclusion **91**

Bibliography **92**

Appendix **95**

A GitHub Repositories 95

B Videos 95

List of Figures

2.1 Example of assembly of element stiffness's 6

2.2 Beam element with nodal degrees of freedom. 7

2.3 12 degree of freedom 3D beam element. 9

2.4 Rayleigh damping shown as a contributions from the mass and stiffness of the structure . 14

3.1 Neuron with two inputs. 17

3.2 Model of a fully connected neural network [40]. 18

3.3 The normalized loss function landscape and a possible path 21

3.4 Showcasing how underfitting, good fitting and overfitting may look like. [1]. 21

3.5 Five different types of activation functions. 22

3.6 A graph and its components, vertices or nodes with edges. 25

3.7 A graph with corresponding tensor representation 25

3.8 How message passing, node to node, works. 26

3.9 Possible structure for a Graph neural network. 27

4.1 General workflow for The Velociraptor2D. 29

4.2 General workflow for The Velociraptor3D 34

4.3 Discretization process for the arch. 40

4.4 Discretization process for the arch with Velociraptor2D. 40

4.5 Validation of results of FEM solver in Robot. 41

4.6 Discretization process for the arch with Velociraptor3D. 41

4.7 3D truss bridge with deformed geometry. 42

4.8 Beam 44 shown in green. 43

4.9 Different bridges and their deformed geometry. 44

4.10 One 3D Gridshell structure, deformed scaled 1 in green colour. 45

4.11 Three different 3D Gridshell structures, undeformed and deformed. 46

4.12 Fixed beam 4m long beam, used for verifying the dynamic solver. 47

4.13 Time history for displacement in the z-direction. 47

4.14 Time history for displacement in x-direction 47

4.15 Time history for rotation in the y-direction 48

4.16 Comparison plot: Time history for the displacing V3D vs Robot structural analysis 49

5.1 Arch 50

5.2 The moment diagram of the arch shown in Figure 5.1. 51

5.3 *Colibri* component for generating datasets. 53

5.4 Comparison plots: LeakyReLU vs. ReLU. 57

5.5 Extended training performance: ReLU vs. LeakyReLU. 58

5.6 Comparison plots: average percent error for ReLU vs. Leaky ReLU. 59

5.7 Comparison plots: training epochs' time duration for ReLU vs. LeakyReLU 60

5.8 Graph showing train loss and test loss for L1 and MSE loss function during training process. 61

5.9 Comparison plots for models trained with the L1 and MSE loss functions. 61
5.10 Testing the influence of data size on the model's predicting accuracy. 62
5.11 MSE training and test loss for the ArchNN with early stopping implemented. 63
5.12 Average percentage test loss for the ArchNN with early stopping implemented. 63
5.13 Adding complexity: testing how adding convolutional layers affects learning. 64
5.14 Moment diagram, both predicted and calculated, for five arch structures drawn for 66
5.15 One example of a truss bridge used for training, undeformed geometry and deformed. . . . 68
5.16 Time History for node 10. 68
5.17 Approximation of two different time history functions 71
5.18 Training loss and test loss for the TrussNN 75
5.19 Comparison plot: training loss for models trained with ReLU vs. LeakyReLU. 76
5.20 Comparison plot: L1 test loss for models trained with ReLU vs. LeakyReLU. 76
5.21 Comparison plot: Prediction with LeakyRelU and ReLu. 77
5.22 Comparison plots: Mean squared error loss vs L1 loss. 78
5.23 Comparison plots: Prediction with Mean squared error loss function vs L1 test loss. 78
5.24 Retrieving the best performing model: Introducing early saving to the TrussNN training. . 79
5.25 TrussNN bridge with node numbering. 80
5.26 TrussNN predictions for lowest loss value and highest loss value plotted for a selection of nodes against the fitted damped sine wave and the Velociraptor3D displacement time histories. 81
5.27 Truss bridge with geometry different from the one TrussNN is trained on. 82
5.28 Time history plots for the bridge geometry shown in 5.27 for a selection of nodes. 83
7.1 Velociraptor structural analysis framework workflow. 91

List of Tables

4.1 Input and output for component AddCrossSection 30
4.2 Input and output for component CreateLoad 30
4.3 Input and output for component CreateSupport 30
4.4 Input and output for component CreateBeamElements 31
4.5 Input and output for component AssembleModel 31
4.6 Input and output for component SolverStatic 31
4.7 Input and output for component DynamicSolver 32
4.8 Input and output for component ToTextFile 32
4.9 Input and output for component ForceCheck 33
4.10 Input and output for component TimeHistory 33
4.11 Deconstructors within the V2D environment 33
4.12 Input and output for component AddCrossSection3D 35
4.13 Input and output for component CreateLoad3D 35
4.14 Input and output for component CreateSupport3D 35

4.15 Input and output for component CreateBeamElements3D 36
4.16 Input and output for component SetLocalAxis3D 36
4.17 Input and output for component AssembleModel3D 36
4.18 Input and output for component SolverStatic3D 37
4.19 Input and output for component DynamicSolver3D 37
4.20 Input and output for component ToTextFile3D 38
4.21 Input and output for component Checky 38
4.22 Input and output for component EdgeIndex 38
4.23 Deconstructors within the V3D environment 39
4.24 Results Truss Bridge 3d. 42
4.25 Comparing beam forces for beam 44. 43
4.26 Dimensions and maximum displacements for the bridges shown in Fig. 4.9. 44
4.27 Results gridshell 3d 45
4.28 Dimensions and maximum displacements for the gridshells shown in Fig. 4.11. 46
4.29 Validation of the Newmark method against the Eigenvalue problem. 48
5.1 PyTorch Geometric Data Object [37]. 51
5.2 Model testing: Comparison between Karamba3D, Velociraptor2D and ArchNN. 65
5.3 Model testing: ReLU vs. LeakyReLU 67
5.4 The TrussNN's three best and three worst predictions within the test dataset and the bridge's corresponding lengths and heights. 80
5.5 Node 9 labels and parameters for the lowest loss prediction and the highest loss prediction. 82

List of Procedures

2.1 Procedure for determining local axis vectors of the 3D beam element. 11
2.2 Procedure showing implementation of Newmark method 16
5.3 ArchNN class that defines the architecture of the model. 54
5.4 Runner function and print_info function. 55
5.5 Training method. 56
5.6 Testing method. 56
5.7 Procedure showing the method of obtaining the time history fitted function 70
5.8 TrussNN model architecture. 73
5.9 TrussNN's training and testing functions. 74

1 Introduction

This thesis was written for the Conceptual Structural Design Group (CSDG) at the Norwegian University of Science and Technology (NTNU), within the field of computational design. The field of conceptual design aims to reduce the gap between architects and engineers, exploring fields such as computational design, where the process of creating designs and building computational models closely intertwines.

1.1 Background

Computational design is often spoken about in the same sentence as parametric design, which is often done with visual programming languages such as Rhino's Grasshopper or Autodesk's Dynamo. Parametric design, also called *Algorithm-Aided Design* (ADD), is a design process where the structural elements are shaped by algorithmic processes rather than direct manipulation. This design method has seen a surge in popularity among structural design firms in recent years, due largely to its efficiency in making alterations to structural models. This effectiveness is rooted in the method's characteristic of being governed by parameters and rules, much like conventional code programs.

Implementing *Finite Element Analysis* (FEA) in a platform such as Grasshopper offers the advantage of instant results when changes are made to a model. This seamless interaction eliminates the need to alternate between modelling software and FEA software, thus streamlining the modelling process. There exist plugins that implement FEA in Grasshopper, Karamba3D is an example. However, implementing time integration methods for outputting time histories is something that hasn't been done, to the authors' knowledge. This introduces another way to quickly examine the behaviour of structures.

Conversely, employing FEA plugins within Grasshopper may result in a substantial slowdown in model performance when handling a high volume of elements. This can potentially inhibit the flexible nature of the parametric design process.

Implementing *Graph Neural Networks* (GNNs) for structural engineering problems has seen some research in the past, like the paper *Towards Reusable Surrogate Models: Graph-Based Transfer Learning on Trusses* written by Eamon Whalen and Caitlin Mueller on the use of Graph Surrogate Models, using GNNs, to predict displacement fields for trusses [47].

Harnessing the conjunction of GNNs' predictive capabilities, the straightforward representation of structural designs as graphs, and the adaptability of parametric models along with FEA plugins, introduces an efficient approach for training machine learning models capable of swift predictions. In certain situations, a broad understanding of the model's behaviour may take precedence over the need for precision, making the promptness of results for a parametric model more critical than their absolute accuracy.

1.2 Research Goal

"To explore the possibility of combining Finite Element Analysis and Artificial Intelligence in a structural analysis framework within a parametric environment."

The *Artificial Intelligende* (AI) components, powered by GNN models, would deliver swift real-time calculations during the model design process, while the FEA components would serve the purpose of training the AI components and verifying the results. This combined approach, aligned with the expressed goal, aims to optimize efficiency by facilitating seamless collaboration between FEA and AI techniques, all while ensuring comprehensive analysis and accurate results throughout the structural design process. The use of FEA tools would be reserved for periods of lesser design changes to prevent software slowdown. These tools could be employed during an overnight training process, significant model revisions, or for final result validation. By strategically utilizing FEA tools in such scenarios, the computational burden can be reduced, allowing for smoother interactions and faster response times during the iterative design phase.

1.3 Structure of the Thesis

This thesis is divided into two main parts - one focusing on Finite Element Analysis (FEA) and the other on *Neural Networks* (NN). Initially, the theory of the *Finite Element Method* (FEM) for both static and dynamic analysis of 2D and 3D beam elements is explained, laying the foundation for the development of *VelociraptorFEA*. Subsequently, the basis for NNs and Graph Neural Network (GNN) models to be developed is laid out by discussing the foundations of NNs and GNNs.

The VelociraptorFEA method chapter explains the functionality and composition of the components developed for the Grasshopper plugins, Velociraptor2D and Velociraptor3D. This chapter culminates with benchmark results comparing the plugins with the Grasshopper FEA plugin, Karamba 3D, and a well-established FEA software, Robot Structural Analysis.

For the *VelociraptorAI* method part, the focus is concentrated on testing the NN architecture of the two models - ArchNN and TrussNN. ArchNN discusses the process of predicting nodal moments in 20-node 2D arches. This includes generating data using Velociraptor2D, presenting these arches as graphs, building the model architecture, as well as testing and improving this architecture and the training process. In the end, the model predictions are compared to Velociraptor2D and Karamba3D, and some observations are made in a closing statement. TrussNN is given the task of predicting displacement time history plots in the z-direction for the 32 nodes of 3D truss bridges with a certain geometry. The time histories generated by Velociraptor3D contain 200 displacement values for each node, and these time history plots are therefore fitted to an eight-coefficient damped sine wave, which the TrussNN model is trained to predict. This process is laid out in the TrussNN section along with the same process implemented for the ArchNN section.

The final chapter of the thesis is dedicated to the discussion of the results obtained throughout the work.

1.4 Software

This chapter provides a brief overview of the various software utilized in this thesis, including the type of software and its specific applications within the thesis.

Rhinoceros 3D 7 and Grasshopper

Rhinoceros 3D 7 (Rhino) is a 3D development platform for modelling and rendering computer-aided designs. Grasshopper is visual a programming language that allows for parametric design within the Rhino environment [2]. This environment is used for the creation of the plug-ins V2D and V3D.

Karamba3D

Karamba3D is an FEA plug-in for Grasshopper. Karamba3D is an interactive, parametric engineering tool that allows you to perform quick and accurate FEA [14]. Karamba3D was used in the validation process of Velociraptor2D and Velociraptor3D.

Colibri 2.0.0

Colibri 2 is a plugin framework for Grasshopper and facilitates the iteration of all potential combinations of a series of inputs and compiles the resulting data into a CSV format, containing input and output values for each iteration. The software has been utilized for the generation of data intended for the NN part of the study [41].

Robot Structural Analysis

Robot Structural Analysis (Robot) is a complete FEA program delivered by Autodesk. The software has been employed to validate the self-developed plug-ins, particularly the dynamic solver. This validation process involved utilizing Newmark's method to compute time history plots, ensuring the accuracy and reliability of the plug-ins' performance [3].

Visual Studio 2022

Visual Studio 2022 is an IDE for cross-platform application. It has templates for plug-in design in Grasshopper and has therefore been used for C#-development of own plug-ins for Grasshopper[21].

Visual Studio Code

Visual Studio Code (VS Code) is a broad IDE for development operations. It has been used for Python programming for the creation of neural network models and for data processing [22].

PyTorch and PyTorch Geometric

PyTorch is an open source machine learning library within Python which provides a fully featured framework for building machine learning models. PyTorch Geometric is a library that allows for deep learning on graphs [26].

2 FEM Theory

2.1 Introduction

A common approach for numerically solving differential equations that occur in engineering and mathematical modelling is the Finite Element Method (FEM). The technique emerged to address the problems within structural mechanics. However, conventional topics of heat transfer, fluid flow, mass transport, electromagnetic potential and structural analysis are all typical problems that can be solved using FEM [30].

The finite element approach uses discretization of a structure or problem into smaller elements, where element properties are defined to model real-life problems. There are many different elements such as beam-, shell- or solid elements, that all have different strengths and weaknesses regarding accuracy and computational speed. There have been large catastrophes in the construction sector due to a lack of knowledge in FEM [32]. Therefore one has to be careful of the choices taken and have a strong understanding of what consequences they do further on in the analysis. This is extra important this day in age, where engineers often rely solely on computer programs [20].

As mentioned there are many different elements, this thesis concerns the beam element, which will be described in the following sections.

2.2 Euler-Bernoulli Beam Theory

Euler-Bernoulli beam theory is known as the classic beam theory and is one of the simpler elements within the finite element method. The theory is, as Bell writes, based on a few assumptions [4]:

Assumptions:

- Small displacements.
- Linear elastic and homogeneous material.
- Plane section perpendicular to the beam axis remain plane and perpendicular to the beam axis after deformation.
- The beam is prismatic. This means that the beam parameters; Young's modules, areal, and second moment of inertia stay the same throughout the beam.
- Normal stresses in the z-direction are neglected.

The beam's differential equation, Eq. 2.1, describes the relationship between the deflection of the beam *w* and the applied loads *q(x)*. Solving it thereby gives insights into the beams' structural behaviour and it forms the basis for analytical and numerical methods used in structural analysis with beams.

$$\frac{d^4w}{dx^4} + \frac{q(x)}{EI} = 0 \tag{2.1}$$

The differential equation is derived by firstly addressing the kinematics, Eq 2.2, and inserting this equation into Hookes law, Eq. 2.3.

$$\varepsilon = \frac{du}{dx} = z\frac{d\Theta}{dx} = -z\frac{d^2w}{dx^2} = -zw'' \tag{2.2}$$

$$\sigma = E\varepsilon = -zEw'' \tag{2.3}$$

By definition, the moment **M** can be extended to Eq. 2.4.

$$M = \int^A \sigma\, dA = -\int^A E\frac{d^2w}{dx^2}z^2, dA = -E\frac{d^2w}{dx^2}\int^A z^2 dA = -EI\frac{d^2w}{dx^2} \tag{2.4}$$

By demanding equilibrium for an infinitesimal small beam element, the two equilibrium equations Eq. 2.5 and Eq. 2.6 can be combined to Eq. 2.7:

$$Force : V + dv + qdx - V = 0 \rightarrow \frac{dV}{dx} = -q \tag{2.5}$$

$$Moment : dM - Vdx = 0 \rightarrow \frac{dM}{dx} = V \tag{2.6}$$

$$\frac{d^2M}{dx^2} = \frac{dV}{dx} \rightarrow \frac{d^2M}{dx^2} + q = 0 \tag{2.7}$$

Lastly by derivation of the latter formulation of Eq. 2.4 twice and inserting into Eq. 2.7, the beams differential equation, Eq. 2.1, is complete.

2.3 FEM Static

System Stiffness Relationship

Equation 2.8 represents the relationship between the system stiffness matrix **K**, the load vector containing applied forces **R**, and the displacement vector containing nodal displacements **r**. The equations presented in this section are sourced from [9].

$$\mathbf{R} = \mathbf{Kr} \tag{2.8}$$

The stiffness matrix can be derived using equation 2.9 where **C** is the material matrix and **B** is the strain-displacement matrix.

$$\mathbf{K} = \int_{V_e} \mathbf{B}^T \mathbf{C} \mathbf{B} \, dV \tag{2.9}$$

Equation 2.10 shows how to derive **B** from the matrix **N**, which contains the shape functions describing the beam's displacement field.

$$\mathbf{B} = \Delta \mathbf{N} \tag{2.10}$$

The matrix **K** captures the relationship between the applied loads and the corresponding displacements, reflecting the stiffness of the system and the distribution of strains throughout the structure. It is formed by summing the contributions of individual element stiffness matrices $\mathbf{k_i}$ in a manner that accounts for the connectivity of nodes. In this project, a direct method based on nodal numbering, as illustrated in Figure 2.1, is employed to assemble the matrix.

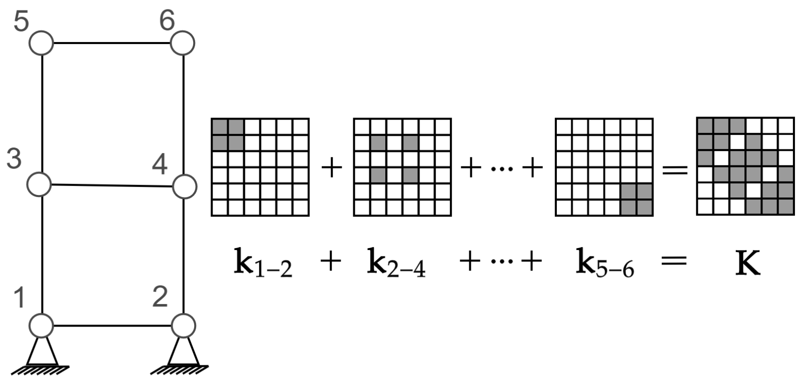


Figure 2.1: Example of assembly of element stiffness's

To account for the rotation of element local axis systems relative to the global axis system, a transformation matrix needs to be applied during the global assembly process, as depicted in Figure 2.1. This transformation can be achieved using Equation 2.11, where the local stiffness matrix $\mathbf{k_{local}}$ is multiplied by the inverse of **T** and then by **T**.

$$\mathbf{k_{global}} = \mathbf{T}^{-1} \cdot \mathbf{k_{local}} \cdot \mathbf{T} \tag{2.11}$$

Load matrix

The external loads acting on the system are represented by a vector called the load matrix. This matrix is constructed by breaking down the external forces acting on the elements into nodal or dispersed forces. Both types of forces are represented by matrices in the format [number of degrees of freedom (DOF) x 1].

There are different methods for translating distributed loads to the nodal point force matrix. In the case of uniformly distributed loads, load lumping is commonly used, where each node is assigned half of the total load acting on the element. For more complex loading shapes, the use of shape functions is typically employed. However, for this thesis, only nodal forces are considered, so there is no need for further theory or derivation of shape functions [4].

Boundary conditions

Applying boundary conditions in FEM is essential for accurate results. Fixing specific degrees of freedom restricts the structure from experiencing translations or rotations at those locations. This can be achieved by removing the corresponding rows and columns in the stiffness matrix and replacing them with diagonal entries of 1. Static condensing is also a widely used option [4].

Displacement

The displacements are determined by solving the system stiffness relationship, shown in Equation 2.12 [4].

$$\mathbf{r} = \mathbf{K}^{-1}\mathbf{R} \tag{2.12}$$

Forces

The forces in each beam element are retrieved by multiplying the displacements of the beam with its stiffness matrix $\mathbf{k}_{element}$, both defined in the local axis system of the beam. The local displacements are achieved by multiplying the transformation matrix $\mathbf{T}$ by the beam displacements defined in the global axis system, which are retrieved from the global displacement vector $\mathbf{r}$ [4].

2.3.1 2D formulations

The 2D-beam element to be considered has translational and rotational degrees of freedom at each node as Figure 2.2 shows.

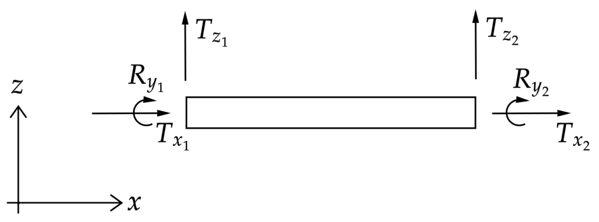


Figure 2.2: Beam element with nodal degrees of freedom.

The numbering of the degrees of freedom (DOF) is shown in Equation 2.13.

$$\mathbf{d}_e^\top = \begin{bmatrix} T_{x1} & T_{z1} & R_{y1} & T_{x2} & T_{z2} & R_{y2} \end{bmatrix} \tag{2.13}$$

The element stiffness matrix, Eq. 2.14, for the two noded Euler-Bernoulli beam element with three DOF at each node is as following[4]:

$$\mathbf{k_e} = \begin{bmatrix} \frac{EA}{L} & 0 & 0 & -\frac{EA}{L} & 0 & 0 \\ 0 & \frac{12EI}{L^3} & -\frac{6EI}{L^2} & 0 & -\frac{12EI}{L^3} & -\frac{6EI}{L^2} \\ 0 & -\frac{6EI}{L^2} & \frac{4EI}{L} & 0 & \frac{6EI}{L^2} & \frac{2EI}{L} \\ -\frac{EA}{L} & 0 & 0 & \frac{EA}{L} & 0 & 0 \\ 0 & -\frac{12EI}{L^3} & \frac{6EI}{L^2} & 0 & \frac{12EI}{L^3} & \frac{6EI}{L^2} \\ 0 & -\frac{6EI}{L^2} & \frac{2EI}{L} & 0 & \frac{6EI}{L^2} & \frac{4EI}{L} \end{bmatrix} \tag{2.14}$$

The transformation matrix gathered from [9] is defined as:

$$T = \begin{bmatrix} c & s & 0 & 0 & 0 & 0 \\ -c & -s & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & c & s & 0 \\ 0 & 0 & 0 & -c & -s & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.15}$$

Where:

$$\mathbf{c} = \cos\theta, \ \mathbf{s} = \sin\theta \tag{2.16}$$

For a single beam element, the nodal force matrix will have forces directly placed into its correlating place based on which node and DOF it is acting on, as seen in Eq. 2.17:

$$\mathbf{R_e^0}^\top = \begin{bmatrix} F_{x1} & F_{z_1} & M_{y_1} & F_{x_2} & F_{z_2} & M_{y_2} \end{bmatrix} \tag{2.17}$$

2.3.2 3D formulations

The 2D beam element can be expanded to 3D, by including out-of-plane translation and rotation. The beam element now becomes a 12 DOF element shown in figure 2.3.

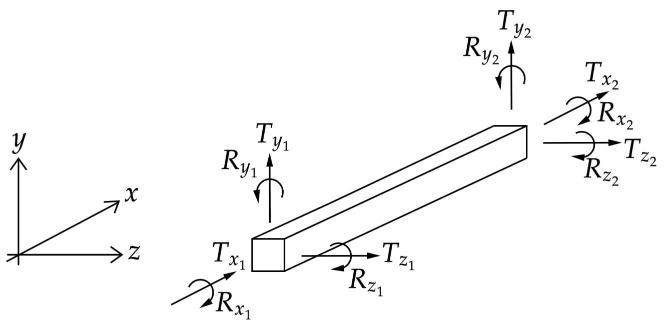


Figure 2.3: 12 degree of freedom 3D beam element.

The numbering of the degrees of freedom is shown in Equation 2.18.

$$\mathbf{d_e^T} = \begin{bmatrix} T_{x1} & T_{y1} & T_{z1} & R_{x1} & R_{y1} & R_{z1} & T_{x2} & T_{y2} & T_{z2} & R_{x2} & R_{y2} & R_{z2} \end{bmatrix} \tag{2.18}$$

The element stiffness matrix for the two noded Euler-Bernoulli beam element with six DOF at each node, Eq. 2.19 is as following[4]:

$$\mathbf{k_e} = \begin{bmatrix}
\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} & 0 & -\frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} \\
0 & 0 & \frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 & 0 & 0 & -\frac{12EI_y}{L^3} & 0 & -\frac{6EI}{L^2} & 0 \\
0 & 0 & 0 & \frac{GJ}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{GJ}{L} & 0 & 0 \\
0 & 0 & -\frac{6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & 0 & 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 \\
0 & \frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L} & 0 & -\frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} \\
-\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 & \frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{12EI_z}{L^3} & 0 & 0 & 0 & -\frac{6EI_z}{L^2} & 0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & -\frac{6EI_z}{L^2} \\
0 & 0 & -\frac{12EI_y}{L^3} & 0 & \frac{6EI_y}{L^2} & 0 & 0 & 0 & \frac{12EI_y}{L^3} & 0 & \frac{6EI}{L^2} & 0 \\
0 & 0 & 0 & -\frac{GJ}{L} & 0 & 0 & 0 & 0 & 0 & \frac{GJ}{L} & 0 & 0 \\
0 & 0 & -\frac{6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 & 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & 0 \\
0 & \frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} & 0 & -\frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L}
\end{bmatrix} \tag{2.19}$$

The 3D transformation matrix is implemented as outlined in the paper by Villanger and Åland [46], based on theory from [4]. The transformation matrix, **T**, is built up of the smaller matrix **t** stacked diagonally four times to account for all degrees of freedom.

$$\mathbf{T} = \begin{bmatrix} t & 0 & 0 & 0 \\ 0 & t & 0 & 0 \\ 0 & 0 & t & 0 \\ 0 & 0 & 0 & t \end{bmatrix} \tag{2.20}$$

Where **t** consists of the local axis vectors $\mathbf{x}_l$, $\mathbf{y}_l$ and $\mathbf{z}_l$, like shown in Equation 2.21.

$$\mathbf{t} = \begin{bmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{bmatrix} \tag{2.21}$$

The local vectors for the 3D beam element shown in Figure 2.3 with length l and start and end nodes $\{x_1, y_1, z_1\}$ and $\{x_2, y_2, z_2\}$, can be calculated as shown in Procedure 2.1.

For c_x and c_z equal to zero:

$$x_l = \{0,\ c_y,\ 0\}$$

$$y_l = \{-c_y * c_1,\ 0,\ s_1\}$$

$$z_l = \{c_y * s_1,\ 0,\ c_1\}$$

Else:

$$x_l = \{c_x,\ c_y,\ c_z\}$$

$$y_l = \left\{ \frac{-c_x * c_y * c_1 - c_z * s_1}{c_{xz}},\ c_{xz} * c_1,\ \frac{-c_y * c_z * c_1 + c_x * s_1}{c_{xz}} \right\}$$

$$z_l = \left\{ \frac{c_x * c_y * s_1 - c_z * c_1}{c_{xz}},\ -c_{xz} * s_1,\ \frac{c_y * c_z * s_1 + c_x * c_1}{c_{xz}} \right\}$$

Where:

$$c_x = \frac{x_2 - x_1}{l}$$

$$c_y = \frac{y_2 - y_1}{l}$$

$$c_z = \frac{z_2 - z_1}{l}$$

$$c_1 = \cos(\alpha)$$

$$s_1 = \sin(\alpha)$$

$$c_{xz} = \sqrt{c_x^2 + c_z^2}$$

Procedure 2.1: Procedure for determining local axis vectors of the 3D beam element.

Where α denotes the cross section's rotation around the local x-axis. Procedure 2.1 shows that the calculation of the local axis vectors differs when an element is parallel to the global y-axis.

For a single beam element, the nodal force matrix will have forces directly placed into its correlating place based on which DOF it is acting on, as seen in Eq. 2.22:

$$\mathbf{R^{0^T}} = \begin{bmatrix} F_{x1} & F_{y1} & F_{z1} & M_{x1} & M_{y1} & M_{z1} & F_{x2} & F_{y2} & F_{z2} & M_{x2} & M_{y2} & M_{z2} \end{bmatrix} \tag{2.22}$$

2.4 FEM Dynamic

An important area of study within structural engineering is dynamic analysis, where structures are analyzed over time. Dynamic analysis opens for time-dependent loading like earthquakes, wind and other vibrations. The equation of motion, Eq. 2.23, describes the forces produced in the structure as a result of an applied dynamic or static force. The *inertia force*, *damping force* and *spring force* are respectively products of the mass matrix **M** and acceleration vector $\ddot{\mathbf{x}}$, the damping matrix **C** and velocity vector $\dot{\mathbf{x}}$, and the stiffness matrix **K** and displacement vector **x**.

$$\mathbf{M}\ddot{\mathbf{x}} + \mathbf{C}\dot{\mathbf{x}} + \mathbf{K}\mathbf{x} = \mathbf{F}(t) \tag{2.23}$$

2.4.1 Mass matrix

The mass matrix describes how the mass is distributed in the different degrees of freedom in the system and is needed to solve dynamic problems. It can be derived by Equation 2.24 gathered from [9].

$$\mathbf{M} = \int_{V_e} \rho \mathbf{N}^T \mathbf{N} dV \tag{2.24}$$

For the 2D beam element, the mass matrix becomes as Equation 2.25 displays, where ρ is the density of the material, A is the area of the cross-section and L is the length of the beam element [24].

$$\mathbf{m_e} = \frac{\rho AL}{420}\begin{bmatrix} 140 & 0 & 0 & 70 & 0 & 0 \\ 0 & 156 & 22L & 0 & 54 & -13L \\ 0 & 22L & 4L^2 & 0 & 13L & -3L^2 \\ 70 & 0 & 0 & 140 & 0 & 0 \\ 0 & 54 & 13L & 0 & 156 & -22L \\ 0 & -13L & -3L2 & 0 & -22L & 4L^2 \end{bmatrix} \tag{2.25}$$

For the 3D beam element, the mass matrix becomes as Equation 2.26 displays. Here some new parameters are introduced such as r_x^2 and a, which is the equal to $\frac{I_x}{A}$ and $\frac{L}{2}$, respectively [24].

$$\mathbf{m_e} = \frac{\rho AL}{210}\begin{bmatrix} 70 & 0 & 0 & 0 & 0 & 0 & 35 & 0 & 0 & 0 & 0 & 0 \\ 0 & 78 & 0 & 0 & 0 & 22a & 0 & 27 & 0 & 0 & 0 & -13a \\ 0 & 0 & 78 & 0 & -22a & 0 & 0 & 0 & 27 & 0 & 13a & 0 \\ 0 & 0 & 0 & 70r_x^2 & 0 & 0 & 0 & 0 & 0 & -35r_x^2 & 0 & 0 \\ 0 & 0 & -22a & 0 & 8a^2 & 0 & 0 & 0 & -13a & 0 & -6a^2 & 0 \\ 0 & 22a & 0 & 0 & 0 & 8a^2 & 0 & 13a & 0 & 0 & 0 & -6a^2 \\ 35 & 0 & 0 & 0 & 0 & 0 & 70 & 0 & 0 & 0 & 0 & 0 \\ 0 & 27 & 0 & 0 & 0 & 13a & 0 & 78 & 0 & 0 & 0 & -22a \\ 0 & 0 & 27 & 0 & -13a & 0 & 0 & 0 & 78 & 0 & 22a & 0 \\ 0 & 0 & 0 & -35r_x^2 & 0 & 0 & 0 & 0 & 0 & 70r_x^2 & 0 & 0 \\ 0 & 0 & 13a & 0 & -6a^2 & 0 & 0 & 0 & 22a & 0 & 8a^2 & 0 \\ 0 & -13a & 0 & 0 & 0 & -6a^2 & 0 & -22a^2 & 0 & 0 & 0 & 8a^2 \end{bmatrix} \tag{2.26}$$

2.4.2 Damping matrix

The damping matrix, denoted as **C**, characterizes the energy dissipation from the system, which typically stems from various mechanisms. For physical buildings and structures, these mechanisms may include thermal effects resulting from elastic strain in the material, internal and external friction between construction components, and the opening and closing of micro-cracks in concrete. Due to the intricate and multifaceted nature of these damping phenomena, it becomes extremely challenging to develop precise mathematical models encompassing all these mechanisms. As a result, simplified methods have been devised to define the damping matrix, allowing for practical and manageable representation of damping effects in structural analysis[7]. In classical damping, specifically Rayleigh damping, the damping effect is assumed to be proportional to the mass and stiffness of the structure. This proportionality is expressed by the coefficients a_0 and a_1 in Equation 2.27 [7].

$$\mathbf{C} = a_0\mathbf{M} + a_1\mathbf{K} \tag{2.27}$$

Figure 2.4 illustrates Rayleigh damping as a combination of the dashed line and the dotted curve, representing the stiffness-proportional (ζ_{n_1}) and the inverse mass-proportional (ζ_{n_0}) contributions, respectively.

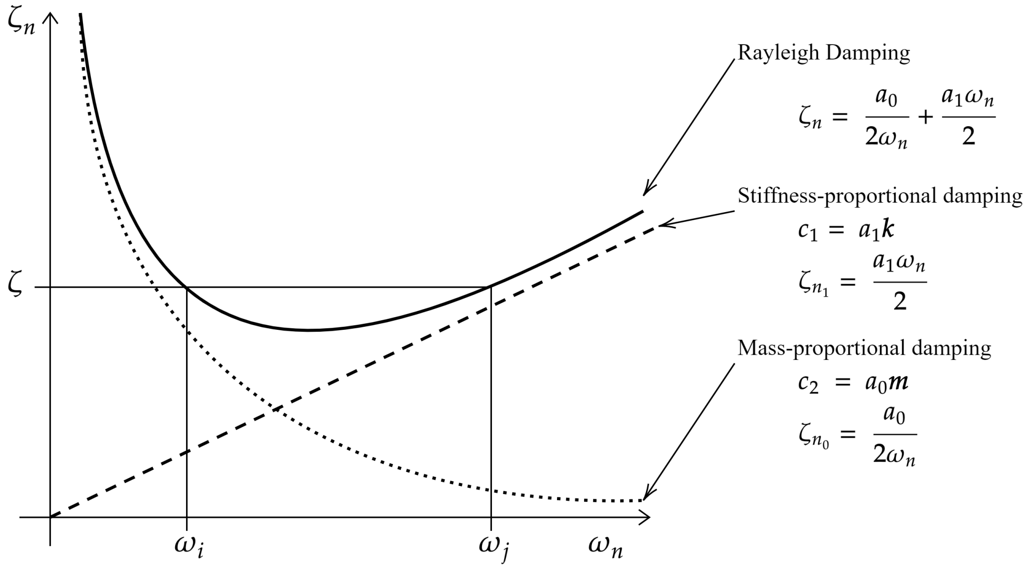


Figure 2.4: Rayleigh damping shown as a contributions from the mass and stiffness of the structure

By specifying the damping ratio ζ_{n_0} and ζ_{n_1} to be equal as ζ_n, and choosing natural frequencies ω_i and ω_i within a reasonable range, the coefficients a_0 and a_1 can be determined.

As a result, Equation 2.28 provides the coefficient that corresponds to the approximation, where ω_i and ω_i is the natural frequency for the ith and the jth mode.

$$a_0 = \zeta_n \frac{2 w_i w_j}{w_i + w_j}, \quad a_1 = \zeta_n \frac{2}{w_i + w_j} \tag{2.28}$$

The careful selection of the ith and jth modes is crucial for practical applications. For instance, in a dynamic analysis targeting the first six modes with a desire for uniform damping ratios, the damping ratios ζ_n for the first and next to last eigenfrequency should be specified. A review of Figure 2.4 shows that the damping ratios for the second, third, and fourth modes will then turn out slightly lower than the specified ζ_n, while the sixth mode is slightly higher. This is because the graph has lower values between the two specified modes and increases outside. The modes outside the two specified modes will have an increasing damping ratio in accordance with their frequency, essentially eliminating them from the response analysis [7].

Thereby, deriving the damping matrix from Equation 2.27 is relatively straightforward, which contributes to the widespread adoption of this formulation for damping.

2.4.3 The Eigenvalue Problem

This section provides a brief explanation of the eigenvalue problem and its solution for determining the natural frequencies and modes of a structure after [7]. The free vibrations of a structure in one of its natural vibration modes can be mathematically represented by Equation 2.29, which can be inserted into the equation of motion, Equation 2.23, resulting in Equation 2.30.

$$\mathbf{u}(t) = q_n(t)\phi_n \tag{2.29}$$

$$[-\omega_n^2 \mathbf{m}\phi_n + \mathbf{k}\phi_n] q_n(t) = \mathbf{0} \tag{2.30}$$

This equation has two possible solutions. The first solution is when $q_n(t) = 0$, indicating a structure at rest. The second solution is obtained when:

$$\omega_n^2 \mathbf{m}\phi_n = \mathbf{k}\phi_n \quad \rightarrow \quad [\mathbf{k} - \omega_n^2 \mathbf{m}]\phi_n = \mathbf{0} \tag{2.31}$$

Equation 2.31 represents the matrix eigenvalue problem, which reveals the natural frequencies and modes of the structure. Once again, there are two ways to satisfy the latter formulation of Equation 2.31. The first way is when $\phi_0 = 0$, which indicates no motion and is a trivial solution that is not of interest. The second way is given by:

$$det[\mathbf{k} - \omega_n^2 \mathbf{m}] = 0 \tag{2.32}$$

Equation 2.32 provides a solution for the structural problems addressed in this thesis since the mass **m** and stiffness **k** matrices are symmetric and positive definite for all cases.

2.4.4 Newmark's method

One method of solving the equation of motion is by the Newmark method, named after Nathan M. Newmark who developed it in the late 1950s [48]. The Newmark method is a numerical integration scheme that enables the calculation of the dynamic response of a structure subjected to a given set of time-dependent loads. It is a common method because it is relatively easy to implement and provides accurate results for a wide range of problems.

Procedure

The Newmark family of time-stepping methods are based on the two following equations:

$$\dot{u} = \dot{u}_i + [(1-\gamma)\Delta t]\,\ddot{u}_i + (\gamma\Delta t)\ddot{u}_{i+1} \tag{2.33}$$

$$u_{i+1} = u_i + (\Delta t)\dot{u}_i\left[(0.5-\beta)(\Delta t)^2\right]\ddot{u}_i + \left[\beta(\Delta t)^2\right]\ddot{u}_{i+1} \tag{2.34}$$

Which are derived from a Taylor's series expansion of the Equation of Motion, 2.23. Equations 2.23, 2.33 and 2.34 are solved by iteration for each time step, and for each DOF using matrix FEM notation [48]. The parameters β and γ define the variations of acceleration over a time step and determine the stability and accuracy of the algorithm [7]. The flowchart below, Procedure 2.2 describes the implementation of the Newmark method which was programmed in C# for this project.

$$t = 0$$

Given:

$$d_0, v_0$$

Initial calculation

$$a_0 = M^{-1}\left(f_0 - Cv_0 - Kd_0\right)$$

Predictor step

$$\tilde{d}_{n+1} = d_n + \Delta t v_n + \frac{1}{2}\left(1-2\beta\right)(\Delta t)^2 a_n$$

$$\tilde{v}_{n+1} = v_n + (1-\gamma)\Delta t a_n$$

Solution step

$$f^*_{n+1} = f_{n+1} - C\tilde{v}_{n+1} - K\tilde{d}_{n+1}$$

$$M^* = M + \gamma\Delta t C + \beta(\Delta t)^2 K$$

$$a_{n+1} = M^{*-1} f^*_{n+1}$$

Corrector step

$$d_{n+1} = \tilde{d}_{n+1} + \beta(\Delta t)^2 a_{n+1}$$

$$v_{n+1} = \tilde{v}_{n+1} + \gamma\Delta t a_{n+1}$$

Procedure 2.2: Procedure showing implementation of Newmark method

The most popular variant of the Newmark method is the *Average Acceleration Method*, where $\gamma = 1/2$ and $\beta = 1/4$. This makes the method unconditionally stable, securing convergence for all time increments [7].

3 Neural Networks Theory

3.1 Introduction

Machine learning (ML) is a branch within artificial intelligence (AI) that focuses on trainable models that can predict outcomes of certain problems [13]. This innovation has prompted new techniques and software in a variety of fields, such as robotics, big data analysis, image, and voice recognition. The innovation and development of machine learning have excelled in an exponential way and have evolved significantly since the 1980s and 1990s when there was a renewed interest in neural networks [11]. These algorithms were inspired by the human neural system and were able to handle more complex data. However, limitations to the size of datasets acted as a barrier until the development of deep learning algorithms emerged in the 2000s. Today, machine learning is a rapidly growing field with applications in a wide range of industries [17], especially in the last year with the development of large language models [47].

The selection of network architecture is determined by the specific task at hand, considering factors such as data characteristics and problem complexity. Understanding the problem domain plays a crucial role in designing an effective neural network. In this thesis, feed-forward neural networks will be discussed, as they serve as a foundation for the methodology discussed in Section 5. The theory then progresses toward graph neural networks, which will be the primary focus of investigation and development for the machine learning part of this research.

3.2 Fundamentals of Neural Networks

The neuron is the basic building block of a neural network. Figure 3.1 shows the functionality of a neuron with two inputs. The connections between neurons have a weight associated with them which determines the strength of the connection, thus how large the influence between them becomes. Firstly the inputs x_i, are multiplied by their corresponding weight, w_i.Then an extra parameter is added, the bias b, to help offset the output towards the negative or positive side [43]. Finally, the *activation function* f, is applied to produce the output [51].

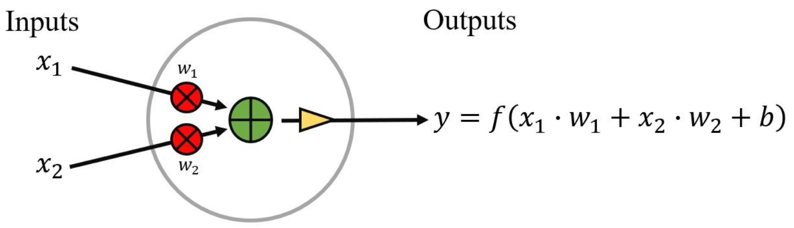


Figure 3.1: Neuron with two inputs.

A feed-forward neural network is formed by connecting multiple neurons in sequential layers. This type of architecture allows for the combination of a few to millions of neurons. The multi-layer neural network has one input layer, one output layer and often several hidden layers. Figure 3.2 shows what this may look like. The data propagates from the input layer and through each of these layers, hence the name feed-forward. [12].

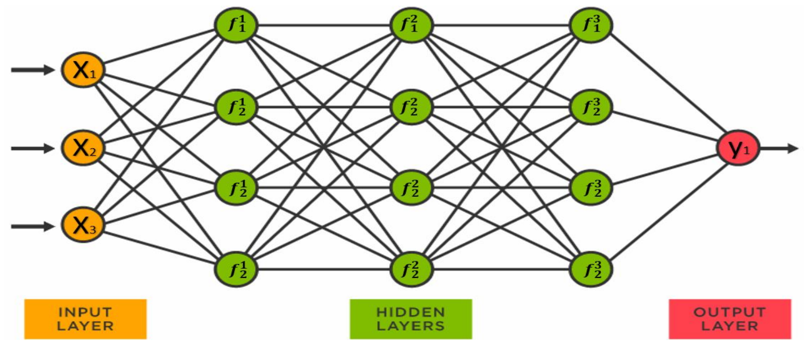


Figure 3.2: Model of a fully connected neural network [40].

In the given scenario above, each layer consists of fully interconnected neurons and the hidden layers have an identical number of neurons. The neurons can be partially connected, where the nodes only connect to some of the nodes in the following layer. Also, the number of nodes in the hidden layers may vary from layer to layer. The feed-forward neural networks are typically represented by combining several functions. For example, the network shown in Figure 3.2, can be represented by the function 3.1, where $f^{(1)}$ is the first layer and so on [12].

$$f(\mathbf{x}) = f^{(5)}(f^{(4)}(f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x}))))) \tag{3.1}$$

By tweaking the functions to make predictions on new data by changing its parameters, neural networks get the ability to handle and predict complex data. The goal of a feed-forward network is to predict a set of labels $\mathbf{y}$ by matching a function $f(x)$, to a hypothetical function $f^*(x)$. To achieve the best approximation, the feedforward network learns the best value of a set of parameters θ for a mapping $\mathbf{y} = f(x;\theta)$, where the function f maps an input $\mathbf{x}$ to a label, or set of labels, $\mathbf{y}$. These parameters θ consist of all the learnable parameters of the model, such as *weights* and *biases*. The process of learning these parameters is done in training [12]. The key terms and functionality of the neural network will be discussed in the following sections.

Training a feed-forward neural network and building the neural network learning algorithm, requires multiple design decisions such as choosing the optimizer, the loss function, the activation functions that compute the hidden layer values, and the overall architecture design of the model [12]. The model structure is designed by choosing the *hyperparameters*, the variables that are set prior to training. These include the number of *hidden units*, the number of *training epochs*, the *batch size*, and the *learning rate*. These play a crucial role in the model's ability to learn the parameters θ [29].

3.2.1 Learning Process

The learning process in neural networks involves training the model to learn from diverse training examples, **x**, and generate output that coincides with the target label, **y** [12]. The most commonly used learning method is *supervised learning*, where the program is provided with input data and corresponding labels[17]. For example, in an image classification task, the program is trained with images along with their corresponding labels indicating the object in the image.

In contrast, *unsupervised learning* aims to discover patterns and structures in data without the use of labelled information. The program explores the data and identifies hidden patterns or clusters. An example of unsupervised learning is clustering, where the program groups similar data points together based on their inherent characteristics.

Another learning technique is *reinforcement learning*, which leverages the use of rewards and punishment to improve learning by the algorithm. Through a feedback loop between the algorithm and its experience, the program strives to maximize long-term rewards [19] & [12]. An example of reinforcement learning is training a virtual game character to navigate through a maze. The character becomes better at navigation over time, through rewards for successfully reaching goals and penalties for colliding with obstacles.

Beyond these commonly discussed techniques, there are various other learning approaches available, each suited for specific problem types and data structures. The choice of learning technique depends on the nature of the problem and the specific goals of the task at hand.

During the training process, which will be further explained in the next chapter, a loss is calculated to evaluate the model's predictions. This is done using a *loss function* denoted as $\mathscr{L}(y,\hat{y})$. The purpose of the loss function is to measure the accuracy of the model's predictions by comparing the output $f(x)$ with the true labels **y** and quantifying the deviation. Various loss functions can be employed, as discussed in later sections.

After the loss is calculated, a critical step called *backpropagation* is performed. It involves computing the gradients of the loss function using gradient descent and then updating the weights between neurons in the network. Backpropagation is a fundamental process that significantly contributes to the success and convergence of neural networks. The primary objective of backpropagation is to minimize the loss function in subsequent iterations, allowing the network to progressively learn and make more accurate predictions [51]. The model's updated parameters, denoted as $\Delta\theta$, can be represented as shown in Equation 3.2 [17].

$$\Delta\theta^i = -\alpha \frac{\partial \mathscr{L}(y,\hat{y})}{\partial \theta^i} \tag{3.2}$$

In the equation, α represents the learning rate, and θ encompasses relevant parameters such as weights and biases. The learning rate controls, in simple terms how fast the network learns by its mistakes, or how strong it follows the gradient calculated through the backpropagation [17].

The learning rate and other hyperparameters are decided by a process of trial and error during training, as well as by comparing the model objective to other models [29].

3.2.2 Training Process

The training of a neural network is an iterative process in which data is continuously supplied to the model. By processing this data, the model gradually refines its internal parameters, learning and adapting to the inherent patterns in the training dataset [42].

Before the training process can start, the network's weights and biases, which are the parameters to be adjusted during the training process, are usually initialized with random values. This is an essential step before training can start [6].

After initialization, the actual training process begins. Each individual piece of data, or batch of data, is fed into the network. The input data then goes through a series of computations known as *forward propagation* where the data propagates through the network's layers, starting at the input layer and ending at the output layer. Within each neuron, the inputs are multiplied by their associated weights, and the results are summed. Then, a bias term is added to this sum. The resulting value is passed through an activation function, which determines the output that gets sent to the next layer. The activation function's role and its importance are discussed in the next section. When the input has propagated through all the layers, the network has made a prediction based on the current values of its parameters [6].

Next, a loss function is used to calculate the loss, or error, between the prediction and the input's label. This loss quantifies how far off the prediction was, and would ideally be zero. Now that the model has a measurement of its performance, backpropagation is performed. As explained in the previous section, in backpropagation, the network adjusts its weights and biases in a direction that will reduce the overall loss. In the training process, this is often done with an optimizer using gradient descent [42].

Finally, these steps; forward propagation, loss calculation and backpropagation are executed on each data batch until the entire training set has been trained on. This process is called one epoch and is repeated multiple times. This iterative process allows the network to gradually learn from the data, improving its predictions over time. The network finishes its training when it reaches a certain pre-defined prediction or until a predetermined number of epochs has been completed [42].

It is important that the model is trained on a wide variety of training examples, to ensure generalization in the model. This variability will also be beneficial by helping the model escape local minima and explore more of the loss landscape. Figure 3.3 displays a possible loss landscape, and a possible path for the loss to take during training.[15].

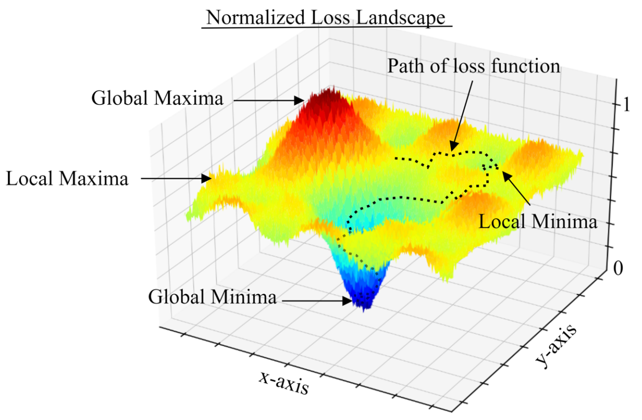


Figure 3.3: The normalized loss function landscape and a possible path for the loss function during training.

The direction and length of the steps in the path of the loss are determined by the calculated gradients and the learning rate. A low learning rate promotes stable learning but may lead to slow convergence or non-optimal solutions. In such cases, the mapping might get trapped in a local minimum or struggle to overcome a saddle point due to weak gradients. On the other hand, a high learning rate can cause an unstable network that overshoots the optimal mapping due to large gradients, although it may result in faster convergence.

While training a model, it is important to perform evaluations using a separate test, or validation, dataset. This is because, during the training process, a phenomenon known as overfitting can occur. Overfitting happens when the network becomes excessively adapted to the training data, finding non-existing trends or noise. This is often the result of running too many training epochs or constructing a network with an overly large number of neurons. Consequently, even as the network's training loss continues to decrease, the test loss may begin to increase, suggesting the network's diminishing ability to generalize from the training set to unseen data. Figure 3.4 illustrates a typical manifestation of overfitting [12].

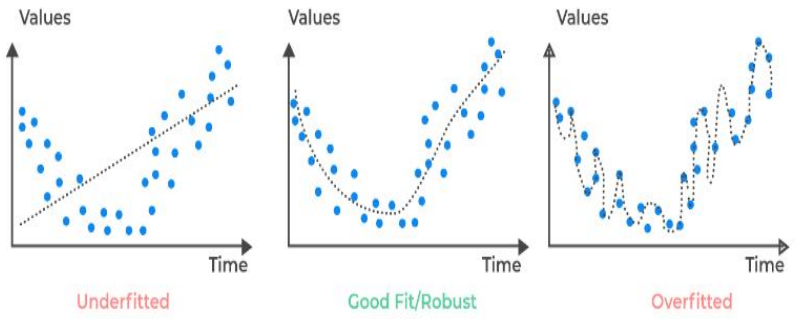


Figure 3.4: Showcasing how underfitting, good fitting and overfitting may look like. [1].

In contrast, the network may be underfitted as well, which can happen if the network is too simple, or is not trained long enough or on enough training data. For example, if the data represents non-linearity, and the network does not have the capability to capture non-linearity. In both cases, the network will perform poorly on new and unseen datasets. To prevent overfitting, techniques such as dropout, the use of weight decay, and early stopping can be used [12].

Dropout is a much-used regularization technique that refers to randomly ”turning off” nodes in the network, which results in a modified network architecture for each training iteration. Without dropout, co-adaption may develop in the network. That is when nodes compensate for wrongs in earlier nodes, which creates intricate and highly data-specified networks. Dropouts combats this, by eliminating these adaptations [49]. Weight decay helps with the generalization by penalizing large weights in the network which distributes the weights more evenly. Weight decay is an added term to the loss function, which helps adjust the gradients before backpropagation [12]. In general, more data might also help combat overfitting. By creating synthetic, or fake data, the algorithm can train on a larger variation and become more robust. [12].

Early stopping is a simple way of ensuring a good final product after training the model. With early stopping, the training is set to stop when a certain prediction accuracy is achieved [12].

3.2.3 Activation Functions

Activation functions play an important part in artificial neural networks as they add non-linearity and simplify complicated mappings between inputs and output, thus improving learning. There exists a wide range of activation functions to choose from. When choosing activation functions, there are several things to consider, like computational efficiency and training efficiency. ReLu, Leaky ReLu, Exponential Linear Unit (ELU), Sigmoid, Tanh, which can be seen in Figure 3.5, are among popular choices [35].

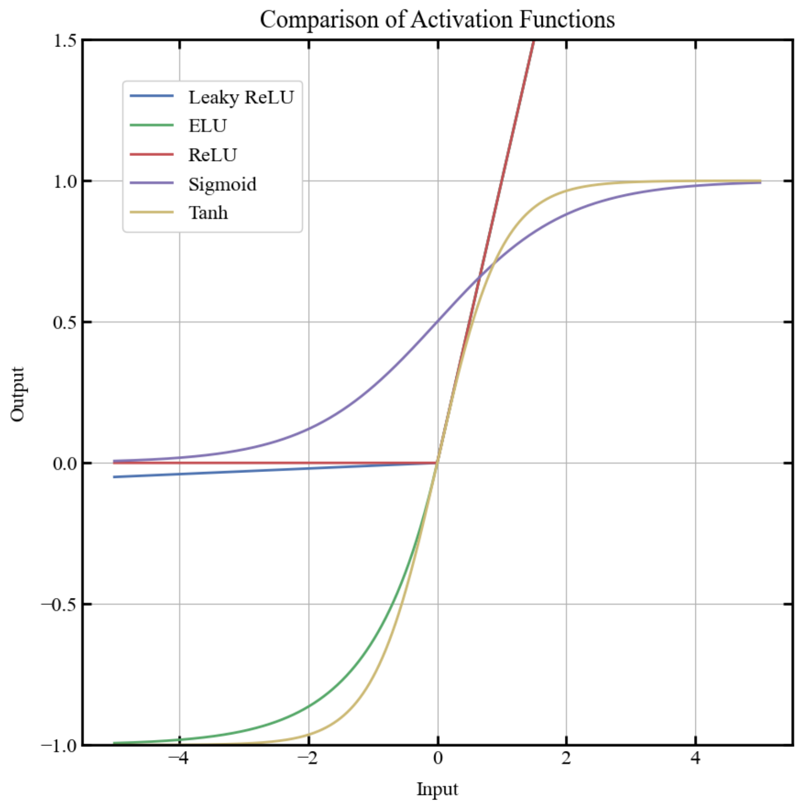


Figure 3.5: Five different types of activation functions.

$$
\begin{aligned}
\text{ReLU}(x) &= \max(0, x) \\
\text{LeakyReLU}(x) &= \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \\
\text{ELU}(x) &= \begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \\
\text{Sigmoid}(x) &= \frac{1}{1 + e^{-x}} \\
\text{Tanh}(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}}
\end{aligned}
$$

Different activation functions are preferred for different types of prediction problems. For classification problems, a combination of Sigmoid functions and/or Tanh functions are proven to give best results. Tanh and Sigmoid are popular choices for recurrent neural networks, while ReLU, or variations of ReLU, are more recommended for Multi Layered Perceptrons or Convolutional Nets ([5] and [35]).

ReLU is the most widely used activation function, because of its effective and steady performance in most cases [34]. It involves simpler mathematical operations than Sigmoid and Tanh. In addition, it offers sparse activation which happens because all negative values output 0, meaning fewer active neurons and a lighter network. The ReLU also avoids the vanishing gradient problem. The vanishing gradient problem occurs when activations reach near the horizontal part of the curve for activation functions like the Tanh or Sigmoid function. This small or non-existing gradient causes the network to become drastically slow or prevents it from further learning. However, while solving the vanishing gradient problem, the ReLU can introduce a different gradient problem ([5] & [35]).

The 'dying ReLU' problem is a phenomenon often associated with the use of the ReLU activation function, leading to its occasional replacement with variants like Leaky ReLU or ELU. This problem arises when a ReLU unit consistently outputs zero for all input values during training, effectively becoming 'dead'. Once a neuron enters this state, it is unlikely to recover because the gradient of ReLU is zero for all negative inputs. Consequently, during the backpropagation process, no gradients pass through the neuron and its weights remain unchanged. This issue hampers the model's ability to learn and adjust to the data, hence limiting its overall performance [18].

The 'dying ReLU' problem is not a consistent occurrence due to the stochastic nature of gradient descent-based optimizers. These optimizers process multiple input values during each iteration, introducing variation in the weight updates. As long as not all inputs consistently push the ReLU into the negative region, resulting in a 'dead' neuron, the weights can still be updated and the learning process can continue [18].

3.2.4 Loss functions

Choosing the right loss function, $\mathscr{L}(y,\hat{y})$, is significantly important since it determines how the network handles the differences between the predictions and the labels. This can be done through different functions, such as mean square error (MSE), mean square percentage (MSPE), L1 loss function (L1), or a custom loss function [27] & [23].

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.3}$$

$$MSPE = \frac{100}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{y_i}\right)^2 \tag{3.4}$$

$$L1 = \sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{3.5}$$

MSE loss, Eq. 3.3, as the name suggests, squares the difference, which makes larger errors penalized more heavily than smaller errors. The simplicity with respect to the mathematical formulation makes it simple to calculate gradients [23] & [12]. MSPE loss, Eq. 3.4, is similar to MSE, however, it gives the loss of the square in a percentage format. It can thereby represent the loss in a more logical way, but at the same time uses more computational time and are sensitive to numbers close to, or equal to, zero. L1 loss, Eq. 3.5, is more stable against outliers since the error is not squared. Lastly, there is not one loss function which is the best, since different applications require different loss functions.

3.3 Graph Neural Networks

Graph Neural Networks (GNNs) were introduced in a paper titled *The Graph Neural Network Model* authored by Franco Scarselli Et al. in 2009. They proposed an adaption of the neural network, that was capable of processing data structured in the form of a graph, a powerful and effective way of representing data. This gave the neural networks the ability to tackle more intricate and complex data structures [33].

Traditionally, data for neural networks could be pictures, represented by grids, or text and time series represented in a sequenced manner. This introduces one of the main advantages of GNNs, the ability to handle irregular data [50]. With the utilization of graph representations, predictability within fields such as social media networks, molecular structures, fake news detection and traffic networks, significantly improves [31].

The order of the input nodes is arbitrary for the GNN, which means that the sequence the nodes are inputted does not affect the output prediction. This can be a valuable feature, creating robust modules that can handle unorganized data. The reason for this invariance is the connectivity between the nodes. The connectivity, which tells what lies between the nodes, gives the network crucial context [50].

Thereby, the GNNs offer similar usage of traditional NNs, however with clear advantages when the data is most appropriately represented as a graph where the connections matter for the predictions.

3.3.1 Utilizing graphs

As mentioned, there are different types of problems where GNNs can be utilized such as Graph-, Node- and Edge-level tasks. Graph-level problems are when the goal is to make a prediction for a whole graph. A physical structure represented as a graph, can be an example, does the structure as a whole exceed a certain safety limit or not? Node-level problems deal with predictions of attributes of the individual nodes within the graph. To continue with the structural example, a node-level task would be if the objective was to predict nodal deformations in the structure rather than an overall safety assessment. Edge-level problems are concerned with edge attribute prediction. To conclude the structural example, an edge-level problem could be to predict the lateral torsional buckling coefficient for the beams represented by edges in between the nodes [31].

3.3.2 Graphs

A graph is built up of vertices or nodes **V**, with edges connecting the different nodes together. The nodes harbour the details regarding each individual node, while the attributes of the edges, **E**, depict the relationship or connections between two adjacent nodes. In addition, information about the entire graph can be stored in a master node **U**. Figure 3.6 shows that graphs can have undirected or directed edges, which determines if the data flow both ways or not [31].

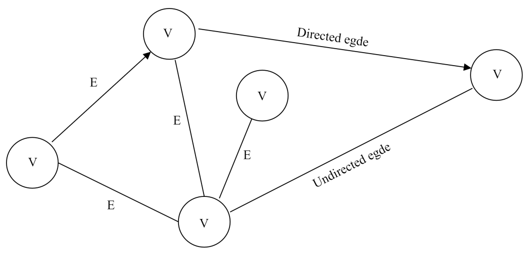
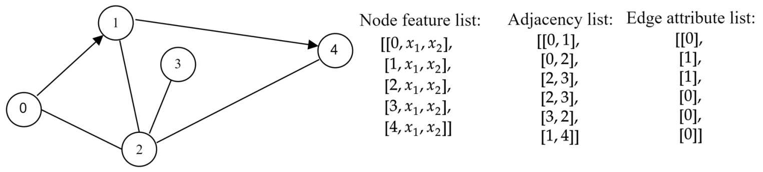


Figure 3.6: A graph and its components, vertices or nodes with edges.

3.3.3 Tensor representation

Structuring of the graphs varies from example to example. Graphs can for example be stored in tensors with a *node feature list*, *edge attribute list*, and an *adjacency list*. Figure 3.7 shows what this may look like.



Node feature list:	Adjacency list:	Edge attribute list:
[[0, x_1, x_2],	[[0, 1],	[[0],
[1, x_1, x_2],	[0, 2],	[1],
[2, x_1, x_2],	[2, 3],	[1],
[3, x_1, x_2],	[2, 3],	[0],
[4, x_1, x_2]]	[3, 2],	[0],
	[1, 4]]	[0]]

Figure 3.7: A graph with corresponding tensor representation

Here, the node feature list contains two attributes x_1 and x_2, which are attributes connected to the specific node. These attributes can be a wide array of things, such as nodal positions or age and gender for a person represented by a node. In general, the node feature tensor will have the shape $[n_{nodes}, node_{dim}]$. The adjacency list describes which nodes are connected and in what direction. In this example, the edge attribute list includes a binary of 0 or 1. However, in general, the edge attribute list can contain an arbitrary amount of data [31]. The utilization of tensor representation creates a structured approach to constructing the large datasets needed for training the neural network and is the central data abstraction used for popular machine learning frameworks such as PyTorch and TensorFlow [37] & [39].

3.3.4 Building graph neural networks

The connectivity in a GNN is expressed through the message passing operations, where the nodes and edges learn layer-by-layer their own context in the graph structure. Figure 3.8 displays node to node message passing from the node N_i. The message passing works in a sequential manner. Initially, each node gathers information from its connected nodes. Subsequently, the collected messages are aggregated and transformed through a pooling operation. This pooled message is then passed to the connected nodes in the subsequent layer after undergoing an update function. The nodes **N** will do this simultaneously and it can occur edge to edge, edge to node, and node to edge as well [31].

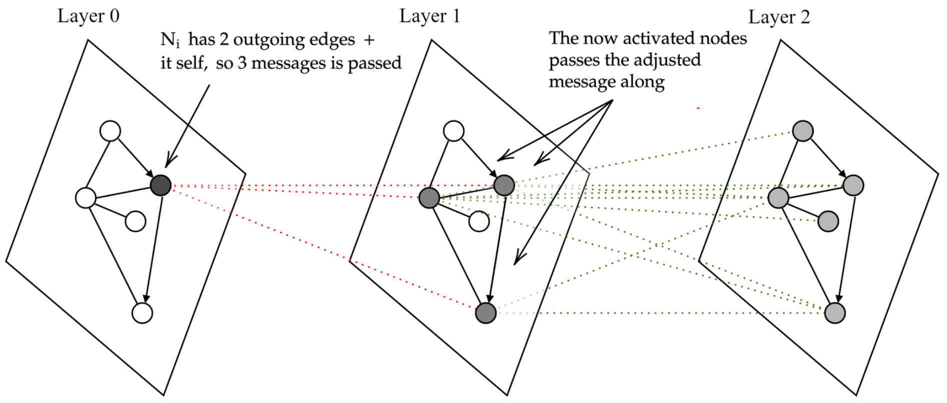

Layer 0
N_i has 2 outgoing edges + it self, so 3 messages is passed
Layer 1
The now activated nodes passes the adjusted message along
Layer 2

Figure 3.8: How message passing, node to node, works.

A complete graph neural network is made up of multiple message passing layers and classification layers. Figure 3.9 displays a possible end-to-end configuration of a graph neural network. This example includes message passing between all elements in the graph, but as mentioned, this is not necessary for all problems and should be determined case by case.

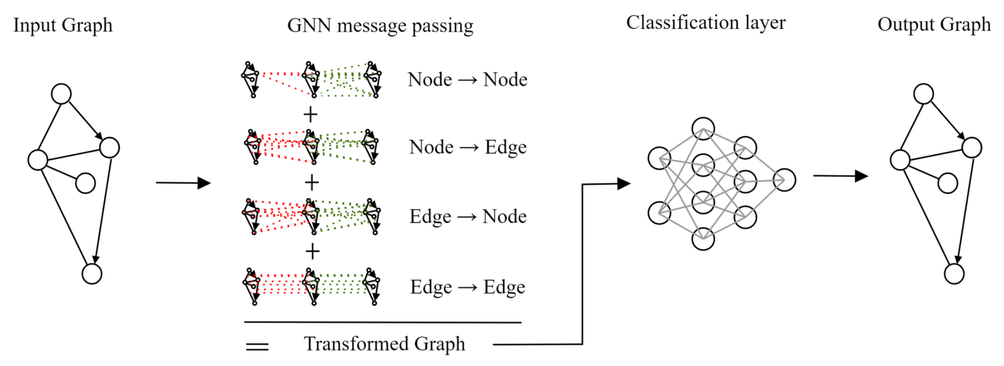


Figure 3.9: Possible structure for a Graph neural network.

The initial graph is the input, and graph aggregation takes place through the layers, connecting different parts of the graph. Once all the messages have been propagated through the layers, the information flows into the subsequent stage of the program. This stage typically consists of a specialized layer responsible for generating predictions and outputting this in the desired output format [31].

3.3.5 GNNs for structural engineering

The utilization of graph neural networks in structural engineering offers distinct advantages, primarily due to their ability to precisely represent geometry as a graph. This approach proves particularly beneficial for beam and truss structures, where geometry can be accurately encoded using node features for coordinates and edge features for element information. This encoding also enables handling arbitrary geometries, making graph neural networks a powerful tool in structural engineering [47].

FeaStNet, *Feature-Steered Graph Convolutions*, is an example of a graph neural network layer within the PyTorch framework that can be used in a structural engineering context. While the name implies it, FeaStNet is not a standard convolutional network. Instead, it is a graph convolutional network specifically designed for handling graph-structured data. It allows for an arbitrary graph topology and is invariant to feature space. This means that direct spatial coordinates can be used as input for the network, which is often essential in practical cases since it allows for an arbitrary geometry [45] & [47].

In contrast, standard Convolution Neural Networks (CNNs), do not easily extend to data that is not represented in a grid format [45]. They are primarily used for image-driven pattern recognition tasks, by taking advantage of the image input assumption to encode the network more effectively by reducing the number of parameters. CNNs consist of convolutional, pooling, and fully connected layers. The convolutional layers are responsible for capturing patterns in the image, such as edges, textures, or shapes, by applying different sets of filters in a sliding manner. As the filters are applied to the image, more complex visual representations can be learned. The pooling layer is crucial in downsampling and reducing spatial dimensions. Finally, the fully connected layer performs classification and makes the final predictions [36].

4 VelociraptorFEA: Static and Dynamic Beam Calculation

This chapter focuses on showcasing two developed plug-ins: Velociraptor2D (V2D) and Velociraptor3D (V3D). These plug-ins serve as finite element analysis (FEA) framework, with V2D operating in the 2D space and V3D operating in the 3D space. It is worth noting that V3D is an expansion of V2D, resulting in a similarity in appearance and usage between the two. In the following sections, both plug-ins will be described in detail, highlighting their features and functionalities. They have been developed in Visual Studio, and written object orientated in the programming language C#. Note that there will be some necessary repetition in the following description, due to the similarity of the two plug-ins.

4.1 Velociraptor2D Method

The initial plug-in developed is called Velociraptor2D, which is a finite element analysis framework designed for analyzing 2D structures. V2D accurately computes the static and dynamic responses to applied loads. This section will showcase and provide an explanation of all the components within the V2D environment. Figure 4.1 illustrates the general workflow of the software in Grasshopper. The code written for V2D can be found in the Git Repository: Velociraptor2D.

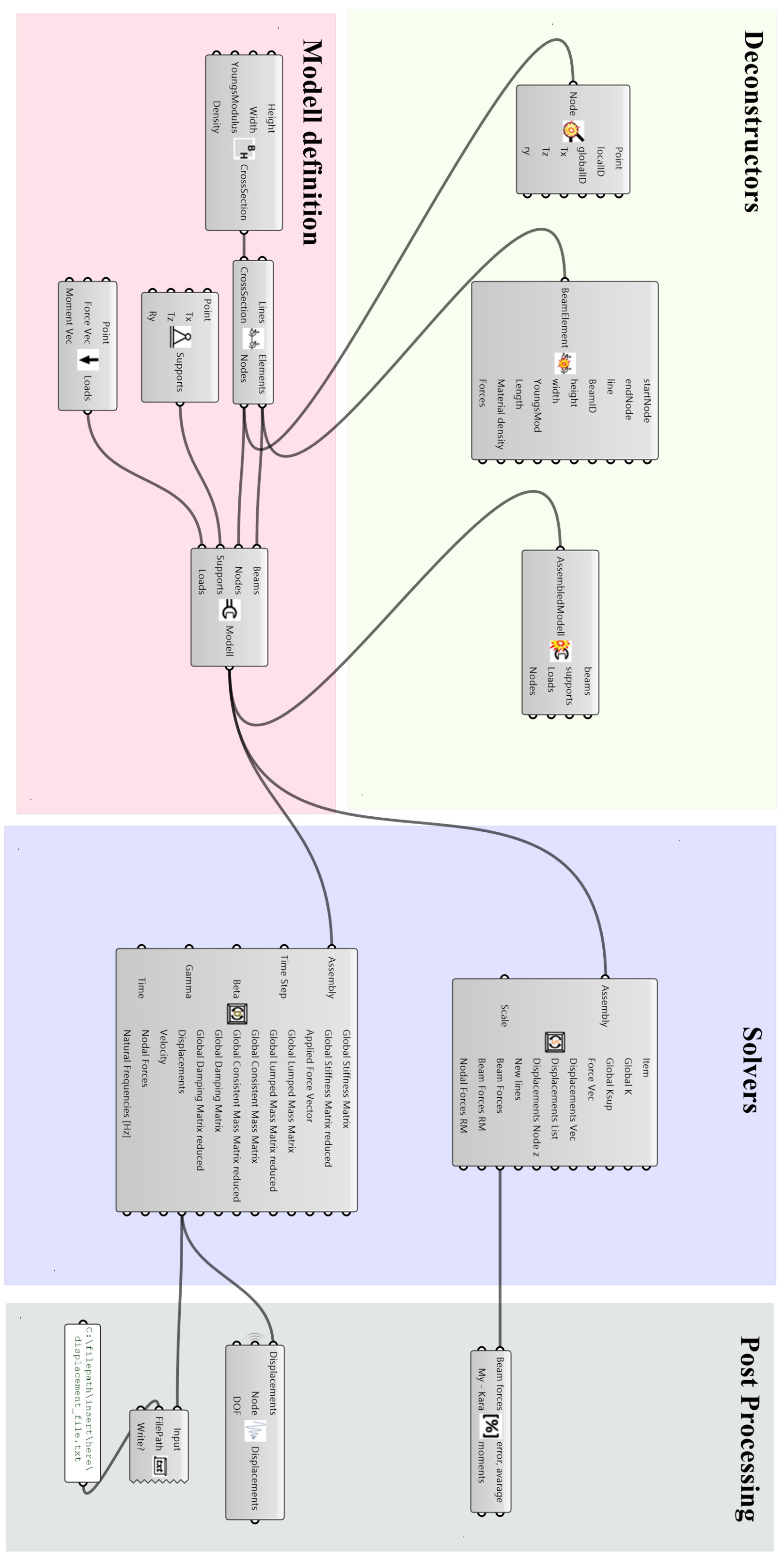


Figure 4.1: General workflow for The Velociraptor2D.

4.1.1 Components

AddCrossSection, Table 4.1, creates *CrossSection* objects by receiving the height and width of the cross section and the Youngs Modulus and density of the material.

Table 4.1: Input and output for component AddCrossSection

AddCrossSection			
Input	Type	Output	Type
Height	Integer	CrossSection	CrossSection Obj.
Width	Integer		
YoungsModulus	Integer		
Density	Integer		

CreateLoad, Table 4.2, creates *Load* objects by receiving the nodal position of the point force and a vector. This can be a force vector or a moment vector, both represented as 3D-Vectors. For the *ForceVector* only the value of the x- and y-component will be used and for the *MomentVector* only the y-value. Thereby, if there is a component in the vector outside of the 2D environment, it will be disregarded.

Table 4.2: Input and output for component CreateLoad

CreateLoad			
Input	Type	Output	Type
Point	Point3D	Loads	List<Load Obj.>
ForceVector	Vector3D		
MomentVector	Vector3D		

CreateSupport, Table 4.3, generates *Support* objects by taking the support point and *boolean* inputs indicating the freedom or locking status for each degree of freedom (DOF) as inputs.

Table 4.3: Input and output for component CreateSupport

CreateSupport			
Input	Type	Output	Type
Point	Point3D	Support	List<Support Obj.>
Tx	Boolean		
Tz	Boolean		
Ry	Boolean		

CreateBeamElements, Table 4.4, generates *BeamElement* objects by using a list of Grasshopper lines and their corresponding *CrossSection* objects as inputs. Additionally, it describes the creation of *Node* objects by utilizing the endpoints of each beam. If a node already exists at either one of the beam endpoint locations, the algorithm identifies the corresponding node and adds it to the *BeamElement* object. Furthermore, the length of each beam is calculated and stored within the object for reference and further analysis.

Table 4.4: Input and output for component CreateBeamElements

CreateBeamElements			
Input	Type	Output	Type
Lines	List<Lines>	Support	List<BeamElement Obj.>
CrossSection	CrossSection Obj.	Nodes	List<Node Obj.>

AssembleModel, Table 4.5, gathers the relevant objects; *BeamElement* objects, *Support* objects, *Load* objects, and *Node* objects in one *Assembly* object.

Table 4.5: Input and output for component AssembleModel

AssembleModel			
Input	Type	Output	Type
Beams	List<BeamElement Obj.>	Modell	Assembly Obj.
Nodes	List<Node Obj.>		
Supports	List<Support Obj.>		
Loads	List<Load Obj.>		

SolverStatic, Table 4.6, in short, takes in the *Assembly* object, deconstructs it, and performs the static finite element analysis. The process starts by building the global stiffness matrix using the elements' stiffness transformed into the global axis system, as described in Section 2.3. Supports are then considered by modifying the global stiffness matrix. The force vector is created based on applied loads and boundary conditions. Nodal displacements are then computed by solving the system of equations represented by the stiffness matrix and force vector.

By having the nodal displacement, the beam forces can be calculated, in an iterative process. The nodal displacements corresponding to the beam nodes are transformed into the beam coordinate system and multiplied with the stiffness, which gives the beam forces.

Lastly, the new geometry can be drawn with an input that determines the scale of the drawn displacement in Rhino.

Table 4.6: Input and output for component SolverStatic

SolverStatic			
Input	Type	Output	Type
Assembly	Assembly Obj.	Modell	Assembly Obj.
Scale	Integer	Item	Generic parameter
		Global K	Rhino Matrix
		Force Vec	LA.Matrix<Double>
		Displacement Vec	LA.Matrix<Double>
		Displacement List	List<string>
		New Lines	List<Line>
		Beam Forces	LA.Matrix<Double>
		Beam Forces RM	Rhino Matrix
		Nodal Forces RM	Rhino Matrix

DynamicSolver, Table 4.7, functions similarly to Table 4.6, taking the Assembly object as input. It constructs the global stiffness matrix and force vector, as well as the global mass matrix by incorporating the mass contributions of each element after transforming them into the global axis system. The damping matrix is also defined based on the process outlined in Section 2.4.2, as standard w_1 and w_2 is set to 0.1 and 100, respectively. This is to include a wide array of eigenvalues in the analysis.

Using Newmark's method, following the Procedure 2.2 the time history displacements are calculated. Additionally, the eigenvalue problem is solved to determine the eigenfrequencies after Section 2.4.3.

Table 4.7: Input and output for component DynamicSolver

DynamicSolver			
Input	Type	Output	Type
Assembly	Assembly Obj.	Global stiffness Matrix	Rhino Matrix
Scale	Integer	Global stiffness Matrix reduced	Rhino Matrix
		Applied Force Vector	Rhino Matrix
		Global Lumped Mass Matrix	Rhino Matrix
		Global Lumped Mass Matrix reduced	Rhino Matrix
		Global consistent Mass Matrix	Rhino Matrix
		Global consistent Mass Matrix reduced	Rhino Matrix
		Global Damping Matrix	Rhino Matrix
		Global Damping Matrix reduced	Rhino Matrix
		Displacements	LA.Matrix<Double>
		Velocity	LA.Matrix<Double>
		Nodal Forces	LA.Matrix<Double>
		Natural Frequencies	Rhino Matrix

ToTextFile, Table 4.8, is the component that can write data to a .txt file.

Table 4.8: Input and output for component ToTextFile

ToTextFile			
Input	Type	Output	Type
Input	List<double>		
FilePath	String		
Write?	Boolean		

ForceCheck, Table 4.9, takes two inputs, the beam forces from V2D and Karamaba3D and calculates the differences in forces, which it returns as an average number for each node in the structure. The moment in each node is also outputted.

Table 4.9: Input and output for component ForceCheck

ForceCheck			
Input	Type	Output	Type
Beam forces	LA.Matrix<Double>	Error, Avg	Double
My - Karamba	List<Double>	ErrorList Beam	List<Double>
		Moments Beam	List<Double>

TimeHistory, Table 4.10, takes all the time history displacements and outputs for the wanted DOF and node in the structure. This allows for easy access to the needed displacements, and to use the GH-component *Quick Graph* to plot directly in Grasshopper.

Table 4.10: Input and output for component TimeHistory

TimeHistory			
Input	Type	Output	Type
Displacements	LA.Matrix<Double>	Displacements	List<Double>
Node	Node Object		
DOF	Integer		

Table 4.11 shows the different deconstructors made. The three different components are useful for validation and create a better understanding of what the plug-in does since it enables the user to peak inside the data that is used for calculations.

Table 4.11: Deconstructors within the V2D environment

Deconstructors		
Input	Type	Output
BeamElement	List<BeamElement Obj.>, or single BeamElement Obj.	Beam Attributes
Node	List<Node Obj.>, or single Node Obj.	Node Attributes
AssembleModel	Assembly Obj.	Assembly Attributes

4.2 Velociraptor3D Method

The second plug-in developed is called Velociraptor3D (V3D), which builds upon the foundation of Velociraptor2D and extends its capabilities to three-dimensional problems. Similar to V2D, V3D encompasses both static and dynamic solvers. This section will present a detailed showcase and explanation of the V3D environment. The general workflow of the plug-in is depicted in Figure 4.2.

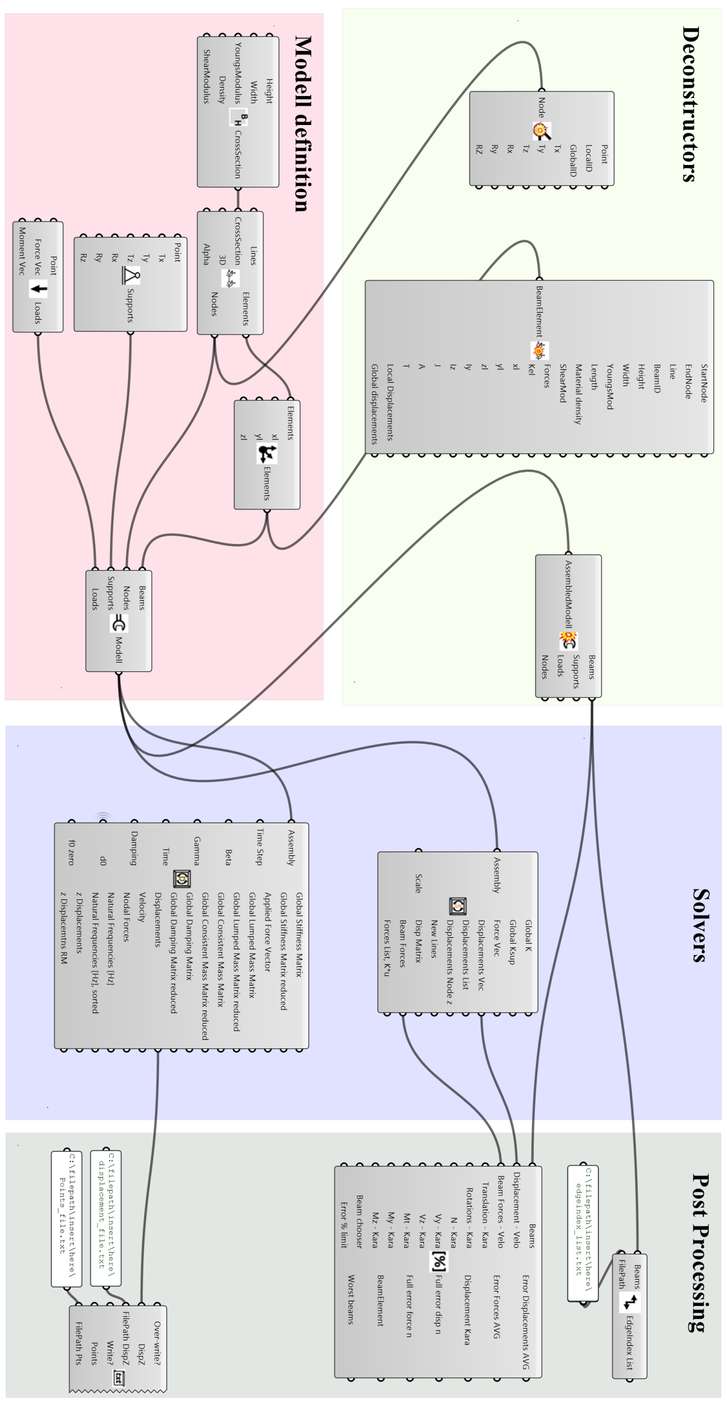


Figure 4.2: General workflow for The Velociraptor3D

4.2.1 Components

AddCrossSection3D, Table 4.12, creates *CrossSection* objects by receiving the height and width of the cross section and the Young's modulus, shear modulus and density of the material.

Table 4.12: Input and output for component AddCrossSection3D

AddCrossSection3D			
Input	Type	Output	Type
Height	Double	CrossSection	CrossSection Obj.
Width	Double		
YoungsModulus	Integer		
Density	Double		
Shear modulus	Double		

CreateLoad, Table 4.13, creates *Load* objects by receiving point where the load is applied and the vector. This can be a force vector or a moment vector, both represented as 3D Vectors.

Table 4.13: Input and output for component CreateLoad3D

CreateLoad			
Input	Type	Output	Type
Point	Point3D	Loads	List<Load Obj.>
ForceVector	Vector3D		
MomentVector	Vector3D		

CreateSupport3D, Table 4.14, generates *Support* objects by taking the support points and *booleans* indicating the freedom or locking status for each DOF.

Table 4.14: Input and output for component CreateSupport3D

CreateSupport3D			
Input	Type	Output	Type
Point	Point3D	Support	List<Support Obj.>
Tx	Boolean		
Ty	Boolean		
Tz	Boolean		
Rx	Boolean		
Ry	Boolean		
Rz	Boolean		

CreateBeamElements3D, Table 4.15, generates *BeamElement* objects by having a list of Grasshopper lines and the corresponding *CrossSection* object as input. It also generates *Node* objects.

Table 4.15: Input and output for component CreateBeamElements3D

CreateBeamElements3D			
Input	Type	Output	Type
Lines	List<Line>	Support	List<Support Obj.>
CrossSection	CrossSection Obj.	Nodes	List<Node Obj.>
3D	Boolean		
Alpha	Double		

SetLocalAxis3D, Table 4.16, gives the user the possibility to define the local axis system for the beam elements. This is important in the 3D space since without it would be hard to be absolutely sure of orientations, and thus the validity of the analysis. When comparing with Karamba3D for example, the local axis system of the Karamba3D elements can be used for the V3D elements, ensuring the same orientations of elements.

Table 4.16: Input and output for component SetLocalAxis3D

SetLocalAxis3D			
Input	Type	Output	Type
Elements	List<BeamElement Obj.>	Elements	List<BeamElement Obj.>
xl	Vector3D		
yl	Vector3D		
zl	Vector3D		

AssembleModel3D, Table 4.17, gathers the relevant objects; *BeamElement* objects, *Support* objects, *Load* objects, and *Node* objects in one *Assembly* object.

Table 4.17: Input and output for component AssembleModel3D

AssembleModel3D			
Input	Type	Output	Type
Beams	List<BeamElement Obj.>	Modell	Assembly Obj.
Nodes	List<Nodes Obj.>		
Supports	List<Supports Obj.>		
Loads	List<Loads Obj.>		

SolverStatic3D, Table 4.18, takes in the *Assembly* object, deconstructs it, and performs static FEA calculations. There is also a scale input that determines the scale of the drawn displacement in Rhino.

Table 4.18: Input and output for component SolverStatic3D

SolverStatic3D			
Input	Type	Output	Type
Assembly	Assembly Obj.	Model	Assembly Obj.
Scale	integer	Global K	Rhino Matrix
		Global Ksup	Rhino Matrix
		Force Vec	LA.Matrix<Double>
		Displacement Vec	LA.Matrix<Double>
		Displacement List	List<string>
		Displacement Node z	List<Double>
		New Lines	List<Line>
		Disp Matrix	Rhino Matrix
		Beam Forces	LA.Matrix<Double>
		Forces List, K*u	List<string>

DynamicSolver3D, Table 4.19, takes the *Assembly* object as an input and performs a dynamic analysis using Newmark's method. The initial displacements obtained from the static solver are input as the value for ”d0”. Additionally, the *boolean* toggle ”f0 zero” determines whether the initial force should be set to zero or if the static force should be applied as the initial force.

Table 4.19: Input and output for component DynamicSolver3D

DynamicSolver3D			
Input	Type	Output	Type
Assembly	Assembly Obj.	Global stiffness Matrix	Rhino Matrix
Time Step	Integer	Global stiffness Matrix reduced	Rhino Matrix
Beta	Double	Applied Force Vector	LA.Matrix<Double>
Gamma	Double	Global Lumped Mass Matrix	Rhino Matrix
Time	Double	Global Lumped Mass Matrix reduced	Rhino Matrix
Damping	Double	Global consistent Mass Matrix	Rhino Matrix
d0	LA.Matrix<Double>	Global consistent Mass Matrix reduced	Rhino Matrix
f0 zero	Boolean	Global Damping Matrix	Rhino Matrix
		Global Damping Matrix reduced	Rhino Matrix
		Displacements	LA.Matrix<Double>
		Velocity	LA.Matrix<Double>
		Natural Frequencies [Hz]	Rhino Matrix
		Natural Frequencies [Hz], sorted	Rhino Matrix
		z Displacements	LA.Matrix<Double>
		z Displacements RM	Rhino Matrix

ToTextFile3D, Table 4.20, takes two inputs, displacement values, and the points and then writes the displacements and nodal coordinates to the file. There is a boolean toggle that decides if the component should append the data or overwrite the existing file.

Table 4.20: Input and output for component ToTextFile3D

ToTextFile3D			
Input	Type	Output	Type
Over-	Boolean		
Write?			
DispZ	LA.Matrix<Double>		
FilePath	String		
DispZ			
Write?	Boolean		
Points	Boolean		
FilePath	String		
Pts			

Checky, Table 4.21, takes in calculated data from V3D and from Karamba3D and calculates the differences in displacements and forces.

Table 4.21: Input and output for component Checky

Checky			
Input	Type	Output	Type
Beams	List<BeamElement obj.>	Error Displacement AVG	Double
Disp - Velo	LA.Matrix<Double>	Error Forces AVG	Double
B.F - Velo	LA.Matrix<Double>	Displacement kara	List<Double>
Trans. - Kara	List<String>	Full error disp n	List<String>
Rot. - Kara	List<String>	Full error force n	List<String>
N-Kara	List<Double>	BeamElement	BeamElement obj.
Vy-Kara	List<Double>	Worst Beams	List<BeamElem. obj.>
Vz-Kara	List<Double>		
Mt-Kara	List<Double>		
My-Kara	List<Double>		
Mz-Kara	List<Double>		
Beam Chooser	List<Double>		
Error% limit	List<Double>		

EdgeIndex, Table 4.22, allows for the creation of the necessary edge indexes for the future machine learning part. It writes the data directly to the file and outputs the list as well.

Table 4.22: Input and output for component EdgeIndex

EdgeIndex			
Input	Type	Output	Type
Beams	List<BeamElement obj.>	EdgeIndex	List<String>
		List	
FilePath	String		

Table 4.23 shows the different deconstructors made, which output multiple different attributes that may be of interest to the user.

Table 4.23: Deconstructors within the V3D environment

Deconstructors		
Input	Type	Output
BeamElement	List<BeamElement Obj.>, or single BeamElements Obj.	Beam Attributes
Node	List<Node Obj.>, or single Node Obj.	Node Attributes
AssembleModel	Assembly Obj.	Assembly Attributes

4.3 V3D and V2D Benchmark Tests

Within this section, multiple benchmark tests will be conducted to evaluate the capabilities of both V2D and V3D in handling diverse scenarios encompassing both static and dynamic analyses. These tests aim to assess the performance and reliability of the plug-ins across various situations.

4.3.1 Benchmark #1: Discretization of the Arch V2D

The first benchmark test is for the arch within the Velociraptor2D framework. It undergoes discretization with a finer and finer mesh. The arch has a rectangular 10 by 10 mm solid steel section and the fixed supports are placed one meter from each other. There is a point load acting in the middle node, straight down with a force of 100N. The arch and the discretization process can be seen in Figure 4.3, where the arch is made from 2 and 10 beam elements, respectively. The deformation has a scale of 30.

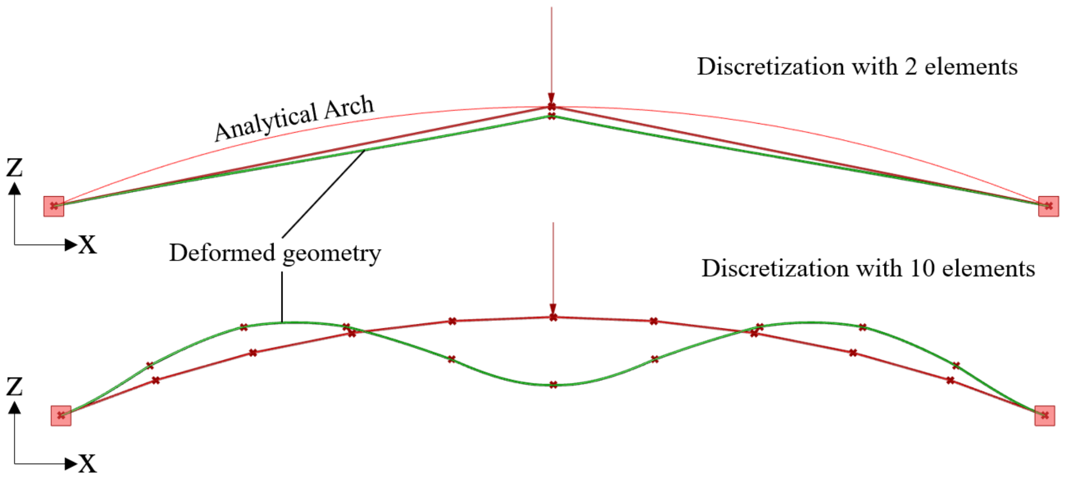

Discretization with 2 elements
Analytical Arch
Z
X
Deformed geometry
Discretization with 10 elements
Z
X

Figure 4.3: Discretization process for the arch.

Figure 4.4 shows how the displacement is converging as the mesh becomes finer. As expected there is a big step from the first step, this is due to a larger part of the load going from being carried by compression to bending.

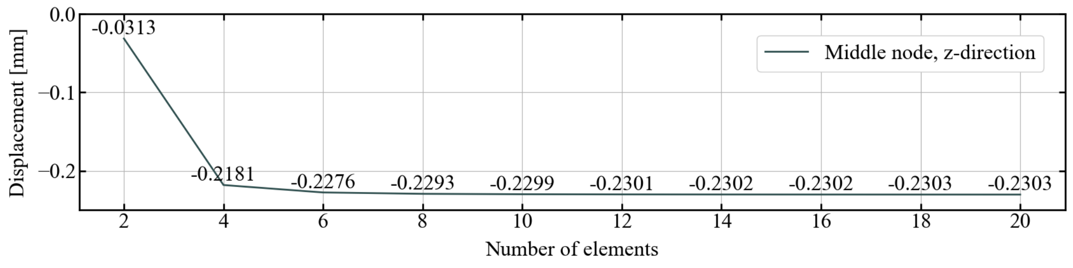


Figure 4.4: Discretization process for the arch with Velociraptor2D.

To validate the final displacement of -0.2303 mm, the arc is calculated in the program Robot Structural Analysis Professional 2023. As Figure 4.5 shows that the final displacement is equal to -0.2302 mm, which gives a difference of 0.043%.

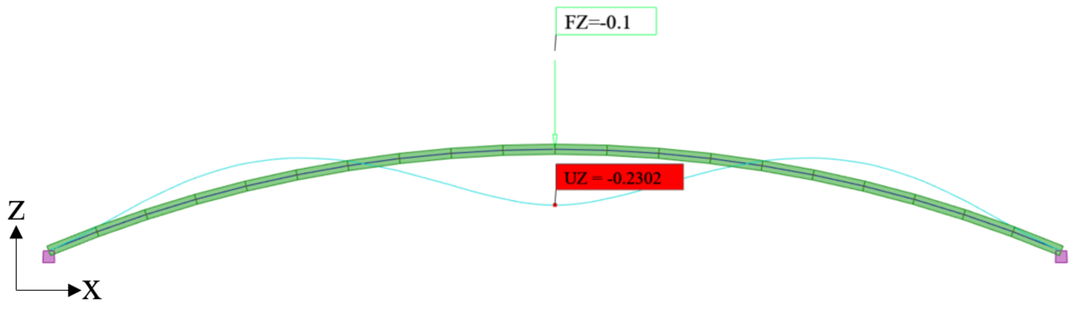


Figure 4.5: Validation of results of FEM solver in Robot.

4.3.2 Benchmark #2: Discretization of the Arch V3D

The second benchmark test is similar to the first, Figure 4.3, only within the Velociraptor3D framework and with the arch rotated 1 radian. This should provide similar results as Benchmark #1.

Figure 4.6 shows how the displacement is converging as the mesh goes from 2 to 20 elements. The displacements of the middle node converge to -0.2303 mm, which is equal to the V2D result. This then shows that the solver is correct for smaller structures and that the transformation matrix is well-defined. The displacements stay the same for an arbitrary rotation of the structure.

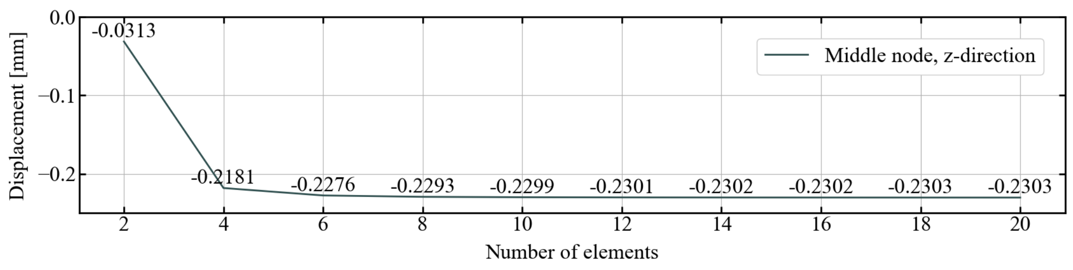


Figure 4.6: Discretization process for the arch with Velociraptor3D.

4.3.3 Benchmark #3: Truss Bridge V3D

Continuing with the evaluation, the Velociraptor3D plugin is tested with larger and more complex structures, and compared to Karamba 3D. As a first step, a 20-meter-long truss bridge is tested. The bridge, shown in Figure 4.7, has 68 elements and 32 nodes. It is fixed in all DOF in the four corner nodes, and subjected to a -1,000 kN point load in each of the two center top nodes. Deformed geometry is shown in green with a scale of 400, while original geometry is shown in red. The cross sections of the bridge elements are solid circular steel beams with a diameter of 200 mm.

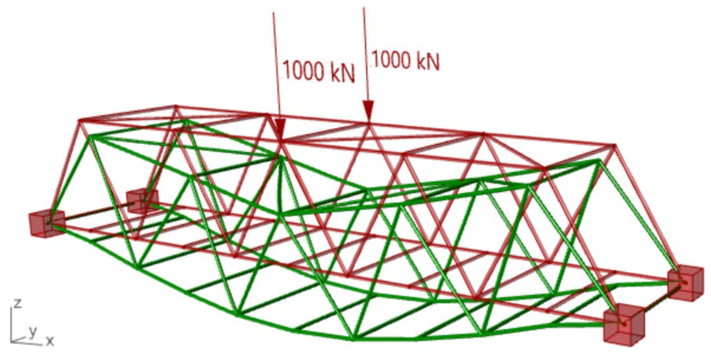


Figure 4.7: 3D truss bridge with deformed geometry.

The average error is calculated for displacements and forces by Equation 4.1. When it comes to deciding the sign of forces, Karamaba3D and Velociraptor3D differ in their approaches, and therefore absolute value is used to ensure correct error calculations.

$$\mathbf{Error} = \frac{|X|^{V3D} - |X|^{K3D}}{|X|^{V3D}} \cdot 100, \quad \mathbf{Error}^{average\%} = \frac{\sum_{i=0}^{n} Error^{\%}}{n} = 0.014\% \tag{4.1}$$

Here, n denotes the number of nodes. In Table 4.24 Velociraptor3D is compared to Karamba3D, presenting differences in displacements and forces for all DOF. The displacements in Karamba3D are given to three significant figures, so Velociraptor3D is rounded to 3 significant figures. The displacement errors are calculated using displacements with three decimals.

Nodal displacement Velociraptor3D vs. Karamba3D	
Direction	Difference
Translation X	0.00 %
Translation Y	0.00 %
Translation Z	0.00 %
Rotation X	0.00 %
Rotation Y	0.00 %
Rotation Z	0.00 %
Nodal forces Velociraptor3D vs. Karamba3D	
Force	Difference
Normal force N	0.105 %
Shear force Vy	0.251 %
Shear force Vz	0.245 %
Moment Mx	0.584 %
Moment My	0.622 %
Moment Mz	3.656 %

Table 4.24: Results Truss Bridge 3d.

Now, one of the truss beams in the bridge, beam 44 shown in green in Fig. 4.8, is evaluated. The beam forces are compared between Velociraptor3D and Karamba3D, and presented in Table 4.25. Here it is shown that the two plugins vary in their process of determining the sign of forces, and we can see a slight difference in value, which is smallest for the normal force and larger for shear forces and moments. This could be due to rounding of numbers and Karamba3D using Timoschenko beam theory for shear deformation, while Velociraptor uses Euler-Bernoulli.

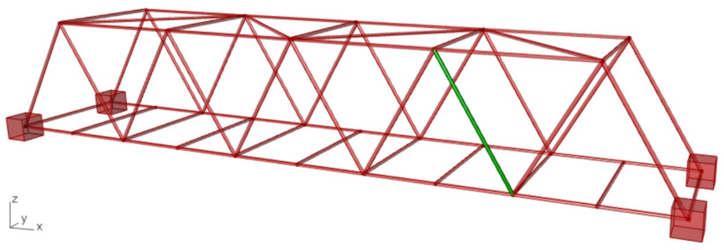


Figure 4.8: Beam 44 shown in green.

Forces in node 1 (Top node)			
Force	Velociraptor3D	Karamba3D	Difference
Normal force N	557156.9 N	-557161.4 N	0.001 %
Shear force Vy	251.7 N	-250.6 N	0.426 %
Shear force Vz	1427.0 N	-1418.4 N	0.602 %
Moment Mx	-129601.2 N	128959.4 N	0.495 %
Moment My	-2897589.5 N	2879051.2 N	0.640 %
Moment Mz	590255.6 N	-587698.0 N	0.433 %
Forces in node 2 (End node)			
Force	Velociraptor3D	Karamba3D	Difference
Normal force N	-557156.9 N	-557161.4 N	0.001 %
Shear force Vy	-251.7 N	-250.6 N	0.426 %
Shear force Vz	-1427.0 N	-1418.4 N	0.602 %
Moment Mx	129601.2 N	128959.4 N	0.495 %
Moment My	-3483952.8 N	-3464072.4 N	0.571 %
Moment Mz	535322.5 N	533083.5 N	0.418 %

Table 4.25: Comparing beam forces for beam 44.

The orientations determining the sign of the forces for Velociraptor3D are defined in Figure 2.3. In Figure 4.9 different truss bridge geometries are shown, to display more versatility of the V3D. All the structures are accompanied by their deformed shape. Table 4.26 shows information about the different bridges. These results will be discussed in 4.4.

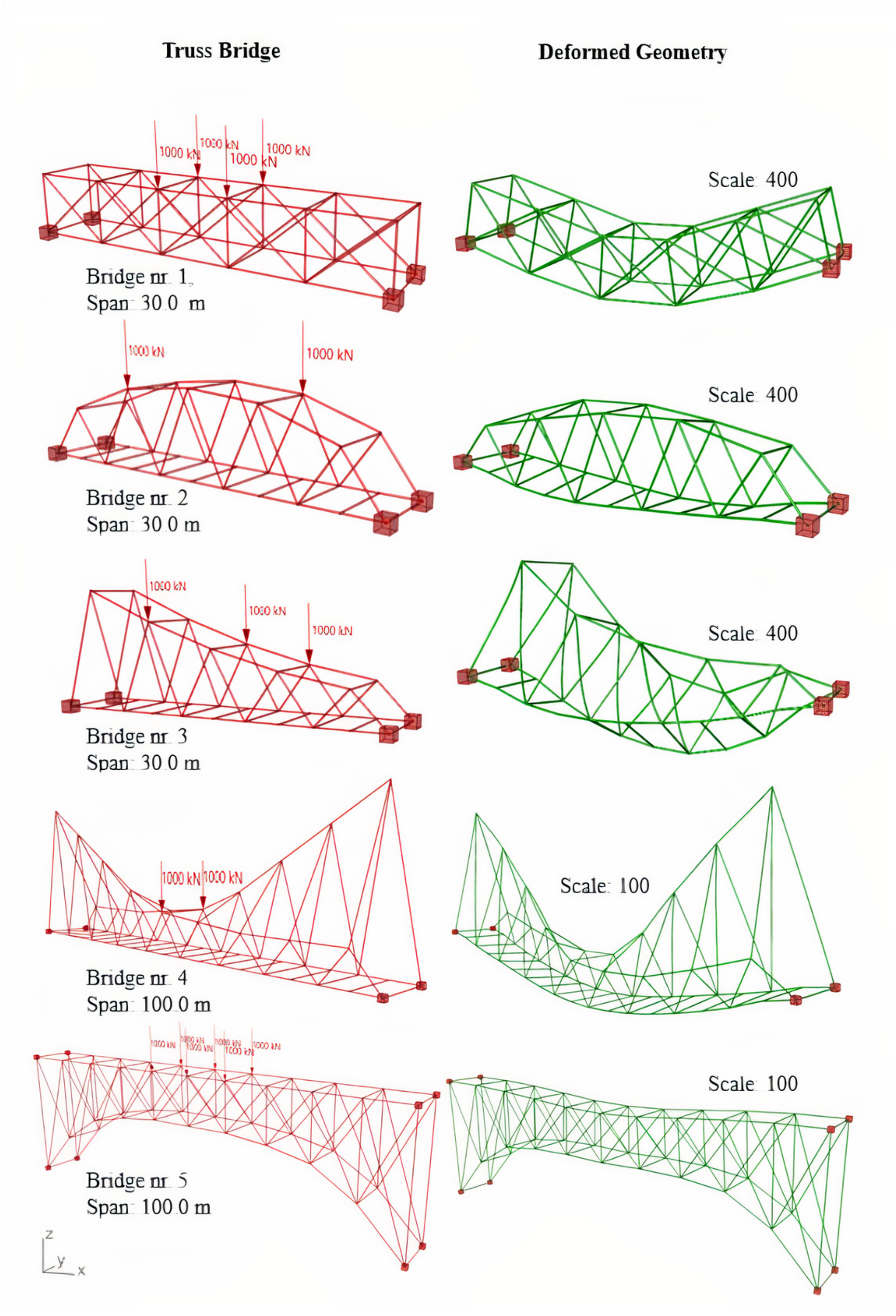


Figure 4.9: Different bridges and their deformed geometry.

Table 4.26: Dimensions and maximum displacements for the bridges shown in Fig. 4.9.

Bridge nr.	Width	Max. height	Min. height	Max. displacement
1	5.0 m	5.0 m	5.0 m	9.27 mm
2	5.0 m	6.5 m	4.0 m	3.93 mm
3	5.0 m	10.0 m	3.0 m	15.09 mm
4	10.0 m	30.0 m	6.0 m	90.37 mm
5	10.0 m	30.0 m	11.0 m	28.55 mm

4.3.4 Benchmark #4: Gridshell V3D

A gridshell, see Figure 4.10, is constructed to further test Velociraptor3D's ability to tackle complex geometry. The structure has a span of 25 meters and a height of 10 meters, consisting of 199 elements and 80 nodes. The cross-section is a solid steel pipe with a diameter of 100 mm. It is fixed in all DOFs in the bottom nodes. As the figure shows the structure is subjected to a nonuniform nodal load of 100 kN in two of the arches.

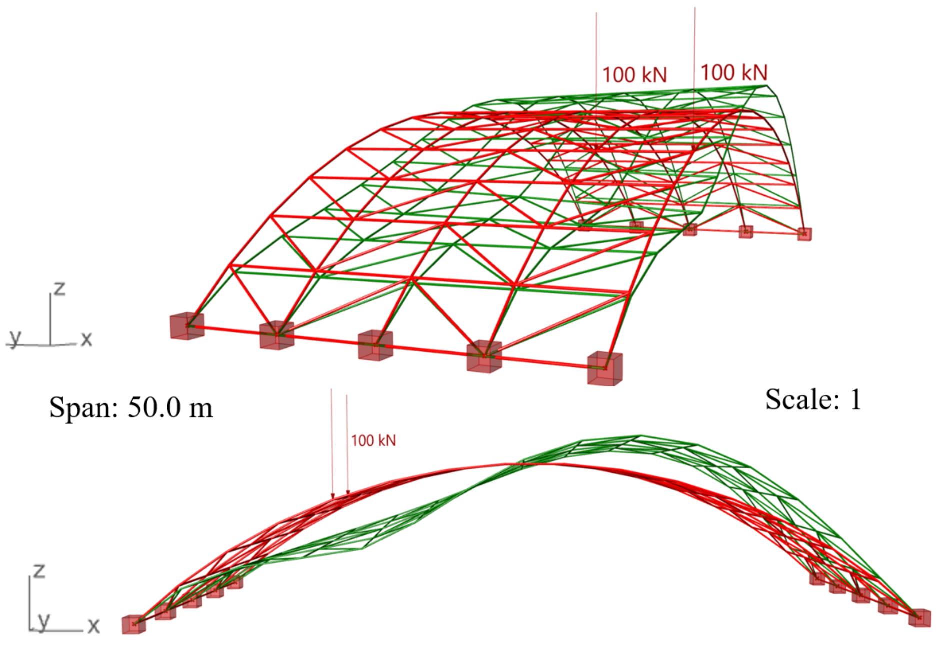


Figure 4.10: One 3D Gridshell structure, deformed scaled 1 in green colour.

The error between Karamba3D and Velociraptor3D is again calculated after Equation 4.1 and shown below in Table 4.27. The error is very small and is likely due to numerical errors combined with the difference in implemented beam theories.

Table 4.27: Results gridshell 3d

Nodal displacement average error	
Direction	Error
Translation X	0.010 %
Translation Y	0.041%
Translation Z	0.009%
Rotation X	0.000%
Rotation Y	0.013%
Rotation Z	0.000%
Nodal forces average error	
Force	Error
Normal force N	0.016%
Shear force Vy	0.109%
Shear force Vz	0.181%
Moment Mx	0.035%
Moment My	0.126%
Moment Mz	0.168%

In Figure 4.11 four different gridshells geometries are shown accompanied by their deformed shape, to showcase the flexibility in design that V3D offers. All structures are made with the same cross section, the solid steel pipe with a diameter of 100 mm. Table 4.28 shows information about the different geometries.

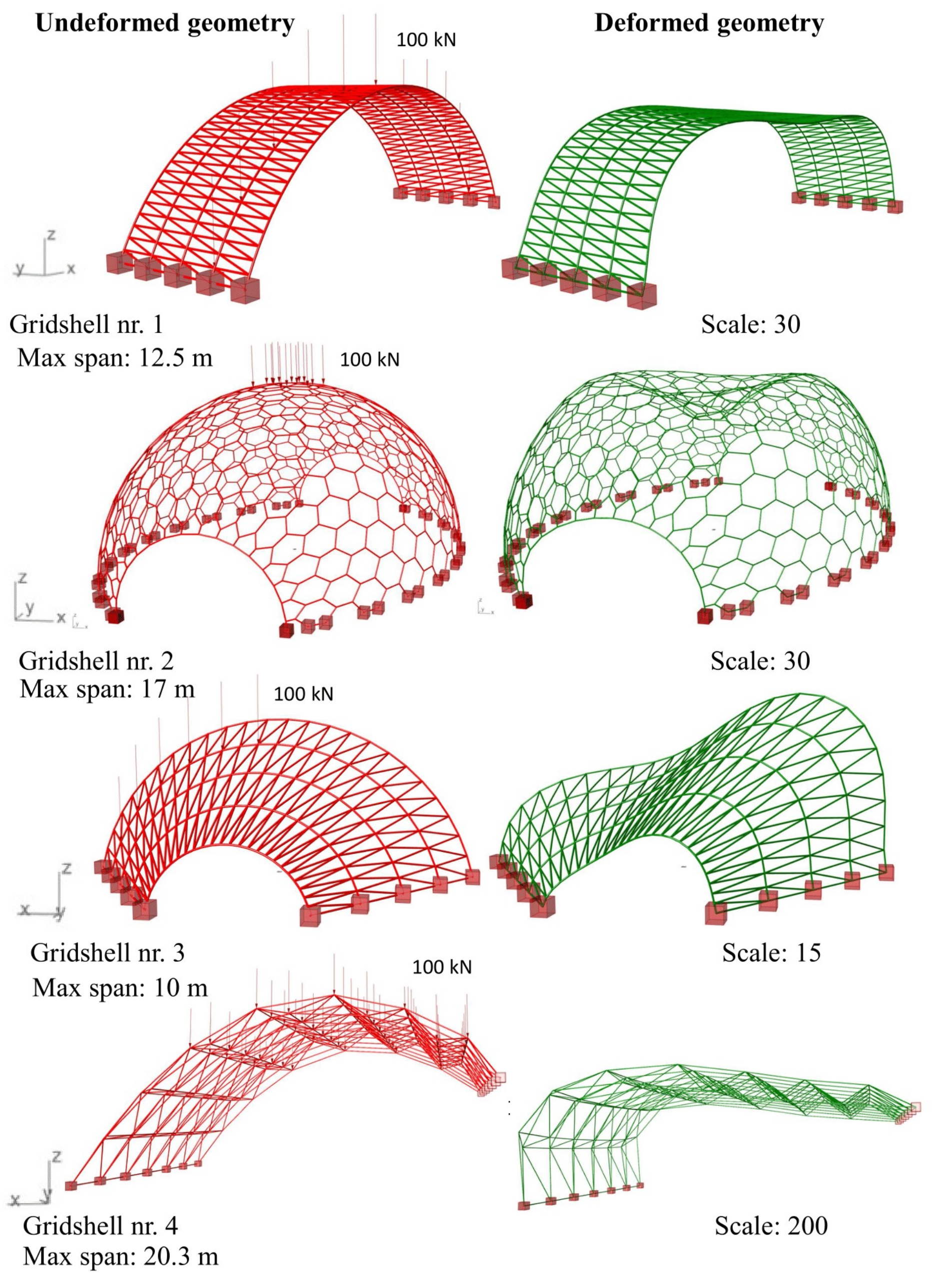


Figure 4.11: Three different 3D Gridshell structures, undeformed and deformed.

Table 4.28: Dimensions and maximum displacements for the gridshells shown in Fig. 4.11.

Nr.	Max. Span	Max. height	Load	Max. disp z - V3D	Max. disp z - Karamba3D
1	12.5 m	4.0 m	11x100 kN	-30.921 mm	-30.998 mm
2	17.0 m	4.0 m	14x100 kN	-208.061 mm	-208.578 mm
3	10.0 m	8.5 m	6x100 kN	-41.553 mm	-41.689 mm
4	20.3 m	9.1 m	58x100 kN	-11.752 mm	-11.753 mm

4.3.5 Benchmark #5: Dynamic Solver V3D

Benchmark #5 deals with the validation of the dynamic solver that has been developed. This will be done by applying theory and comparing it with other numerical programs such as Robot Structural Analysis. The V3D calculates displacements by performing the Newmark average acceleration method, and natural periods by solving the Eigenvalue problem. To validate this, a four-meter-long beam is fixed on both ends and is given an initial displacement to compare time history for displacements and natural frequencies that V3D calculates. Figure 4.12 displays the beam with the different degrees of freedom T_z, T_x, R_y, which will, in turn, be given an initial displacement.

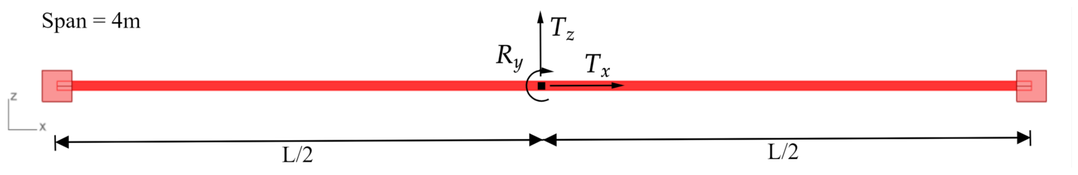


Figure 4.12: Fixed beam 4m long beam, used for verifying the dynamic solver.

The beam has a solid circular cross-section with a diameter of 25mm. Figure 4.13 shows the time history for the beam excited by an initial displacement in the middle node of -82,7mm, in the z-direction, T_z. This value is the static response for the beam when a point load of -1 kN is acting in the middle. Damping is set to 5%.

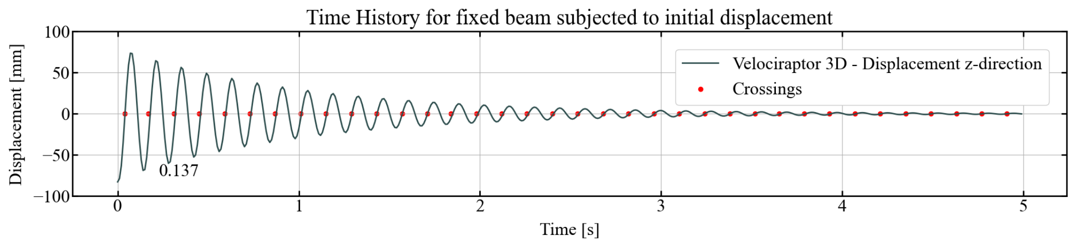


Figure 4.13: Time history for displacement in the z-direction. (T = 5 s, dt = 0.01 s, ζ = 5 %, d_0 = -82.7 mm)

Figure 4.14 shows the time history for the same beam, but in this case with an initial displacement of 97.0mm in the middle node in the x-direction, T_x. This is the static response for a load of 10000kN acting in the x-direction of the beam. Damping is set to 0.05%.

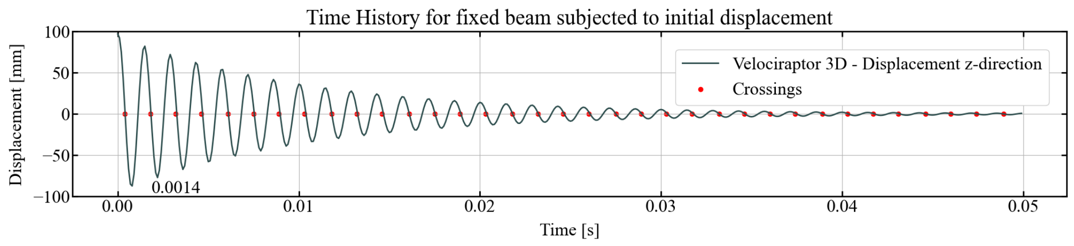


Figure 4.14: Time history for displacement in x-direction (T = 0.05 s, dt = 0.0001 s, ζ = 0.05 %, d_0 = -97.0 mm)

Figure 4.15 shows the time history for the same beam, but in this case with an initial displacement of 0.6208 radians in the middle node in the y-direction, R_y. This is the static response for a moment of 10kNm acting in the y-direction of the beam. Damping is set to 0.05%.

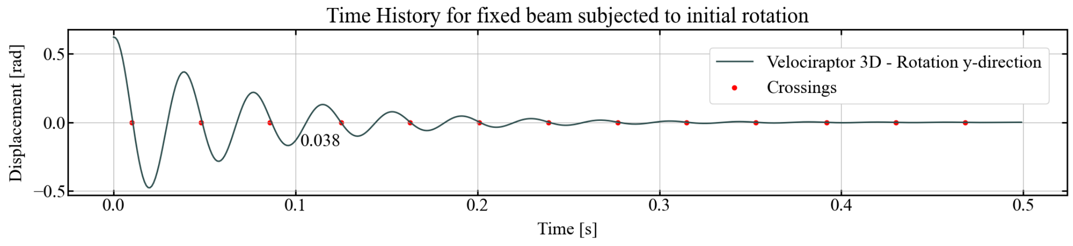


Figure 4.15: Time history for rotation in the y-direction (T = 0.5 s, dt = 0.001 s, ζ = 5 %, r_0 = 0.6208 rad)

Equation 4.2 shows the formulation of the natural and natural damped period, respectively [7]. With these one can calculate the system's natural period and validate this against what the natural periods V3D calculates.

$$T_n = T_D \cdot \sqrt{1-\zeta^2}, \quad \omega_n = \frac{2\pi}{T_D} \tag{4.2}$$

As seen in Figure 4.13 the damped period T_D is 0,137 s, in Figure 4.14 it is 0,0014 s and in Figure 4.15 0.0380 s. By using equations 4.2 the natural frequency can be calculated from the time histories. This is done in table 4.29. As seen the value coincide, which validate the method developed. It is not done, but the same example can be done in the y- direction.

Table 4.29: Validation of the Newmark method against the Eigenvalue problem.

Direction	T_d	T_n	w_n (Newmark)	w_n (Eigenvalue prob.)	Diff.
Translation Z	0.1370 s	0.1368 s	45.930r/s	45.936 r/s	0.013 %
Translation X	0.0014 s	0.0014 s	4487.990r/s	4479.252 r/s	0.195 %
Rotation Y	0.0380 s	0.0380 s	165.554r/s	165.624 r/s	0.042 %

The next step is to validate against external software, Robot Structural Analysis. An identical structure is made and given an initial displacement, the time history for the displacement of the middle node is plotted together with the time history calculated by V3D. The structure is again the fixed beam with the same properties as before. Figure 4.16 shows the two time histories and as seen, they coincide very well.

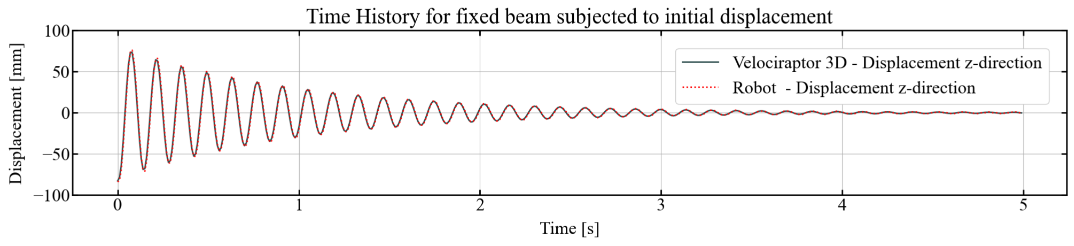


Figure 4.16: Comparison plot: Time history for the displacing V3D vs Robot structural analysis (T = 5 s, dt = 0.01 s, ζ = 5 %, d_0= -82,7mm)

Lastly, the time history displacements of the beam in Figure 4.13 has been animated in Rhino7 and can be seen by following this link: Time History Beam. It displays the beam being released with a displacement of -82.7 mm and vibrating before coming to an end. The displacement is scaled with a factor of 10, and every frame lasts 50 milliseconds.

4.4 VelociraptorFEA Discussion

Convergence is a fundamental concept in finite element analysis, encompassing the accuracy and reliability of the numerical process. It involves not only ensuring that the analysis approaches the exact solution but also understanding the rate at which this convergence occurs. Achieving a balance is crucial, where accuracy is maintained while minimizing computational time to avoid excessive computations.

In the initial benchmarks showcasing the static solver, both Velociraptor2D and Velociraptor3D demonstrated good results when compared to Karamba3D.

Table 4.24, showing a comparison between V3D and K3D for the truss bridge, shows a difference in displacement of 0.00% in all DOF. While V3D outputs displacements in millimetres, K3D outputs displacements in meters. When multiplying the K3D results by 1000 to get millimetres, it is only possible to get 3 significant figures, which is why V3D was rounded to 3 significant figures. By having more significant figures, there would probably be a difference in displacements, explaining the difference in forces.

In the final benchmark, time history plots were compared against eigenfrequencies, effectively comparing the Newmark average acceleration implementation to the Eigenvalue problem solver. This analysis yielded valuable insights, with values aligning well. The time history plot was also compared directly to the numerical tool Robot Structural Analysis and proved to be well functioning.

5 VelociraptorAI

This chapter about VelociraptorAI discusses how arch and truss structures can be represented as graphs, and how Graph Neural Network (GNN) models can be trained to make certain time-saving predictions. The GNN models are trained and tested on data generated by the VelociraptorFEA plugins discussed in Section 2. Modules from PyTorch [26] and PyTorch Geometric [37] are utilized to predict nodal moments in 2D arches and nodal time histories for 3D truss bridges.

The PyTorch Geometric modules are used to represent the data as graphs and are further used together with the PyTorch Neural Network modules to create the Graph Neural Networks ArchNN and TrussNN. The code for the two networks can be found in the VelociraptorAI Git Repository: VelociraptorAI.

5.1 ArchNN Method

This section introduces ArchNN, the first of two machine learning models to be discussed in this chapter. ArchNN leverages the power of GNNs to model and learn from structural arches represented as graph-structured data. These arches are composed of 19 elements and 20 nodes and are subjected to a uniformly distributed vertical load. Figure 5.1 shows an example of such an arch, with a length of 100 and a height of 35.

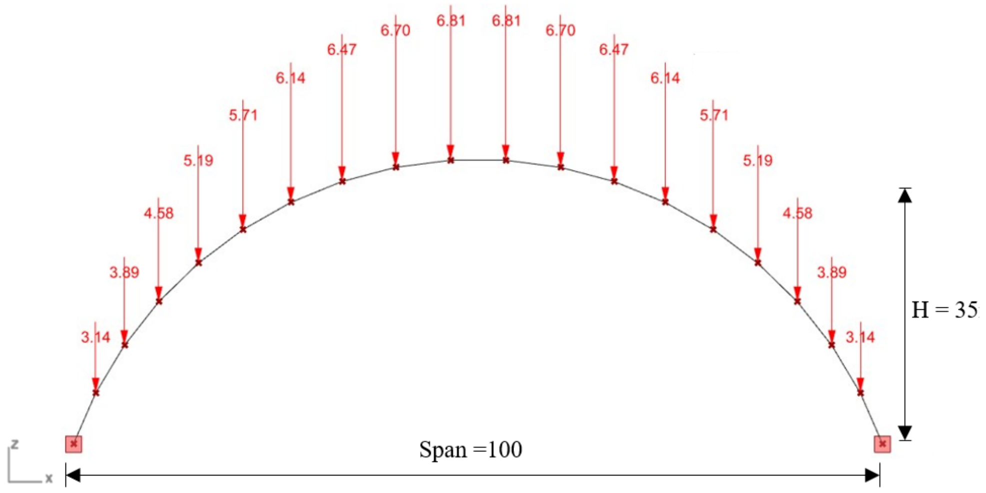


Figure 5.1: Arch

The ArchNN model will be developed using dimensionless values, this is to make a proof of concept and make it easier to adapt the wanted units afterwards. The load, denoted as q_{Ed}, is a uniformly distributed load of 10 per unit length, applied horizontally and then lumped to the nodes. This implies that nodes positioned at the top of the arch, where it exhibits a flatter structure, receive a larger portion of the load compared to nodes situated at steeper sections of the arch. This is illustrated in Figure 5.1.

The moment diagram, including extreme values, corresponding to the arch depicted in Figure 5.1 is shown in Figure 5.2.

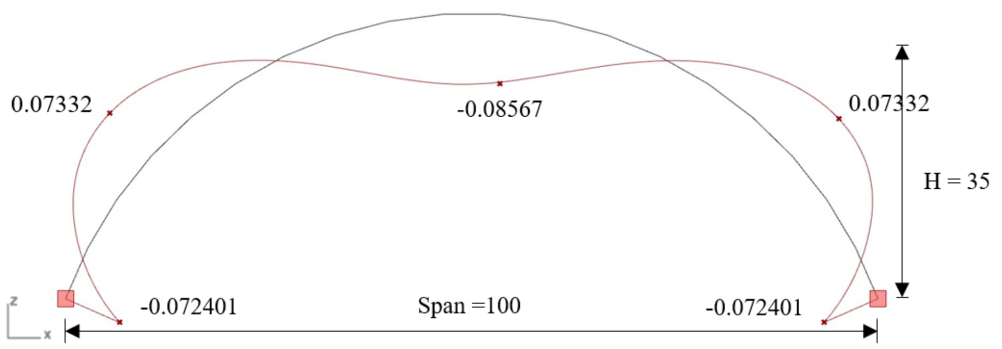


Figure 5.2: The moment diagram of the arch shown in Figure 5.1.

ArchNN's goal is to predict all the bending moments (M_y) for the arch's nodes. The following sections present the process for gathering and representing data, building and training the model, and designing the optimal model architecture. Following this, results are presented for the final model along with an example comparing Karamba3D, Velociraptor2D and ArchNN.

5.1.1 Data Representation

The PyTorch Geometric (PyG) *Data* object [37], shown with class parameters in Table 5.1, is used to create the graph representations with tensors as discussed in Section 3.3.3.

Table 5.1: PyTorch Geometric Data Object [37].

torch_geometric.data.Data(x, edge_index, edge_attr, y, pos, **kwargs)			
Parameter	Data Type	Description	Shape
x	torch.Tensor	The node feature matrix	[num_nodes, num_node_features]
edge_index	torch.Tensor	Graph connectivity matrix	[2, num_edges]
edge_attr	torch.Tensor	Edge feature matrix	[num_edges, num_edge_features]
y	torch.Tensor	Graph-level or node-level ground-truth labels	Arbitrary shape
pos	torch.Tensor	Node position matrix	[num_nodes, num_dimensions]
**kwargs	Optional	Additional attributes	Arbitrary shape

As shown in Table 5.1, the data types of the feature matrices are PyTorch Tensors. A set of vertices, or nodes, $V = \{v_1, \ldots, v_n\}$ are gathered in the node feature matrix, consisting of node numbering, x-coordinates and z-coordinates.

$$\text{PyG.Data.x} = \begin{bmatrix} 0 & 0.0 & 0.0 \\ 1 & 10.0 & 5.0 \\ 2 & 20.0 & 10.0 \\ 3 & 30.0 & 5.0 \\ 4 & 40.0 & 0.0 \end{bmatrix} \tag{5.1}$$

Equation 5.1 shows an example of how the *x* entry of the PyG.Data object would look like describing a forty-long and ten-tall 5-node arch. For the connectivity of the arch, double-directed edges have been opted for, which is equivalent to having undirected edges [37]. The graph connectivity array is saved in the *edge_index* tensor of the Data object, which for the 5-node arch would be:

$$\text{PyG.Data.edge\_index} = \Big[\big[0,1\big],\big[1,0\big],\ldots,\big[3,4\big],\big[3,4\big]\Big] \tag{5.2}$$

Because the arches used for generating training and testing data all have the same cross-sections and material properties, no edge feature matrix is included. The nodal moments are placed as node-level ground truth labels in the *y* tensor:

$$\text{PyG.Data.y} = [-3.1, 0.12, -1.8, 0.12, -3.1] \tag{5.3}$$

5.1.2 Dataset Generation

To develop a rigid model, it's necessary to generate a sufficiently large dataset. The Velociraptor2D Grasshopper plug-in is used for generating FEA results for a set of arches. The plugin *Colibri* is used for generating sets of different arches [41]. Two sliders are set to control the length and height of the arch. One support is set to x- and z-coordinates (0, 0) while the other moves along the x-axis. The height varies between the negative half of the length, to the positive half of the length, producing both arches, straight beams, and sagging beams. All of them are referred to as arches when referencing the ArchNN model. The necessary information, including node numbering, coordinates and nodal moments, are written to text files. The Colibri component is shown below in Figure 5.3. For this setting, the component produces 3201 different arch geometries.

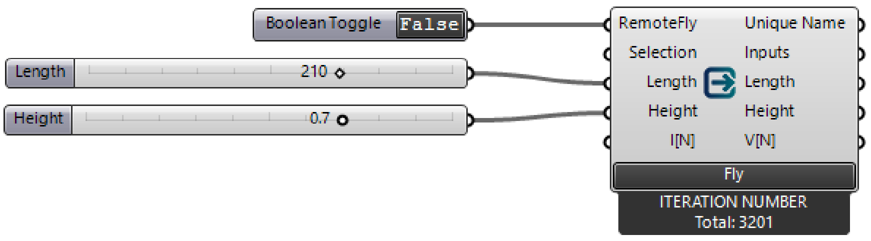


Figure 5.3: *Colibri* component for generating datasets.

After the text file is produced, the data is processed, split up, shuffled, and the PyG.Data objects are created. One part of the dataset is set aside before shuffling, to have test data outside the trained field. The rest of the data is split up into training and validation data after shuffling.

5.1.3 Procedure for Evaluating Model Performance

When building machine learning models it is crucial to evaluate the machine learning model using diverse testing methods to thoroughly understand its predictive capabilities and ascertain the reliability of its predictions.

In the evaluation of the trained ArchNN model's performance, the training loss and test loss are first monitored. Subsequently, it is important to further test the model performance, to get an insight into how well the model is predicting nodal moments. To do this, the average percentage error, Equation 4.1, is calculated for test arches. While loss functions like MSE loss and L1 loss both quantify the difference between the predictions and the labels, the average percentage error provides an indication of the prediction error's magnitude. To illustrate, a seemingly minor deviation in a prediction of 0.03 might signify a 100% error if the label is 0.03 and the prediction is 0.06.

Recognizing that predictions of smaller numbers can lead to disproportionately large average percentage errors, the results are presented with a more nuanced view. The smaller numbers often occur when the moment is shifting sign, which can happen four times for some arches in the generated datasets, as illustrated in Figure 5.2. To test if the few worst errors have a significant influence on the overall error, an average percentage error is computed after excluding the two most extreme errors. Lastly, an average percentage error specific to the nodes situated around the arch's midpoint is presented to test the model's performance in this area, which is often crucial for the design,

5.1.4 Model Definition

Procedure 5.3 presents the current architecture of the ArchNN model. Section 5.1.6 will experiment with various alterations to this structure in pursuit of enhanced performance. The ArchNN code includes a constructor, a *reset_parameters* method that is called during initialization to reset the convolutional layers' parameters, and a *forward* method that is called when the model is doing a prediction, executing forward propagation. This NN consists of three linear layers (L) [25] and 5 FeaStConvolutional layers (C) [38]. This architecture can be represented as L16-C32-C64-C128-C256-C128-L64-L1, where the numbers indicate the output size of each layer.

```
# Importing necessary libraries
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn.conv import feast_conv

class ArchNN(torch.nn.Module):
    def __init__(self, in_channels, heads, t_inv = True):
        super(ArchNN, self).__init__()

        # This fully connected layer transforms the input to have 16 features
        self.fc0 = nn.Linear(in_channels, 16)

        # FeastConv convolutional layers
        # Each layer doubles the number of features until reaching 256
        self.conv1 = feast_conv.FeaStConv(16, 32, heads=heads, t_inv=t_inv)
        self.conv2 = feast_conv.FeaStConv(32, 64, heads=heads, t_inv=t_inv)
        self.conv3 = feast_conv.FeaStConv(64, 128, heads=heads, t_inv=t_inv)
        self.conv4 = feast_conv.FeaStConv(128, 256, heads=heads, t_inv=t_inv)

        # This layer halves the number of features to 128
        self.conv5 = feast_conv.FeaStConv(256, 128, heads=heads, t_inv=t_inv)

        # Fully connected layers reduce the dimensionality of the output
        self.fc1 = nn.Linear(128, 64)
        self.fc2 = nn.Linear(64, 1)

        # Resets the parameters of the convolutional layers
        self.reset_parameters()

    # The reset_parameters function
    def reset_parameters(self):
        self.conv1.reset_parameters()
        self.conv2.reset_parameters()
        self.conv3.reset_parameters()
        self.conv4.reset_parameters()
        self.conv5.reset_parameters()

    # The forward function passes the data through the neural network
    def forward(self, data):
        # Getting the node_feature tensor and edge_index tensor
        x, edge_index = data.x, data.edge_index

        # node_feature tensor is passed through the initial fully connected layer
        x = F.leaky_relu(self.fc0(x))

        # node_feature and edge_index tensors are passed through the 5 convolutional layers
        x = F.leaky_relu(self.conv1(x, edge_index))
        x = F.leaky_relu(self.conv2(x, edge_index))
        x = F.leaky_relu(self.conv3(x, edge_index))
        x = F.leaky_relu(self.conv4(x, edge_index))
        x = F.leaky_relu(self.conv5(x, edge_index))

        # The output of the convolutions is passed through two fully connected layers
        x = F.leaky_relu(self.fc1(x))
        x = self.fc2(x)

        # This step removes singleton dimensions
        x = torch.squeeze(x, dim=1)

        # The output of the network is returned
        return x
```

Procedure 5.3: ArchNN class that defines the architecture of the model.

When creating a new model, the constructor in Procedure 5.3 is used with *in_channels* set to the number of features in the node feature matrix which, for the datasets used in this research, is 3 as shown in Figure 5.1. The *t_inv* boolean argument, which is set to *True* is an option to add self-loops to each input node in the graph. This ensures the preservation of the node's own features because the node will consider its own features in addition to its neighbours' features.

5.1.5 Training and validation

Procedure 5.4 shows the *run* function, which controls the training process by calling the *train* and *test* functions for every epoch. After finishing one epoch, the function prints out validation information using the *print_info* function, also shown in 5.4.

```python
# Importing necessary libraries
import time
import torch
import torch.nn.functional as F

# This function prints the current state of training
def print_info(info):
    message = ('Epoch: {}/{}, Duration: {:.3f}s,'
               'Train Loss: {:.4f}, Test Loss:{:.4f}').format(
                   info['current_epoch'], info['epochs'], info['t_duration'],
                   info['train_loss'], info['test_loss'])
    print(message)

# This is the main function to run the training and testing of the model
def run(model, train_loader, test_loader, epochs, optimizer, device):
    # Initialize lists to hold loss values for each epoch
    train_losses = []
    test_losses = []

    # Loop over each epoch
    for epoch in range(1, epochs + 1):
        t = time.time()

        # Train the model on the training data
        train_loss = train(model, train_loader, optimizer, device)

        # Calculate the duration of the training
        t_duration = time.time() - t

        # Test the model on the test data
        test_loss = test(model, test_loader, device)

        # Gather information about the current epoch
        eval_info = {
            'train_loss': train_loss,
            'test_loss': test_loss,
            'current_epoch': epoch,
            'epochs': epochs,
            't_duration': t_duration
        }

        # Print the current state of training
        print_info(eval_info)

        # Append the losses for this epoch to the lists
        train_losses.append(train_loss)
        test_losses.append(test_loss)

    # Return the lists of losses after training is finished
    return train_losses, test_losses
```

Procedure 5.4: Runner function and print_info function.

The training and test datasets are passed through the *train* and *test* functions, respectively, for every epoch. The function for training, Procedure 5.5, first sets the model to training mode before it iterates through the batch of training data it has been given. The ArchNN training processes discussed in the following sections are executed with batches of one arch. This means that for every PyG.Data object (arch), the gradients are set to zero, the loss between a prediction and its label is computed (here with mean squared error (MSE) loss), the gradients are computed with backpropagation, and the weights are adjusted by an optimizer [28].

```python
# This function trains the model for one epoch
def train(model, train_loader, optimizer, device):
    model.train()

    total_loss = 0

    # Loop over each batch in the training data
    for idx, data in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data.to(device))

        # Compute the mean squared error loss
        losses = F.mse_loss(output, data.y.to(device), reduction='none')

        # Compute mean loss for backpropagation
        loss = losses.mean()

        # Backpropagate the gradients
        loss.backward(retain_graph=True)

        # Update the model parameters
        optimizer.step()

        # Add the loss for this batch to the total loss
        total_loss += loss.item()

    # Return the average loss for this epoch
    return total_loss / len(train_loader)
```

Procedure 5.5: Training method.

After training, a validation test is performed on a test data batch using the *test* function, shown in Procedure 5.6. Here, the model is set to evaluation mode, disabling gradient tracking which is not needed during testing [28]. It then iterates through the PyG.Data objects and reports the MSE loss.

```python
# This function tests the model
def test(model, test_loader, device):
    model.eval()

    total_loss = 0

    # Ensure no gradients are calculated
    with torch.no_grad():

        # Loop over each batch in the test data
        for idx, data in enumerate(test_loader):
            out = model(data.to(device))

            # Calculate the loss for this batch and add it to the total loss
            total_loss += F.mse_loss(out, data.y.to(device)).item()

    # Return the average loss for the test data
    return total_loss / len(test_loader)
```

Procedure 5.6: Testing method.

The model architecture given in Procedures 5.3, 5.4, 5.5 and 5.6 serves as a baseline for a functional ArchNN model. Section 5.1.6 will be dedicated to conducting rigorous tests and adjustments on this architecture with the goal of enhancing predictive accuracy.

5.1.6 Designing the Neural Network Architecture of the ArchNN

The approaches described in this section build upon the theoretical foundations laid out in Section 3.2.2. This section explores the key concepts and considerations inherent in designing a neural network model's architecture, providing the necessary background for the methodological decisions taken here.

The model's architecture is chosen through various tests, aiming to select the most appropriate loss function to accurately quantify errors and the activation functions that enhance the model's learning potential. Furthermore, determining other significant aspects such as the number of layers along with their depth, the number of training epochs, and the size of both the test and training data, are critical considerations. These aspects, which will be examined and fine-tuned in this chapter, significantly influence the learning capacity of the model and contribute to maximizing its performance.

Some hyperparameters are not changed through the following designing process, such as the learning rate which is set to 0.0001 since it early on proved as the best choice. Also, the number of FeaStNet heads was set to 8, after comparing with the paper on Graph Surrogate Models *Towards Reusable Surrogate Models: Graph-Based Transfer Learning on Trusses* [47]. The choice of optimizer is ADAM which is a method for Stochastic optimization, that is built into Pytorch[16].

Determining the Architecture's Activation Functions

As discussed in Section 3.2.3, ReLU is a popular choice for convolutional nets. After some early quick tests, it was obvious ReLU and variations of ReLU gave the best results. ELU, ReLU and LeakyReLU all gave promising results, but ReLU and LeakyReLU performed the best. The model was therefore further tested for these two activation functions. Figure 5.4 shows the training of the architecture given in Procedure 5.3 over 100 epochs with a training set consisting of 1000 arches and a validation set consisting of 250 arches. The length of the arches varies between 100 to 175, and the height is between -0.5 and 0.5 times the length.

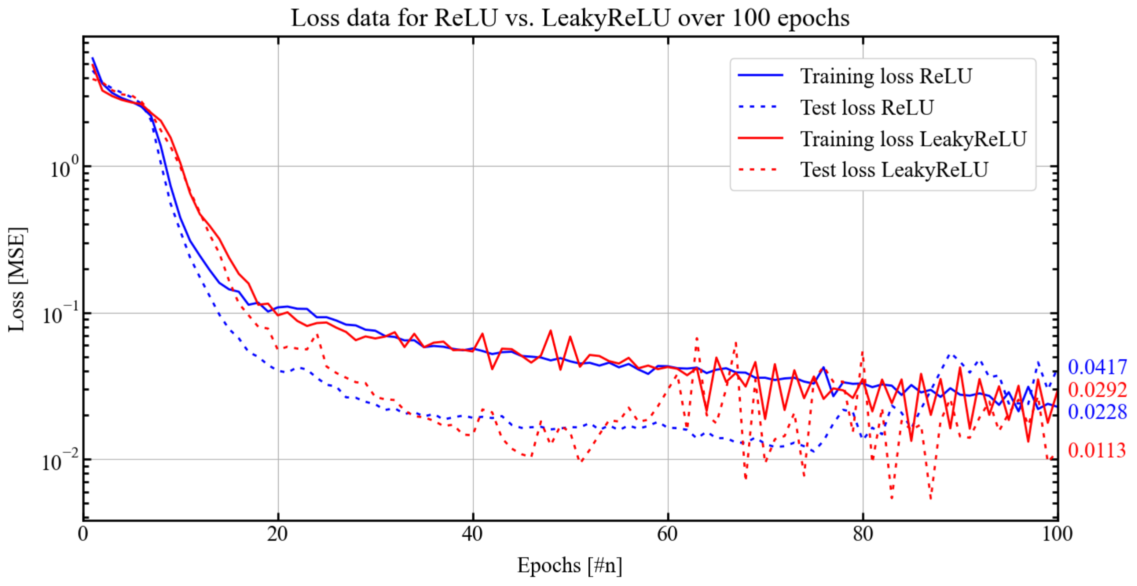


Figure 5.4: Comparison plots: LeakyReLU vs. ReLU.

To compare the two different activation functions, the MSE test and training loss are logged during training and plotted for subsequent analysis. Figure 5.4 shows that the test and training accuracy improves for both ReLU and LeakyReLU. Comparing the final epoch training and test loss values shown at the end of each graph, the model trained with LeakyReLU demonstrates slightly better accuracy than the one trained with ReLU. The LeakyReLU-trained model achieved a training loss of 0.0292 and a test loss of 0.0113, while the ReLU-trained model recorded a training loss of 0.0417 and a test loss of 0.0228. Examining the plots, it's clear that during the first 20 epochs, both models undergo rapid learning. After this point, the rate of improvement tapers of, and the curves reveal a growing pattern of oscillation. One possible explanation for this could be overfitting, as discussed in Section 3.2.2. However, this seems unlikely given that the model's performance on the training data does not appear to be significantly better than its performance on the test data. A noteworthy observation can be made from the test loss graph of the ReLU-trained model. After around the 80th epoch, the test loss begins to increase, a trend that persists towards the end of training. This pattern might indicate potential overfitting in the model trained with ReLU. To investigate this further, the training process is extended to 300 epochs in Figure 5.5. This will provide more data to examine this trend and determine whether it's a sign of overfitting or not.

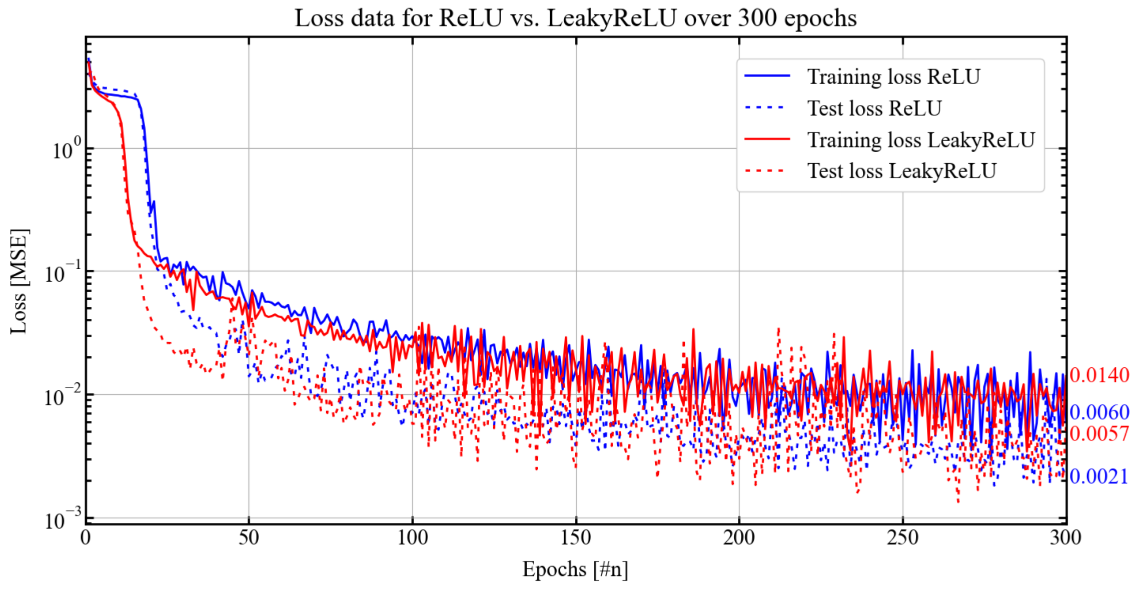


Figure 5.5: Extended training performance: ReLU vs. LeakyReLU.

The learning curves displayed in Figure 5.5 show the same patterns observed in Figure 5.4, suggesting that overfitting is not an overall problem for either activation function. The observed oscillations in the learning curves could potentially be attributed to a variety of factors, including the chosen batch size, learning rate, and others. However, before considering a detailed exploration into these factors and their potential effects, the model's performance will be further tested and compared despite these oscillations.

In addition to comparing the MSE loss, the average percentage error (Equation 4.1) is calculated to further test the performance of the models, as discussed in Section 5.1.3. Figure 5.6 demonstrates the average percentage loss plots for the same training process as displayed in Figure 5.5.

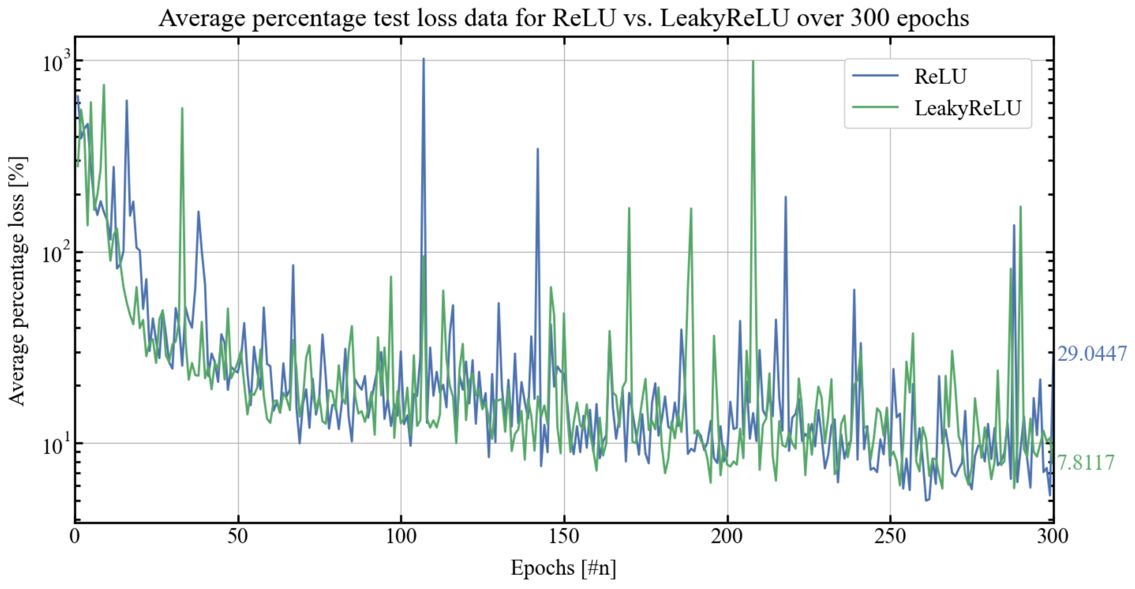


Figure 5.6: Comparison plots: average percent error for ReLU vs. Leaky ReLU.

Figure 5.6 depicts the average percentage test loss for both models, showing a general decreasing trend alongside significant oscillations. The pronounced fluctuations in average percentage test loss may arise from the inflated impact of errors on values closer to zero. Ignoring these oscillations, the final average percentage test loss for the LeakyReLU model substantially outperforms the ReLU model, even though their Mean Squared Error (MSE) values are comparable. Specifically, the average percentage loss for the model trained with LeakyReLU stands at 7.8117, nearly four times lower than the 29.0447 observed with the ReLU model. However, it can be observed that the two models have significantly similar learning rates, and have intertwined oscillations. Unfortunately, the high value of the resulting average percentage loss for the ReLU-trained model corresponds to the peak of one of the discussed oscillations. Introducing early stop when the average percentage loss reaches a certain value can serve as a potential solution to prevent this issue.

In conclusion, both the ReLU-trained model and the LeakyReLU-trained model demonstrated comparable performance results. Despite their similar results, it's not immediately clear which activation function is superior for this specific task. However, an important observation made during the training process was that the ReLU model took a significantly longer time to train compared to the LeakyReLU model. Figure 5.7 shows the time duration plot of each epoch for the same training process displayed in Figure 5.4. Both models are trained on the same CPU.

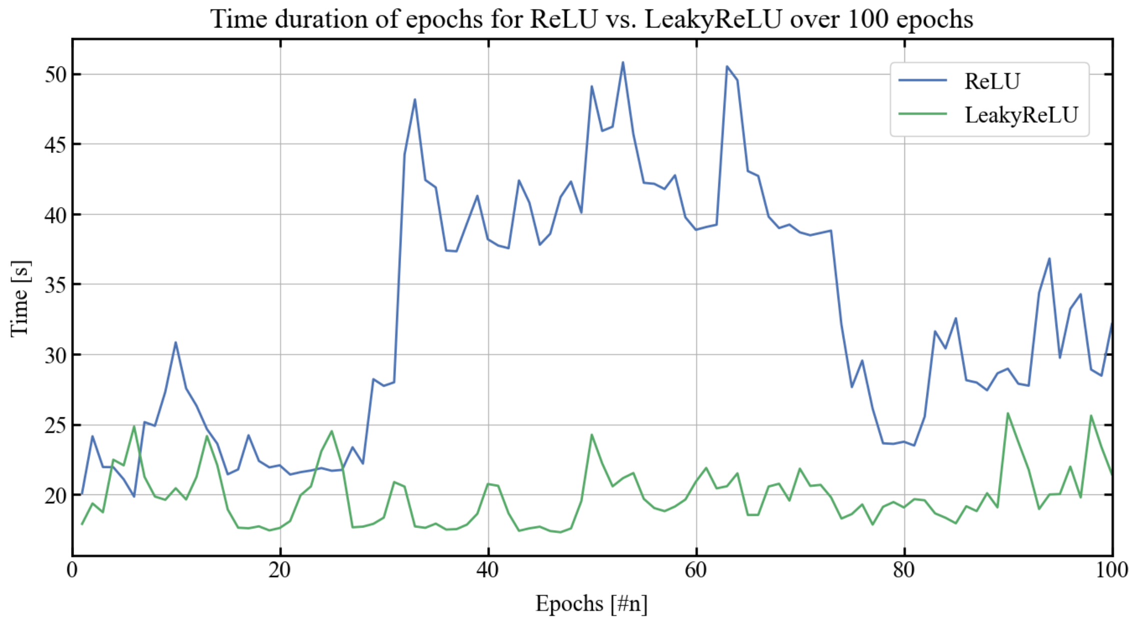


Figure 5.7: Comparison plots: training epochs' time duration for ReLU vs. LeakyReLU

Given the difference in training time, coupled with the potential benefits of LeakyReLU, such as its ability to handle the *dying ReLU problem* discussed in Section 3.2.3, the LeakyReLU has been chosen as the hidden layer activation function in the subsequent sections dedicated to enhancing model performance.

Determining the Architecture's Loss Function

Section 3.2.2 presented different loss function options, and discussed why the choice of loss function is crucial for the model's performance. The ArchNN model predicts a range of values, thereby requiring an average prediction error. The loss functions considered for this task were L1, MSE, and MSPE. Preliminary testing revealed that L1 and MSE displayed superior learning capabilities for the model; therefore, additional tests were focused on these two options. Efforts were also made to train the model using the average percentage loss (employed as a test method for the ArchNN, as detailed in 5.1.3) as the loss function. Nevertheless, this approach didn't result in comparable learning efficiency as achieved with L1 and MSE.

Figure 5.8 demonstrates the training process for the model architecture specified in Procedure 5.3, utilizing both L1 and MSE loss. The L1 training and test losses align closely creating overlapping curves, a pattern not mirrored in the MSE loss scenario. The MSE test loss consistently maintains a value roughly half of the train loss. Several factors could contribute to this discrepancy, including the inherent characteristics of the loss functions, model complexity, and how the data subset is evaluated.

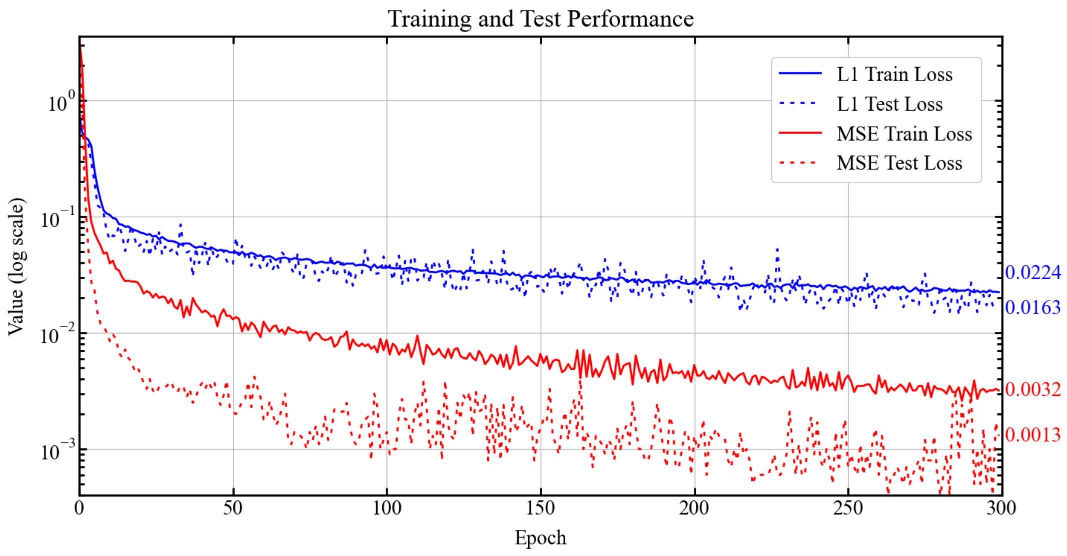


Figure 5.8: Graph showing train loss and test loss for L1 and MSE loss function during training process.

Compared to L1 loss, MSE loss is more sensitive to outliers, given its squaring of the difference. This indicates that the use of L1 loss allows for an equal weighting of errors, thereby enhancing the stability of the training process. This can be observed from the curves, showing less pronounced oscillations for the L1 loss.

To effectively assess the influence of these two loss functions on the predictive abilities of the model, it is imperative to consider the average percentage error. A comparison between the average percentage errors for models using L1 and MSE loss functions is presented in Figure 5.9.

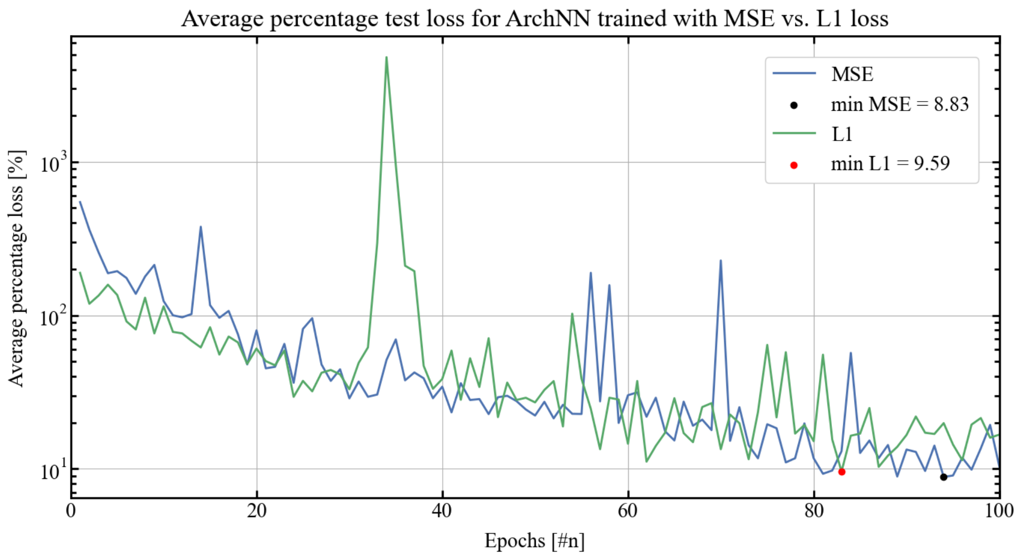


Figure 5.9: Comparison plots for models trained with the L1 and MSE loss functions.

The plots illustrate a slightly smaller average percentage loss for the test data when using MSE, with a value of 8.83%, compared to 9.59% when using L1 loss. While the difference may seem marginal, it contributes to a more accurate model, resulting in higher overall accuracy in predicting nodal moments. Therefore, based on these observations, MSE loss is chosen as the preferred loss function for the model.

Influence of Training Data Quantity on Prediction Accuracy

After examining the impact of different loss functions and activation functions on the model's performance, and determining this part of the model's architecture, the next step involves investigating the relationship between the quantity of training data and prediction accuracy. A well-trained machine learning model requires a balance between the amount of data used and the model's ability to learn from it, as discussed in Section 3.2.2. In this section, it is explored how increasing the quantity of training data could potentially enhance the model's predictive accuracy. This will provide valuable insights on how to strategically allocate data for optimal learning efficiency.

Figure 5.10 compares the average percentage test loss between two ArchNN models, one trained with 1000 example arches and the other trained with 4928 example arches. The architecture of these models incorporates the decisions made in the preceding sections. As can be seen in the plot, the plots illustrate similar shapes, although the model trained with more data consistently predicts with higher accuracy.

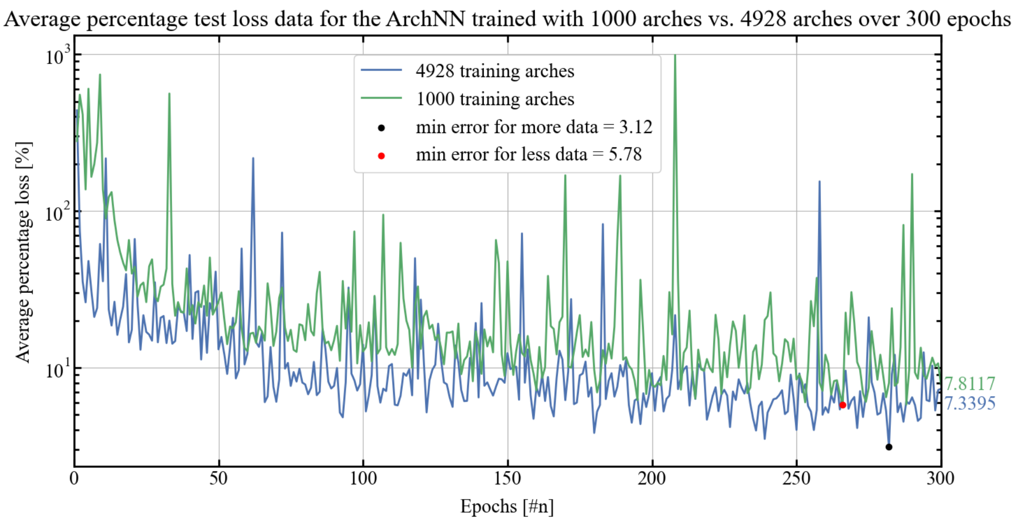


Figure 5.10: Testing the influence of data size on the model's predicting accuracy.

The final values shown at the end of each curve in Figure 5.10 are close in value, yet it is demonstrated how the minimum average percentage error is significantly better for the model trained with a larger dataset. This illustrates how important it is with early stopping when training the final ArchNN model. It is crucial to cease training when the model achieves an acceptable level of prediction accuracy, because of the significant oscillations.

It is necessary to mention the increase in duration each epoch has when the training data is expanded to this magnitude. The time duration average for one epoch for the model trained with 1000 arches is calculated to be 19.9 seconds, as approximately shown in Figure 5.7. The model trained with almost 4.9 times as many arches, has an average epoch time of 85.0 seconds, 4.3 times as long. This tells us that the time duration for each training epoch scales approximately linearly with the size of the training dataset.

Enhancing Training: Increasing Epoch Count and Implementing Early Stopping

It is clear in the plots presented in the three previous plots that the training accuracy increases with a growing amount of epochs. As mentioned, implementing early stopping on the data is crucial for ensuring that the best-trained model is returned. To do this, the model is set to train for 1000 epochs with the training data consisting of 4928 arches, and the *run* function described in Procedure 5.4 is configured to terminate once the average percentage loss descends below 3%. The MSE loss plots are shown in Figure 5.11 and 5.12.

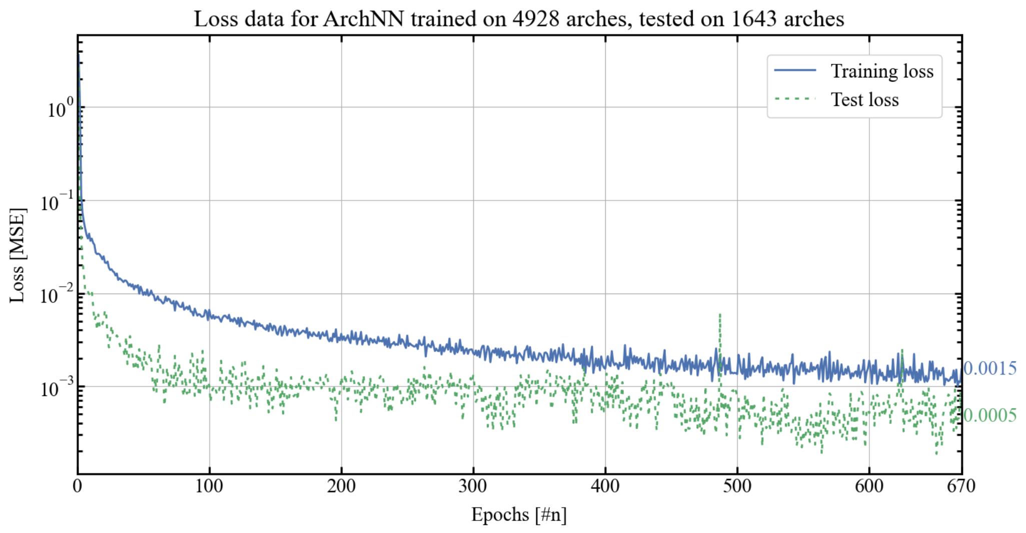


Figure 5.11: MSE training and test loss for the ArchNN with early stopping implemented.

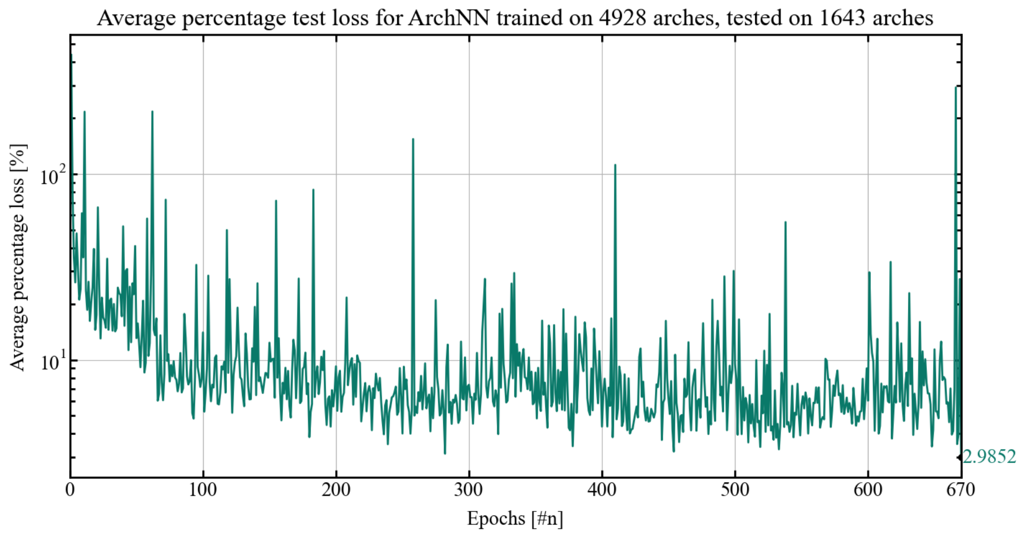


Figure 5.12: Average percentage test loss for the ArchNN with early stopping implemented.

As illustrated in Figures 5.11 and 5.12, the model terminated training after reaching an average percentage error of 2.9852 % in epoch number 670. This average percentage error is calculated for a test dataset consisting of 1643 test arches. Before further testing of this model, one last attempt is conducted to enhance the model's predicting accuracy.

Enhancing Training: Increasing Complexity

Increasing the model complexity can have both positive and negative effects, as discussed in Section 3.2.2. It can allow the model to capture more intricate patterns and relationships in the data, potentially leading to improved performance. However, increasing model complexity also increases the risk of overfitting, and increases computational requirements.

To check if increasing the ArchNN model's complexity enhances the model's learning capabilities, a 6th FeaStNet layer is added to Procedure 5.3. The model architecture for this model is L16-C32-C64-C128-C256-C512-C128-L64-L1. In Figure 5.13, the model from the previous section is compared with this new, more complex, model. Trained and tested on the same datasets and plotted for the first 600 epochs.

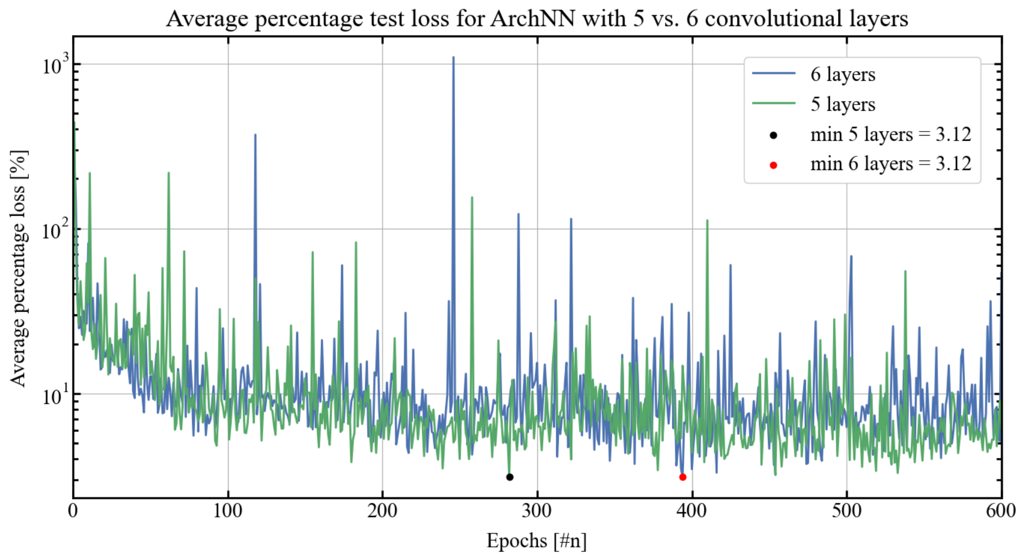


Figure 5.13: Adding complexity: testing how adding convolutional layers affects learning.

Figure 5.13 shows two noticeably similar curves. The plots illustrate that the same minimum is found for both models, with the less complex model arriving there after fewer epochs.

The impact of model complexity on processing time is clearly demonstrated when comparing the average time duration per epoch for two architectures: L16-C32-C64-C128-C256-C128-L64-L1 and L16-C32-C64-C128-C256-C512-C128-L64-L1. The first architecture, with five convolutional layers, took an average of 82.9 seconds per epoch over 600 epochs. In contrast, the second architecture, which includes an additional convolutional layer and more units, required a significantly longer average time of 187.4 seconds per epoch over 600 epochs. More convolutional layers and output units mean a greater number of weights and biases for the model to learn, directly contributing to the computational cost and consequently, the training time.

It is possible that the more complex model could potentially outperform the less complex one if a larger training dataset was utilized. However, such an approach would necessitate additional computational resources and invariably extend the duration of training. Given the acceptable prediction performance of the model with 5 convolutional layers, as indicated by the average percentage loss presented in Figure 5.12, there appears to be no compelling reason to delve further into enhancing model complexity.

5.2 ArchNN Results

After thorough testing, the final model architecture is the one shown in Procedure 5.3. The only alternation made to Procedures 5.4, 5.6 and 5.5 is implementing early stopping when the model achieved an average percentage test loss better than 3%, as explained in Section 5.1.6.

The model trained on 4928 arches with an average percentage loss of 2.99%, shown in Figure 5.12, is now further tested for an arch inside the training set, and compared with Karamba3D and Velociraptor2D. To ensure that the arch tested for is not one of the training arches, the length is set to 104.58 mm, because the length of the training arches were whole numbers. The height of the arch is 36.603 mm. The results are shown in Table 5.2 below.

Table 5.2: Model testing: Comparison between Karamba3D, Velociraptor2D and ArchNN.

Node nr.	Karamba3D	Velociraptor2D	ArchNN
Node 1	-0.08436	0.08436	0.08339
Node 2	0.01256	0.01256	0.01383
Node 3	0.06442	0.06442	0.06443
Node 4	0.07901	0.07901	0.07832
Node 5	0.06560	0.06560	0.06608
Node 6	0.03420	0.03420	0.03452
Node 7	-0.00530	-0.00530	-0.00590
Node 8	-0.04385	-0.04385	-0.04381
Node 9	-0.07401	-0.07401	-0.07376
Node 10	-0.09047	-0.09047	-0.08946
Node 11	-0.09047	-0.09047	-0.09087
Node 12	-0.07401	-0.07401	-0.07365
Node 13	-0.04385	-0.04385	-0.04342
Node 14	-0.00529	-0.00529	-0.00505
Node 15	0.03420	0.03420	0.03501
Node 16	0.06560	0.06560	0.06581
Node 17	0.07901	0.07901	0.07893
Node 18	0.06442	0.06442	0.06430
Node 19	0.01256	0.01256	0.01216
Node 20	-0.08436	-0.08436	-0.08530

Table 5.2 shows all 20 nodal moments for Karamba3D, Velociraptor2D and the ArchNN predictions. The average percentage error between the ArchNN and the V2D/Karamba3D moments shown in Table 5.2 is: *1.94 %*

Before some final test results are displayed, Figure 5.14 shows five different arch structures and their coherent moment diagram drawn for both the prediction from ArchNN and the labels calculated by Velociraptor2D. The prediction is drawn in blue and the labels in red, and as seen, the two coincide for all arches.

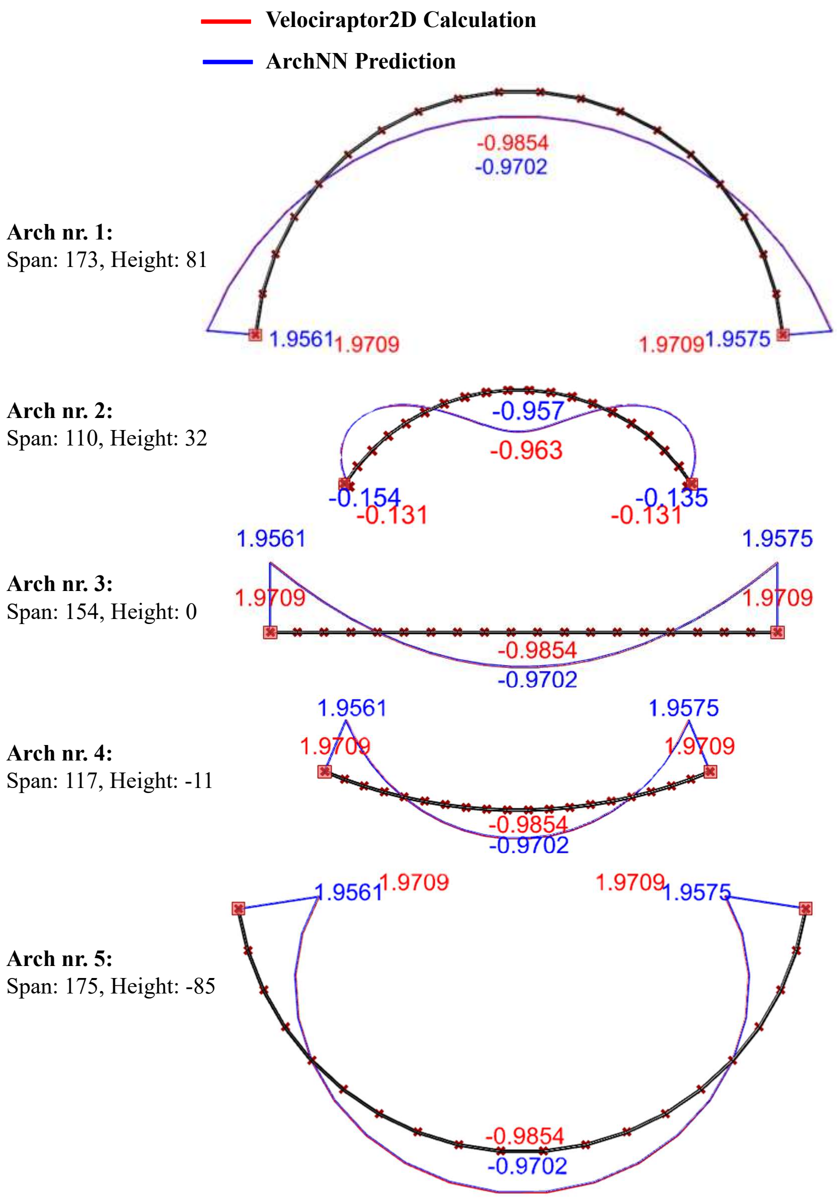


Figure 5.14: Moment diagram, both predicted and calculated, for five arch structures drawn for

To further test the ArchNN, the average percentage error is calculated for unseen data outside of the training domain. For this data, the arch lengths vary from 175 to 200, while for the training data they ranged from 100 to 175. The results in Table 5.3 are presented as explained in Section 5.1.3. The test results for 1000 arches inside the training domain are also shown.

Table 5.3: Model testing: ReLU vs. LeakyReLU

Nodes	1000 arches inside	286 arches outside
All nodes:	2.70 %	7.07 %
Adjusted error (two worst are removed):	0.67 %	2.86 %
Node 8, 9, 10, 11 (middle of arch):	0.43 %	2.60 %

5.3 ArchNN Discussion

In conclusion, the development and fine-tuning of the ArchNN model architecture have resulted in an effective and robust model, delivering predictions with satisfactory loss values. The model architecture established, as detailed in Procedure 5.3, has proven to be effective, demonstrating a respectable balance between model complexity and performance.

As Table 5.3 demonstrates, the model performs better on data inside the training domain, performing especially well on the nodes in the middle of the arch. Although the predictions made for the arches outside the training domain were still relatively strong predictions for the middle nodes, and after the two worst errors were removed, the lengths of these arches were maximum about 14% larger than arches inside the training domain. Moving further away from the training domain would produce worse results, and the ArchNN's training domain should in this case be expanded.

Our investigations into various activation functions revealed Leaky ReLU as the preferred choice for this model, due to its computational efficiency and potential benefits such as its ability to mitigate the 'dying ReLU' problem. Furthermore, after a comparison between L1 and MSE loss functions, we found that the MSE loss function offered a slight edge in reducing test average percentage loss, contributing to more accurate predictions.

The ArchNN offers an effective approach for estimating the nodal moments in an arch, presenting an opportunity for time savings in cases where full-scale FEA modelling is not required. In its current form, the ArchNN model requires *PyG.Data* objects for prediction. However, with clever integration, such as creating a plugin for a program like Grasshopper, the model can be adapted to accept more intuitive inputs like arch length and height. The requisite *PyG.Data* objects could then be generated within the plugin based on these inputs. Further enhancing its utility, the model could also be trained to handle a variety of support and load cases, broadening its applicability.

5.4 TrussNN Method

This section presents TrussNN, the final graph neural network model to be discussed in this thesis, focusing on learning and predicting the dynamic behaviour of various truss bridges. These bridges are composed of 68 beams and 32 connections, which correspond to 68 edges and 32 nodes in graph representation. The load, denoted as F_{Ed}, are two point loads of 1000 kN, applied vertically in the two top centre nodes of the bridge. These are applied to all the truss bridges. Figure 5.15 illustrates an example of a truss bridge with a span of 20 meters, a width of 5 meters, and a height of 5 meters. The same truss bridge is detailed in chapter 4.3.3.

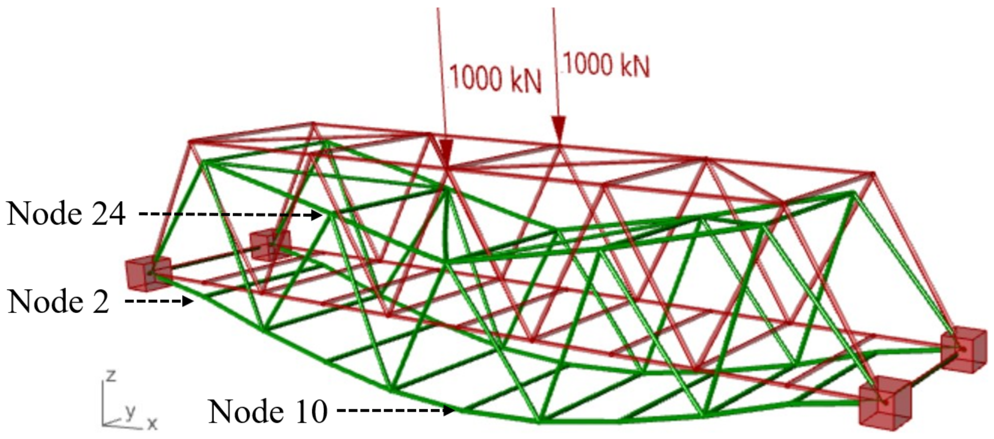


Figure 5.15: One example of a truss bridge used for training, undeformed geometry and deformed. Node 2, 10 and 24 are depicted and will be used throughout for examples.

The dynamic solver in Velociraptor3D calculates the displacements of all degrees of freedom of all nodes, given an initial force or displacement. The forces F_{Ed} in the truss above produce a maximal displacement of -3.5914 mm in the z-direction. By releasing the structure with this displacement, the time history for Node 10 with a damping of 5% becomes as Figure 5.16 shows.

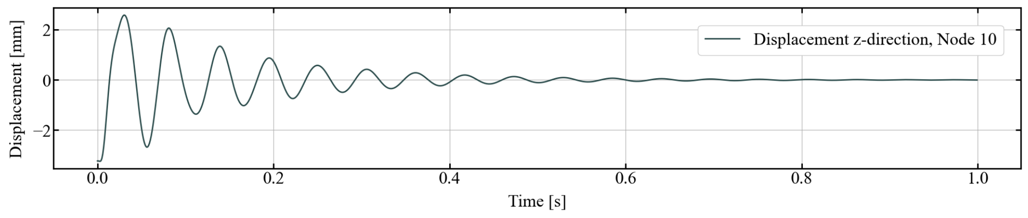


Figure 5.16: Time History for node 10.

It is also possible to animate the displacement for all nodes, which can be seen by following the link: Truss Bridge Animation. Note that, for simplicity, only the displacement in the z-direction is plotted during the animation. They are also scaled by a factor of 1000.

To generate a time history plot like the one shown above, an iterative process utilizing Newmark's method is performed. This process is time-consuming. Therefore, the objective of TrussNN is to directly predict parameters that can recreate the same time history plots. In the subsequent sections, the procedures for data collection and representation, model construction and training, and designing the optimal model architecture will be presented. Furthermore, the results obtained from the final model, along with specific examples, will be showcased. TrussNN is only trained for predicting truss bridge nodal time histories in the z-direction.

5.4.1 Dataset Generation

To generate the dataset, the Colibri plugin is once again utilized, but this time the data is written to a file using the TextToFile3D component within the Velociraptor3D framework. The parameters that undergo changes are the length and height of the bridge. The length varies from 20.0 meters to 30.0 meters in increments of 0.1 meters, while the height ranges from 3.5 meters to 5.5 meters also in increments of 0.1 meters. This process results in the generation of 2121 distinct truss bridge geometries. The Newmark average acceleration method is given a time step of 0.01 seconds over 2 seconds, producing 200 z-displacement values for every node of every bridge.

Considering the complexity of predicting the nodal time history functions directly, the TrussNN is tasked with learning the parameters of functions that fit these time histories. By using this approach, the computationally demanding task of predicting all 200 displacements for every node is avoided. Procedure 5.7 outlines the steps involved in determining the parameters of the functions that fit the displacement time histories generated by Velociraptor3D.

Displacement data is generated for all bridges:

$$Data = [Trussbridge_0, Trussbridge_2, ..., Trussbridge_{2121}]$$

Time history for each Node within each bridge:

$$displacements = [z_{t=0.01}, z_{t=0.02}, z_{t=0.03}, z_{t=0.04}, z_{t=0.05}, \dots, z_{t=2}]$$

A single damped sine wave, $f^s(t)$:

$$f^s(t) = A_1 \cdot e^{-\alpha_1 t} \cdot \sin(\omega_1 t + \phi_1)$$

with initial guess:

$$[A, \alpha, \omega, \phi] = [\frac{z_{max} - z_{min}}{2}, 1, 1, 1]$$

is fit to the displacement time history using SciPy, [8], giving the resulting optimized parameters:

$$[A, \alpha, \omega, \phi] = [A_s, \alpha_s, \omega_s, \phi_s]$$

The error between the fitted $f^s(t)$ and displacement values is calculated:

$$errors = displacements - f^s(t)$$

A new single damped sine wave is fitted to the error wave, providing $f^e(t)$ with the following optimized parameters:

$$[A, \alpha, \omega, \phi] = [A_e, \alpha_e, \omega_e, \phi_e]$$

A double damped sine wave, $f^d(t)$:

$$f^d(t) = A_1 \cdot e^{-\alpha_1 t} \cdot \sin(\omega_1 t + \phi_1) + A_2 \cdot e^{-\alpha_2 t} \cdot \sin(\omega_2 t + \phi_2)$$

with initial guess:

$$[A_1, \alpha_1, \omega_1, \phi_1, A_2, \alpha_2, \omega_2, \phi_2] = [A_s, \alpha_s, \omega_s, \phi_i, A_e, \alpha_e, \omega_e, \phi_e]$$

is fitted to the displacement time histories, providing the final optimized parameters:

$$[A_1, \alpha_1, \omega_1, \phi_1, A_2, \alpha_2, \omega_2, \phi_2]$$

Procedure 5.7: Procedure showing the method of obtaining the time history fitted function parameters, using damped sine waves [10] and the *curve_fit* function from SciPY optimize.

Initially, a single-damped sine wave was employed to model the displacement time history. However, the sine wave struggled to fit certain complex displacement patterns, like the upper graph in Figure 5.17. Consequently, a double-damped sine function was utilized to more accurately capture the behaviour of all nodes. Figure 5.17 shows two different approximations for a random truss bridge in the dataset, Node 2 closer to the support showcasing more complex movement and Node 10 in the middle. As seen, the approximation or fitting, for both cases, is sufficient.

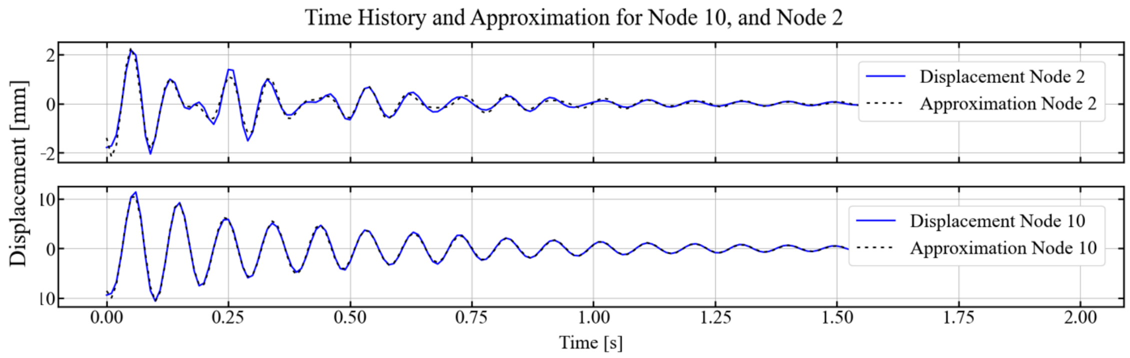


Figure 5.17: Approximation of two different time history functions

5.4.2 Data Representation

After the data is generated it can be graph represented by creating PyTorch Geometric Data Objects, after Table 5.1. Information about the graph's nodes, $V = \{v_1,...,v_n\}$ are gathered in the node feature matrix, consisting of node numbering, x-coordinates, y-coordinates, z-coordinates, and if the node is fixed or not. The *fixed* feature is 0 for free nodes and 1 for fixed nodes. Equation 5.4 shows the feature matrix PyG.Data.x.

$$\text{PyG.Data.x} = \begin{bmatrix} 0 & x_0 & y_0 & z_0 & fixed \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 32 & x_{32} & y_{32} & z_{32} & 0/1 \end{bmatrix} \tag{5.4}$$

The graph connectivity array is stored in the *edge index* tensor of the Data object, as for ArchNN. The EdgeIndex component in the V3D framework generates a list of connected beams, which is then saved to a file for convenient obtainment of these edges. The y entry of the PyG.Data.Object contains the previously obtained approximated parameters. The tensor representation is provided below, where the first row indicates the order, and the second and last rows indicate the labels for Node 2 and Node 10, respectively, as depicted plotted in Figure 5.17.

$$PyG.Data.y = \begin{bmatrix} A_{1_0} & \alpha_{1_0} & \omega_{1_0} & \phi_{1_0} & A_{2_0} & \alpha_{2_0} & \omega_{2_0} & \phi_{2_0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1.3017 & 64.9863 & -1.7354 & 1.9706 & 1.2837 & 90.1256 & -3.0551 & 2.9297 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 11.4185 & 65.2214 & -1.8621 & 2.0698 & -2.7622 & 85.1883 & -1.0588 & 6.2263 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \tag{5.5}$$

5.4.3 Procedure for Evaluating Model Performance

Given the complexity of the TrussNN model, simply representing its performance with an average percentage loss plot, similar to the method used for the ArchNN, might not provide a comprehensive illustration of its predictive accuracy. The TrussNN's task of predicting multiple displacement time histories, each characterized by 8 parameters, complicates the process of summarizing its performance into a single loss value. Consequently, instead of plotting an average percentage loss for the TrussNN model, this report will feature visual examples of the prediction results where the time history plots are compared between the TrusNN prediction, the fitted damped sine wave with label parameters, and the actual nodal time histories gathered from Velociraptor3D. This approach provides a more nuanced and concrete representation of the model's predictive capability, thereby facilitating a clearer understanding of its performance.

5.4.4 Model Definition

The TrussNN model architecture, detailed in Procedure 5.8, diverges from the ArchNN model in the augmentation of output neurons for the fifth convolutional layer. The number of output neurons is changed from 128 to 512. This enhancement aims to discern more intricate patterns inherent in the now more complex input data and labels.

```python
# Importing necessary libraries
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn.conv import feast_conv

class TrussNN(torch.nn.Module):
    def __init__(self, in_channels, heads, t_inv = True):
        super(TrussNN, self).__init__()

        # This fully connected layer transforms the input to have 16 features
        self.fc0 = nn.Linear(in_channels, 16)

        # FeastConv convolutional layers
        # Each layer doubles the number of features until reaching 256
        self.conv1 = feast_conv.FeaStConv(16, 32, heads=heads, t_inv=t_inv)
        self.conv2 = feast_conv.FeaStConv(32, 64, heads=heads, t_inv=t_inv)
        self.conv3 = feast_conv.FeaStConv(64, 128, heads=heads, t_inv=t_inv)
        self.conv4 = feast_conv.FeaStConv(128, 256, heads=heads, t_inv=t_inv)

        # This layer further increases the number of features to 512,
        # adding additional complexity to the model to capture intricate patterns
        self.conv5 = feast_conv.FeaStConv(256, 512, heads=heads, t_inv=t_inv)

        # Fully connected layers reduce the dimensionality of the output
        self.fc1 = nn.Linear(512, 64)
        # This layer sets the output size to 8, because of the 8 parameters
        # to be predicted
        self.fc2 = nn.Linear(64, 8)

        # Resets the parameters of the convolutional layers
        self.reset_parameters()

    # The reset_parameters function
    def reset_parameters(self):
        self.conv1.reset_parameters()
        self.conv2.reset_parameters()
        self.conv3.reset_parameters()
        self.conv4.reset_parameters()
        self.conv5.reset_parameters()

    # The forward function passes the data through the neural network
    def forward(self, data):
        # Getting the node_feature tensor and edge_index tensor
        x, edge_index = data.x, data.edge_index

        # node_feature tensor is passed through the initial fully connected layer
        x = F.leaky_relu(self.fc0(x))

        # node_feature and edge_index tensors are passed through the 5 convolutional layers
        x = F.leaky_relu(self.conv1(x, edge_index))
        x = F.leaky_relu(self.conv2(x, edge_index))
        x = F.leaky_relu(self.conv3(x, edge_index))
        x = F.leaky_relu(self.conv4(x, edge_index))
        x = F.leaky_relu(self.conv5(x, edge_index))

        # The output of the convolutions is passed through two fully connected layers
        x = F.leaky_relu(self.fc1(x))
        x = self.fc2(x)

        # The output of the network is returned
        return x
```

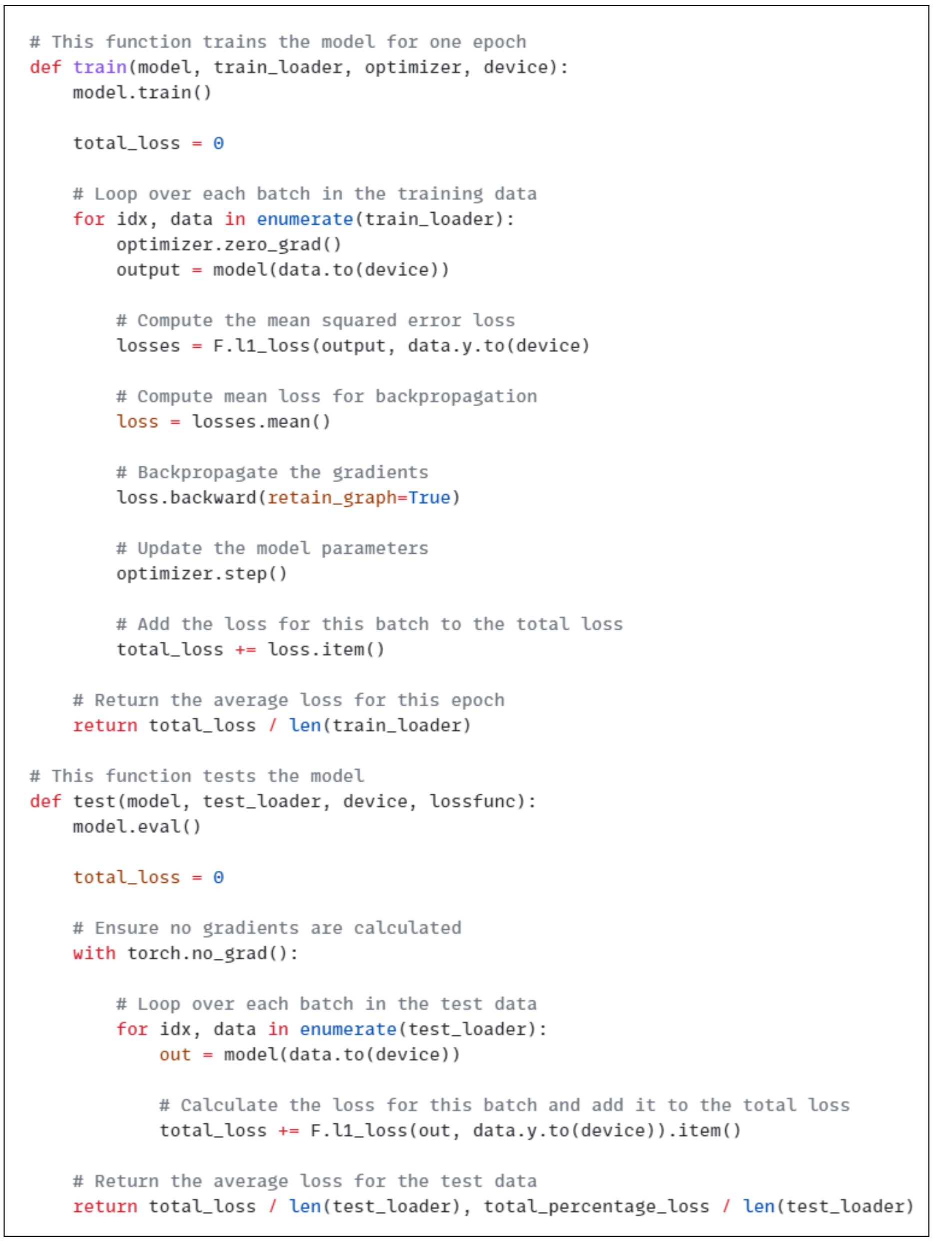
Procedure 5.8: TrussNN model architecture.

Another change made to the model architecture is the size of the output layer to allow for the 8 parameters to be predicted. This adjustment could have been incorporated as an input to the model, mirroring the implementation of *in_channels*, by introducing an *out_channels* input.

5.4.5 Training and Validation

The training process is controlled the same way as for the ArchNN with the *run* and *print_info* functions shown in Procedure 5.4. A batch size of one, meaning backpropagation is performed for every bridge in the training process, is also used for the TrussNN.

The MSE loss function is replaced with L1 loss in the training and test process. The comparison of the MSE and L1 loss functions will be discussed in Section 5.4.6.



```
# This function trains the model for one epoch
def train(model, train_loader, optimizer, device):
    model.train()

    total_loss = 0

    # Loop over each batch in the training data
    for idx, data in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data.to(device))

        # Compute the mean squared error loss
        losses = F.l1_loss(output, data.y.to(device)

        # Compute mean loss for backpropagation
        loss = losses.mean()

        # Backpropagate the gradients
        loss.backward(retain_graph=True)

        # Update the model parameters
        optimizer.step()

        # Add the loss for this batch to the total loss
        total_loss += loss.item()

    # Return the average loss for this epoch
    return total_loss / len(train_loader)

# This function tests the model
def test(model, test_loader, device, lossfunc):
    model.eval()

    total_loss = 0

    # Ensure no gradients are calculated
    with torch.no_grad():

        # Loop over each batch in the test data
        for idx, data in enumerate(test_loader):
            out = model(data.to(device))

            # Calculate the loss for this batch and add it to the total loss
            total_loss += F.l1_loss(out, data.y.to(device)).item()

    # Return the average loss for the test data
    return total_loss / len(test_loader), total_percentage_loss / len(test_loader)
```

Procedure 5.9: TrussNN's training and testing functions.

5.4.6 Designing the Neural Network Architecture

The TrussNN model is tasked with solving a more intricate problem compared to the ArchNN model. As the complexity of the problem increases, it often necessitates the model to have more learnable parameters. Integrating additional output parameters in the fifth convolutional layer, as illustrated in Procedure 5.8, brings the total number of parameters in the model to 1,435,184. Conversely, if the fifth convolutional layer were designed with 128 output neurons - as in the case of the ArchNN - the total parameter count would stand at 623,792.

A test and training loss plot for the model presented in Procedure 5.8 is presented in Figure 5.18.

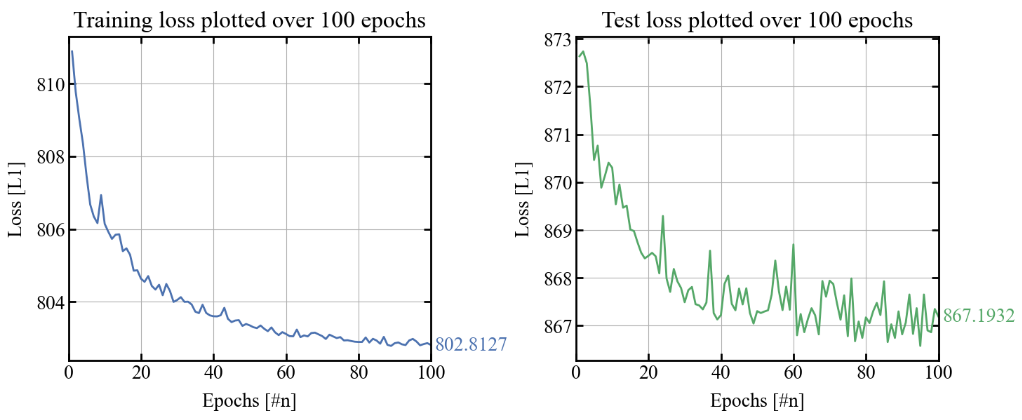


Figure 5.18: Training loss and test loss for the TrussNN using the model architecture shown in Procedure 5.8.

As can be observed from the plots above, it's evident that the TrussNN exhibits higher training and test loss values compared to the ArchNN. In the following sections, different measures will be performed to test the performance of the model.

Determining the Architecture's Activation Function

To check if changing the activation function to ReLU could help the model's learning, the two activation functions ReLU and LeakyReLU are, also for the TrussNN model, plotted against each other. Because TrussNN's test loss is significantly higher than its training loss, the training and test plots are divided into two plots to better catch the models' learning patterns.

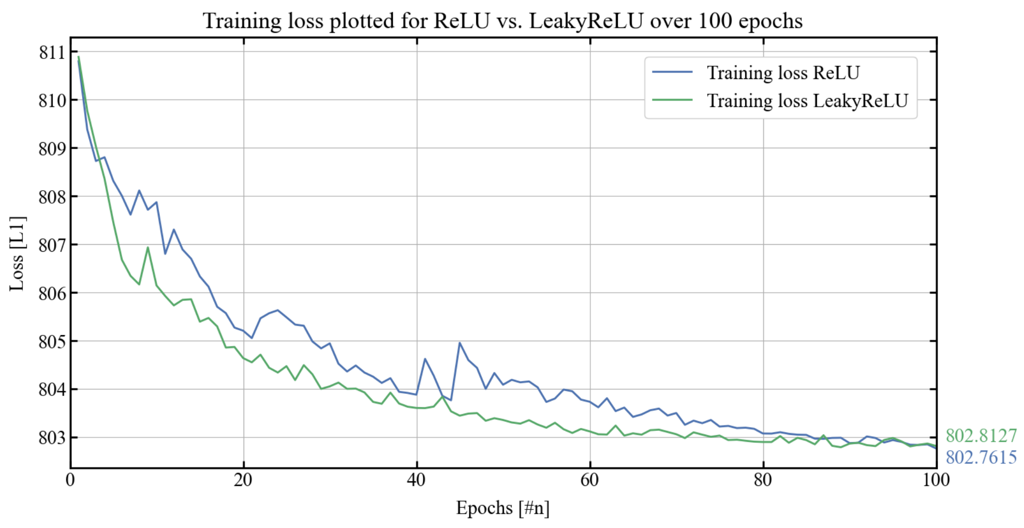


Figure 5.19: Comparison plot: training loss for models trained with ReLU vs. LeakyReLU.

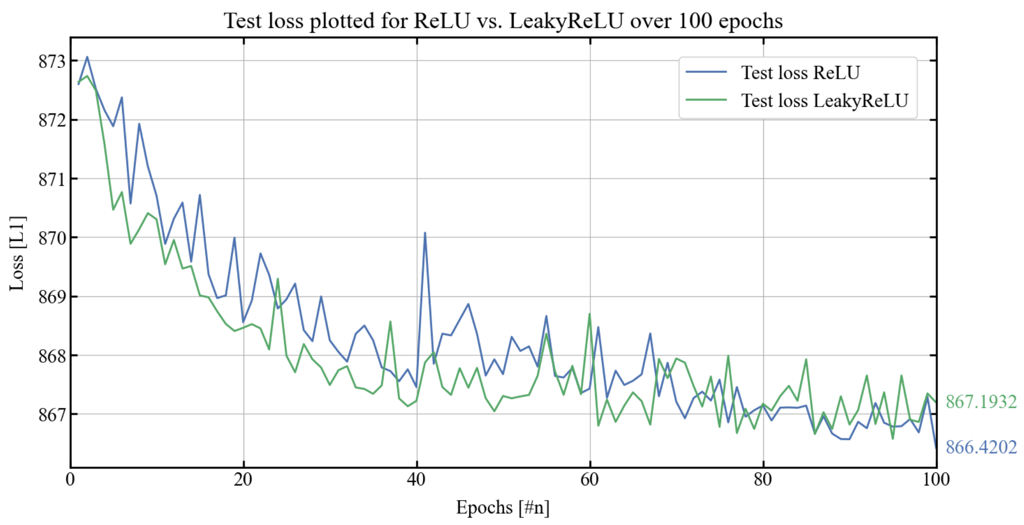


Figure 5.20: Comparison plot: L1 test loss for models trained with ReLU vs. LeakyReLU.

As demonstrated in the plots shown in Figure 5.19 and 5.20, the ReLU-trained model and the LeakyReLU-trained model have significantly similar loss values, with LeakyReLU giving a slightly lower consistent value. Because of the oscillations, which are also apparent for the TrussNN, the ReLU-trained model randomly ends up having a lower test loss value than the LeakyReLU-trained model, 866.42 against 867.19.

Figure 5.21 showcases the predicted functions for Nodes 2, 10, and 24 of the trained model using LeakyReLU and ReLU activation functions. The specific truss bridge (bridge_id = 84) under consideration has a length of 25.56 meters and a height of 3.5 meters. It is randomly selected from the dataset for analysis.

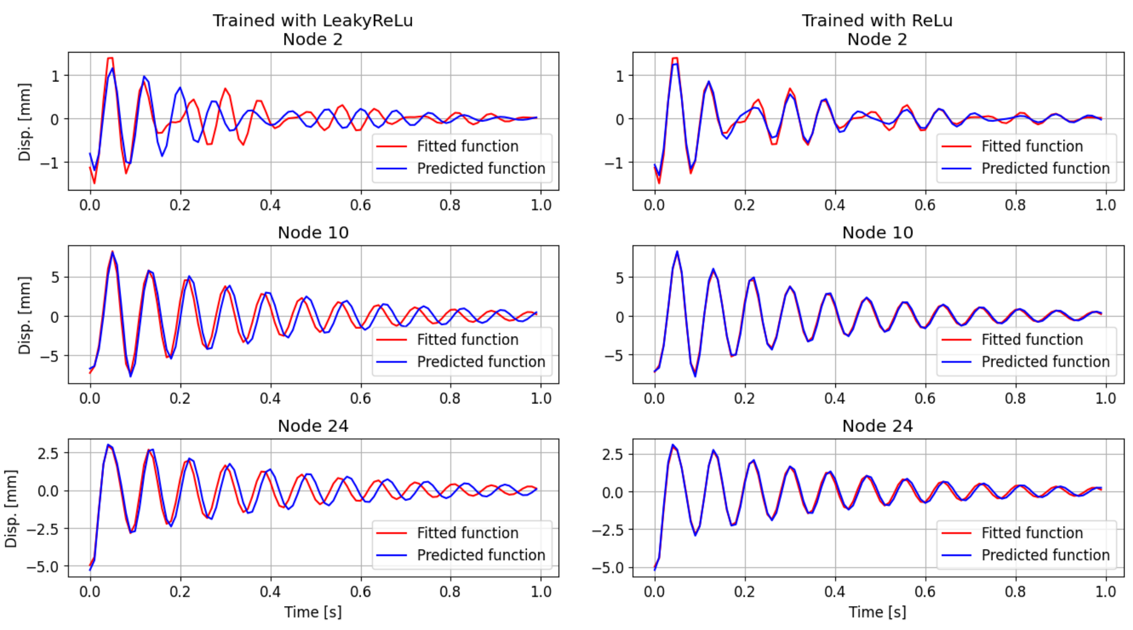


Figure 5.21: Comparison plot: Prediction with LeakyRelU and ReLu.

As depicted in Figure 5.21, the ReLU-trained model performs significantly better than the LeakyRelu-trained model on the test data. This comparison emphasizes the risk associated with relying on the final epoch model to deliver a satisfactory loss value. Given the overall better performance demonstrated by LeakyReLU in the plots from Figures 5.19 and 5.20, LeakyReLU remains the activation function of choice. To ensure a well-performing model, early saving of models with low test loss values is implemented for the final model.

Determining the Architecture's Loss function

As discussed in Section 5.4.2, the node feature matrix incorporates details about whether nodes are fixed or not. This was introduced with the intention of ensuring zero displacements for the fixed nodes. However, because the boundary conditions are identical for all the training examples, this feature is now removed from the matrix to investigate its impact on the test loss.

While including boundary condition information in the feature matrix could prove beneficial in cases where these conditions vary, in the current scenario where all training and testing examples share the same fixed nodes, this feature has been omitted to determine whether it can lead to better loss values.

In Figure 5.22 the training and test performance of two models, one trained using MSE loss and one using L1, are compared. It can be observed that the test loss significantly improves when removing the constraint feature from the feature matrix. The main difference in behaviour for the two functions is that for MSE the test loss lies over the training loss, but for L1 it's the opposite as test loss is under train loss as depicted in Figure 5.22.

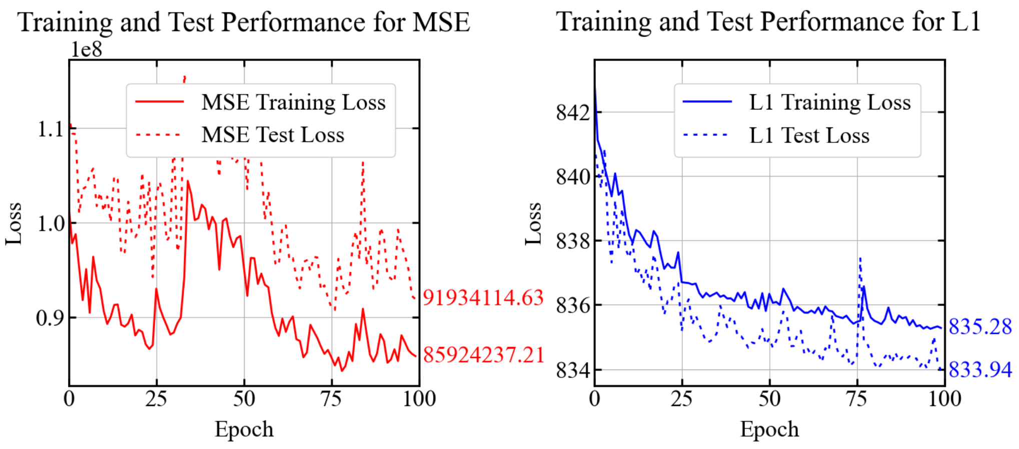


Figure 5.22: Comparison plots: Mean squared error loss vs L1 loss.

As mentioned previously, several factors influence the behaviour and relationship between the train and test loss. The L1 loss behaves as anticipated, showcasing a network that learns in a stable manner. However, this is not the case for the MSE loss. The MSE loss also exhibits a test loss that is approximately 10% higher than the training loss, creating an offset between the two.

By plotting the prediction for Node 2, Node 10, and Node 24, it is clear that L1 outperforms MSE. Figure 5.23 displays the plot for the three nodes, Node 2, 10 and 24, side by side. The truss bridge is the same as the one used in Figure 5.21, (bridge_id = 84).

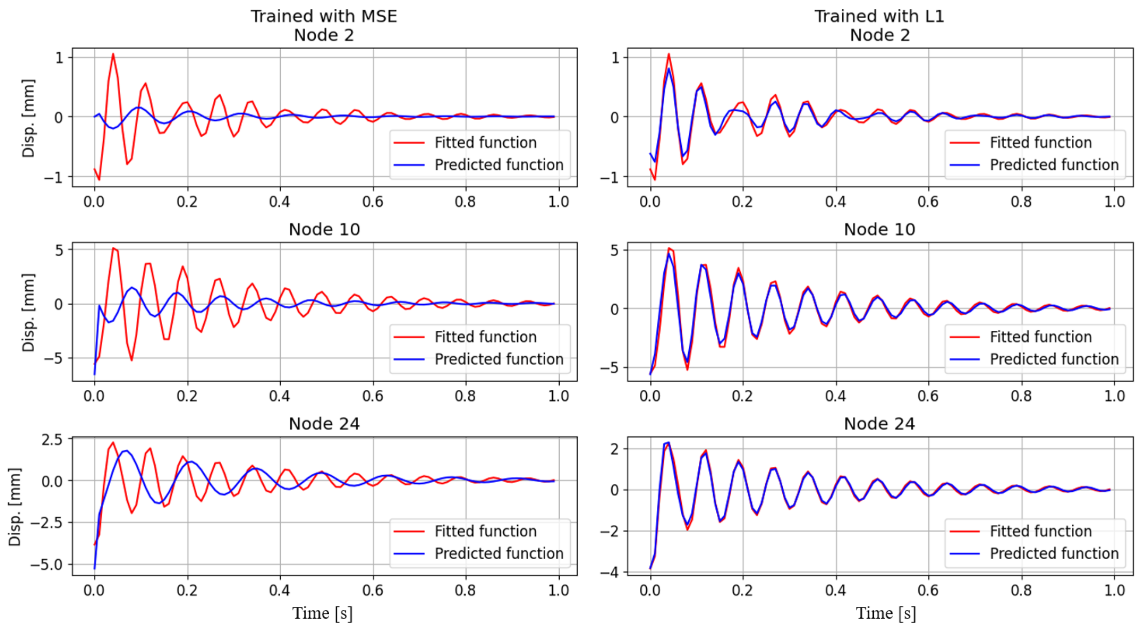


Figure 5.23: Comparison plots: Prediction with Mean squared error loss function vs L1 test loss.

Since the L1-trained model is showing a significantly better learning trend than the MSE-trained model, shown in Figure 5.22, and shows good predictions for a random test example, as shown in Figure 5.23, the L1 loss function is the preferred choice for TrussNN. In further training enhancement, the constraint feature of the feature matrix is removed. Including this feature might have distracted the model by introducing unnecessary noise. Fewer features in the model also help with computational efficiency.

Enhancing Training: Increasing Epoch Count and Implementing Early Stopping

Due to limited computational resources, the range of testing and optimization efforts for TrussNN was restrained. However, focused tests are conducted in the next section to unravel the reasons behind the high loss values and to discern when the model predicts accurately versus when it falters. The model subjected to these tests was obtained by implementing an early saving technique based on a preset condition during an extended epoch count. The epoch count was set to 1000 epochs and the model is saved the first time when falling below a test loss of 835. After this, the model is overwritten every time the model's test performance improves.

Figure 5.24 showcases the training process of the model designed on the decisions made in the preceding sections. For the conservation of computational resources, the TrussNN model architecture was modified to reduce its complexity. It was found that incorporating 512 output neurons in the fifth convolutional layer, as previously discussed in 5.4.4, didn't substantially improve the model's performance. The model was therefore changed to the architecture L16-C32-C64-C128-C256-C128-L64-L8. This revelation can be observed when comparing the test performance plot for the L1 loss in Figure 5.22 with the first 100 epochs of the training process portrayed in Figure 5.24.

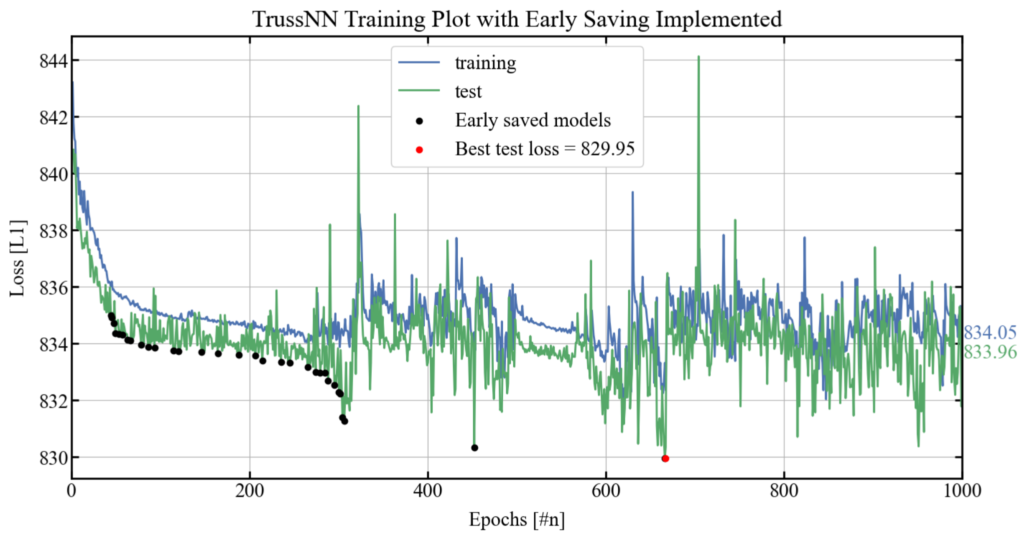


Figure 5.24: Retrieving the best performing model: Introducing early saving to the TrussNN training.

The highest-performing model in the training process, which is showcased in Figure 5.24, achieved an L1 loss of 829.95.

5.5 TrussNN Results

The best and worst three prediction outcomes, with regard to loss values, are analysed to delve deeper into the model's prediction precision and discern potential trends or patterns related to the model's performance across varying bridge lengths and heights. These results are documented in Table 5.4, where each loss is paired with the corresponding bridge length and height from the test set. This comparison seeks to identify any possible correlations between the model's prediction accuracy and the physical attributes (length and height) of the truss bridges.

Table 5.4: The TrussNN's three best and three worst predictions within the test dataset and the bridge's corresponding lengths and heights.

The Three Best TrussNN Predictions		
L1 loss	Bridge Length	Bridge Height
0.569	25.7 m	3.5 m
0.571	25.6 m	3.5 m
0.595	25.3 m	3.6 m
The Three Worst TrussNN Predictions		
L1 loss	Bridge Length	Bridge Height
11441.26	21.1 m	3.5 m
6668.16	20.9 m	3.7 m
5495.13	21.2 m	4.3 m

To investigate the time-history features of bridges for which the model provides low-loss predictions, and contrast them with those where the model yields high-loss predictions, the time-history predictions for the models that gave the lowest and highest test loss values are plotted for a selection of nodes in Figure 5.26. The truss bridge with node numbering is shown in Figure 5.25.

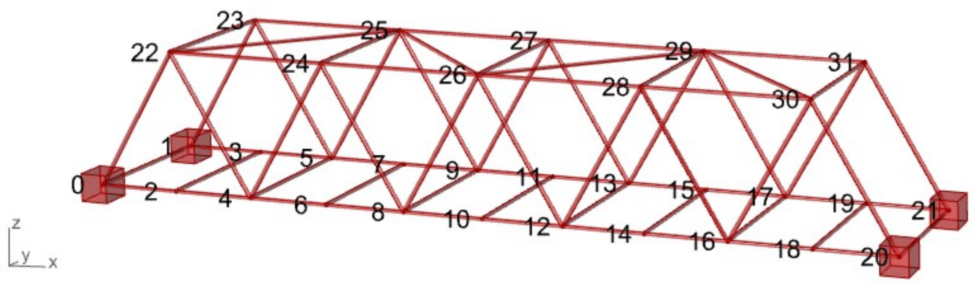


Figure 5.25: TrussNN bridge with node numbering.

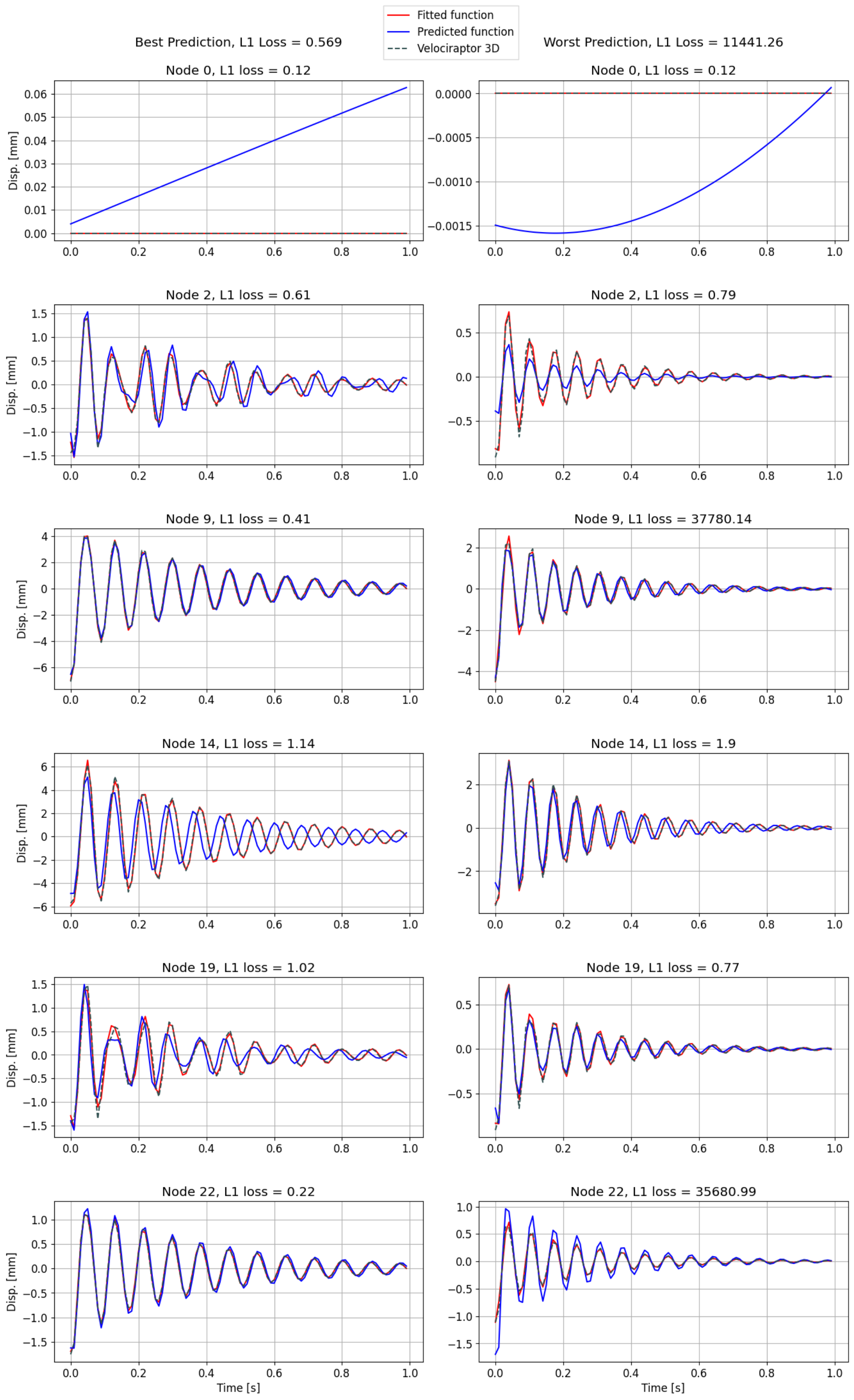


Figure 5.26: TrussNN predictions for lowest loss value and highest loss value plotted for a selection of nodes against the fitted damped sine wave and the Velociraptor3D displacement time histories.

The truss bridge response to the lowest loss and highest loss predictions can be seen in these videos: Truss Bridge Response to Lowest Loss Prediction, Truss Bridge Response to Highest Prediction

The labels and predictions for the lowest loss and highest loss predictions are shown in Table 5.5.

Table 5.5: Node 9 labels and parameters for the lowest loss prediction and the highest loss prediction.

	Lowest loss bridge		Highest loss bridge	
Parameter	Labels	Predictions	Labels	Predictions
A_1	5.20	4.86	3.07	2.75
α_1	75.03	74.52	93.43	95.13
ω_1	- 2.01	- 2.01	- 2.05	-2.22
ϕ_1	2.70	2.51	4.28	4.13
A_2	2.39	2.41	18.79	2.42
α_2	123.71	122.06	- 25029.55	- 154.77
ω_2	- 1.86	- 2.04	-0.095	- 1.077
ϕ_2	30.91	30.52	277382.69	36.024

These results will be discussed in Section 5.6.

5.5.1 Results for Truss Bridge with Different Geometry

The TrussNN is now tested on a geometry that differs from the geometry the model is trained on. The geometry tested for is Bridge nr. 3 in Figure 4.9 from Section 4.3.3, but the load is applied to the two middle nodes, like for the geometries TrussNN is trained on. The bridge is 10 m at its tallest and 3 m at its shortest and is referenced to as *Rising Truss Bridge*. Rising Truss Bridge is shown with node numbering in Figure 5.27.

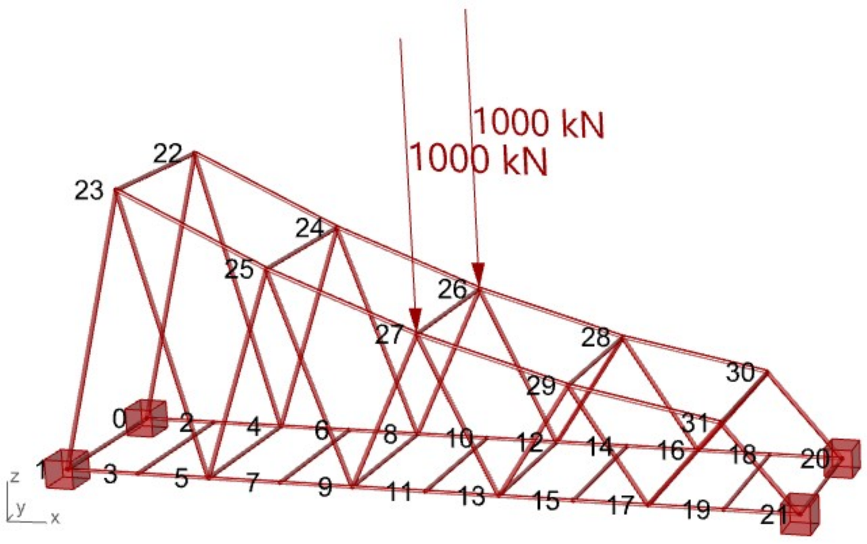


Figure 5.27: Truss bridge with geometry different from the one TrussNN is trained on.

The time history plots for a selection of nodes are shown in Figure 5.28 for Rising Truss Bridge. The figure shows the TrussNN prediction, the fitted damped sine wave, and the Velociraptor3D displacement time history.

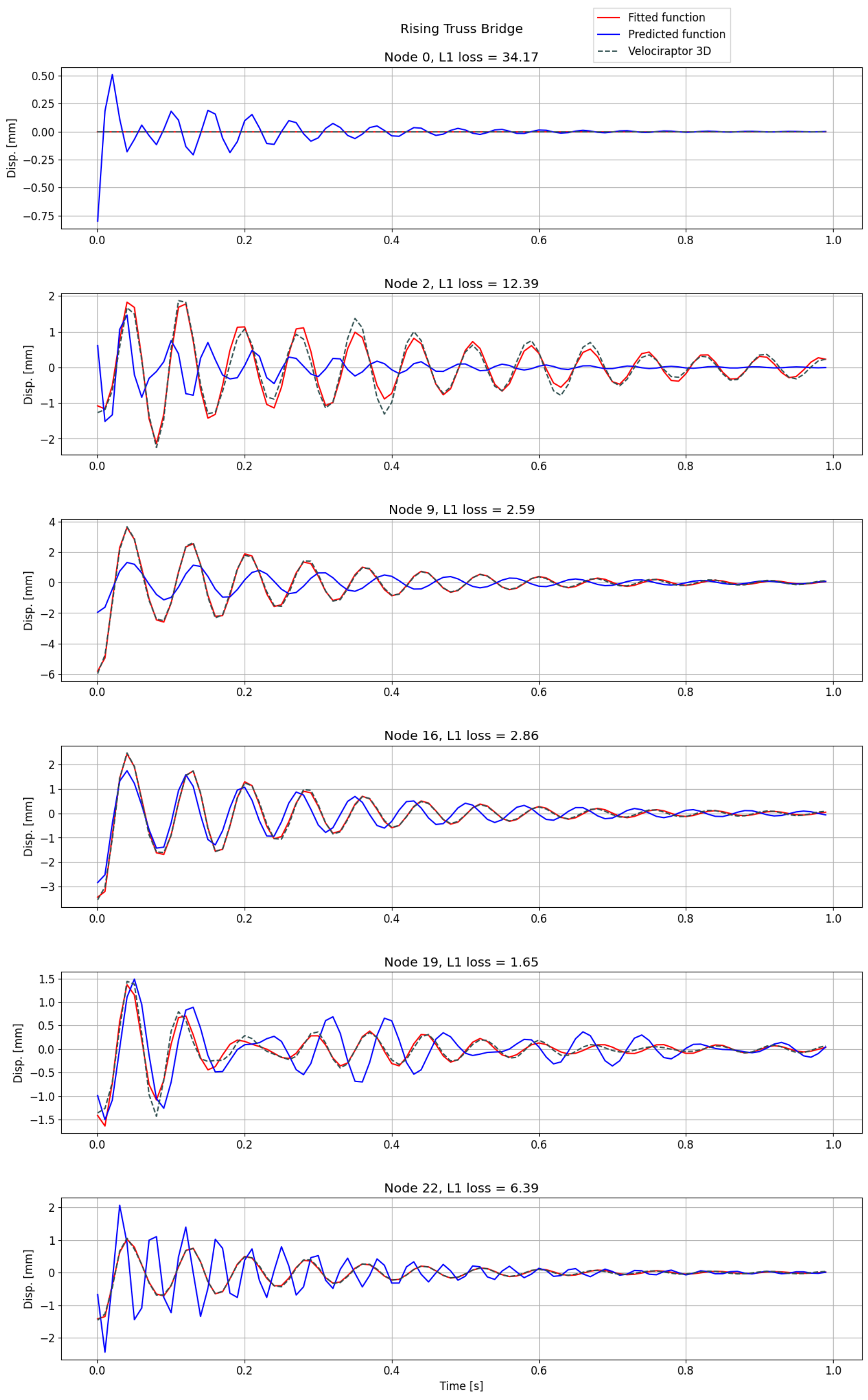


Figure 5.28: Time history plots for the bridge geometry shown in 5.27 for a selection of nodes.

5.5.2 Small Note on Time Usage

As mentioned earlier, one of the notable advantages of the TrussNN model is its potential for significant time savings in predicting the coefficients. In comparison to V3D, which takes approximately 500 milliseconds to calculate the time history displacements for all degrees of freedom in a single Truss Bridge, the TrussNN model only requires around 15.4 milliseconds. It is worth noting the possible argument that the time usage of V3D should be divided by six, to account for its prediction of all DOFs. This gives a time advantage of 500% However, it is not possible for V3D to only calculate one axis as is and therefore the full time usage can directly be compared to the TrussNN's 15 milliseconds. This, in turn, gives a time advantage of 3,300% or 33 times. Please note, that this is a rough estimate and more thorough benchmarks on time usage should be performed.

5.6 TrussNN Discussion

Table 5.4 shows that the three best loss values are all for bridges between 25 and 26 meters long, which could suggest that the model has captured some characteristics or behaviours that are more common or more pronounced in bridges of this length. The worst loss results happen for shorter bridges around 21 meters, which might indicate that TrussNN has a harder time capturing the dynamics of shorter bridges, or that these bridges have certain unique properties that are not as well captured by the model. The heights of the bridges are significantly similar for all the predictions presented in Table 5.4, with the exception of the third worst prediction which has a marginally taller bridge. This implies that while height may play a role, it doesn't necessarily determine the model's prediction accuracy in a straightforward manner.

Figure 5.26 illustrates that the bridge which had the highest loss value exhibits a more uniform displacement time-history shape along its length. The bridge on which the TrussNN performed best, with regard to loss value, has irregularly shaped time-history plots closer to the supports, as shown for nodes 2 and 19. Nodes 9 and 22, however, which lay towards the middle of the bridge have plots that resemble a single damped sine wave, like the highest loss bridge has for all nodes shown. The shorter bridge has a more stable vibration which might produce some problems during the approximation Procedure 5.7, where the labels are generated. Stable vibration gives a well-fitted damped sine wave for the first sine function, f^S, giving very small values for the error sine wave, f^e (5.7), which in turn makes some coefficients very much larger than the others. Table 5.5 displays how two of the coefficient labels are exceptionally large. As discussed in Section 3.2.4, the L1 loss is more stable to outliers, which was observed in Figure 5.23 where the model trained with MSE loss performed much worse than the one trained with L1 loss.

The videos (Truss Bridge Response to Lowest Loss Prediction, Truss Bridge Response to Highest Prediction) showing the truss bridge's response, illustrates that both the lowest loss prediction and highest loss prediction have pretty good overall predictions of the truss bridge's time history. It can be observed by comparing Figure 5.25 to the videos, for example for the lowest loss prediction, how well the TrussNN prediction follows the Velociraptor3D calculation for node 9, and how for node 12 they have increasingly opposite oscillation. These observations can also be made in Figure 5.26.

Table 5.5 shows how the highest loss bridge had good predictions for the 4 first parameters, producing an accurate prediction plot, as illustrated in Figure 5.26. This demonstrates how high loss values don't necessarily correlate with bad predictions for the TrussNN.

The ArchNN model architecture has proven to perform well for a different structural engineering problem, although the losses were much higher for the TrussNN. For data outside the trained field, represented by the Rising Truss Bridge, the performance unsurprisingly was not sufficient, although there were a few nodes that correlated some on the shorter side of the bridge. Once again, more training data is needed to handle differing geometries.

6 Discussion

This chapter explores and discusses various aspects of the thesis, focusing on the finite element analysis part and the machine learning part. Each part will be examined independently, as well as in combination, as the framework it creates together. Furthermore, we will analyze their applicability for real-time digital health monitoring and examine the potential and possibilities offered by the developed framework. Alongside the advantages, the associated problems or challenges that arise in the context of this framework will also be addressed.

6.1 VelociraptorFEA

In the quest for efficiency and improved performance, the integration of digital software has become absolute in an engineer's design process. To achieve intuitive and well-performing solutions, the utilization of advanced tools is key. In this context, the question of if it is possible to create a lightweight and flexible, but still powerful static and dynamic FEA plug-in for Grasshopper arises.

To effectively develop advanced programs, it is often beneficial to adopt a bottom-to-top approach. By starting with a strong understanding and a solid foundation for smaller problems, it becomes possible to incrementally build complexity. This approach outlines the design process of both V2D and V3D. Initially, the development focused on creating the static solver for V2D, ensuring its robustness and functionality. Subsequently, the expansion into dynamic problems within the 2D was done. After successfully establishing a reliable framework with V2D, the natural progression was made towards extending the capabilities to 3D. Once again, the process involved developing the static solver followed by the dynamic solver.

In the benchmark evaluations, in section 4.3, specifically Benchmarks #1 and #2, Velociraptor demonstrated exceptional performance when compared to Karamba3D, both in 2D and 3D analyses. Subsequently, Benchmark #3 involved the analysis of multiple truss bridges, with a particular focus on one specific structure. Once again, Velociraptor delivered reliable results for displacement, while some discrepancies in beam forces were observed. These discrepancies can likely be attributed to implementations of Thimoschenko beam elements in Karamba3D and Euler-Bernoulli beam elements in Velociraptor. Moving on to more complex structures, a diverse range of gridshells was examined, and V3D showcased good performance in these analyses. It's worth mentioning that for the largest gridshell construction, the script became somewhat slow and not as instantaneous as wanted. Lastly, the dynamic capability was assessed by Benchmark #5, where both correct eigenvalues and time history plots are proven. This shows the "lightweight" Velociraptor encompasses accurate dynamics, by time integration and solving the eigenvalue problem.

The advancements in computational power and progress in finite element analysis (FEA) have led to increasingly powerful FEA software. However, this enhanced capability often comes at a price: a more complex user interface and higher demands for the user's knowledge of the specific program. While this development is not entirely negative, it can limit the freedom to experiment and explore different design and structural possibilities. A more trial-and-error approach, where the instant consequences of parameter changes are observable, enhances the designing intuition. This improved intuition, in turn, enables structural engineers to have a deeper understanding of structural behaviour on the spot, without necessarily relying solely on software analysis. This shift, as a synergy effect, empowers engineers to make intuitive-informed decisions, better decisions. In today's age, where over-reliance on software can be a potential drawback, this development holds significant advantages. The FEA properties, both static and dynamic, that the Velociraptor framework offers, provides an easy, fast and flexible design process. V3D provides analysis of beam forces, however, it should be extended to also provide analysis for strains and stresses. It was not prioritized due to it not having direct relevance for further machine learning and naturally the limitation of time.

Overall the performance of the Velociraptor proved successful, but one drawback of computational design using AAD and FEA, in general, is slower scripts when the structures become very large. This becomes a motivation for why the field of neural networks is explored next.

6.2 Neural Networks

The machine learning part of this thesis, consisting of two different neural networks, proved interesting and had good results. Which in turn, helped to prove how graph neural networks may be used more in the coming future within the field of structural analysis.

The ArchNN model demonstrated impressive performance by predicting nodal moments with an error rate of 2.7%. This level of accuracy is often sufficient, at least for preliminary analysis purposes. However, it is important to note that the model used in this evaluation was relatively straightforward, considering its constant cross section and load conditions. By incorporating additional input features such as varying load scenarios and cross sections, the network could be enhanced to achieve greater flexibility. Accommodating these changes may require architectural adjustments, such as incorporating more layers or neurons to handle the increased complexity. Despite this, there is no obvious difficulty in implementing such elaborations. Nevertheless, it would necessitate a larger dataset and a more time-consuming training process.

TrussNN, with a more complex prediction task, gave favourable results and demonstrated the versatility of the network architecture used in ArchNN. Compared to V3D, TrussNN offers a clearer and more streamlined approach by bypassing the iterative Newmark method, thereby saving valuable time. During the training phase, the model had difficulties achieving low loss values and decreasing loss values after a certain amount of epochs. Nonetheless, the overall results were found to be satisfactory after examining the time history plots that the predictions produced. It is worth considering the exploration of different loss functions, as they may have contributed to further improvements. The way the original fitting of the data was done proved somewhat unfortunate since some coefficients of the approximated functions became unnecessarily exceptionally high. It is thought to be the main reason for the training and test loss behaviour. Therefore different approximation functions and methods here should be examined. Additionally, expanding the capabilities of TrussNN to handle arbitrary loading scenarios and larger variations in geometry is certainly within reach, with more data and computational power.

Both ArchNN and TrussNN require further development for inclusion in a structural analysis framework. It is worth noting that the training process for some networks took over three days, highlighting the importance of additional computational power for a more efficient design process with complex networks. The computers used for this thesis did not have CUDA-capable GPUs, allowing for parallel computing which could have significantly accelerated the training process [44]. Increasing computational resources is crucial for exploring more advanced models.

6.3 Velociraptor Framework

In this section, the comprehensive framework offered by Velociraptor will be discussed. VelociraptorFEA was designed with machine learning in mind, making the generation of training data relatively straightforward. The required data is written to files using VelociraptorFEA components and post-processed in Python within VS Code to generate PyTorch Data objects. However, integrating this data generation process directly into the Grasshopper framework would have streamlined the workflow significantly. This could be done by making components that can import Python packages, such as PyTorch, to define, train and load models. By being able to load trained models as well, Grasshopper components could be made for ArchNN and TrussNN, making a complete Velociraptor structural analysis framework with FEA and GNN components.

As experienced when training the ArchNN and TrussNN models, larger datasets improve learning. More complex models would require larger datasets. By integrating dataset generation in Grasshopper, the process could be automated, enabling Grasshopper to generate datasets consisting of millions of different structures. This would be a significant step towards creating a universal network for structural design. However, this being said, defining a dynamic network architecture that has the ability to encompass all possible situations would be necessary.

Expanding on the concept of a *universal network*, integrating Grasshopper with the machine learning part in Python becomes crucial. This would result in a script that is self-trainable, in theory. Grasshopper would generate the structure, Velociraptor would calculate the necessary data and labels, which are then directly fed into the network for training. Implementing such a process requires substantial computational power, which can be achieved through renting supercomputers or dedicating overnight computing over a longer time. This process can run for several days, weeks or months continually improving as it explores various structural scenarios.

Lastly, the idea of bypassing the finite element method with machine learning should be explored, with questions if it is feasible or even practical. The transition to a machine learning approach for structural analysis would for sure be a paradigm shift, as FEM has stood its ground from its creation. For this to happen, NNs would have to show clear benefits and be proven to boost efficiency. Therefore speed is key, it must be able to outperform classical approaches on time and still have sufficient accuracy. Further computational benchmarks that compare efficiency and speed should be performed in a more thorough manner, which is something that was not prioritized in this thesis. However, in the simple check that was performed in Section 5.5.2, TrussNN shows a speed 33 times greater than that of the SolverDynamic3D component of V3D, showing great promise for further studies.

Safety is undoubtedly a paramount factor in structural design. The foundation of the Finite Element Method (FEM) is well-established and provides a transparent process, setting it apart from the neural network approach. Neural networks, especially in larger and more complex networks, can often be perceived as "black boxes" due to their internal functions and structures being less transparent. This presents a challenge since the structural designer's thorough understanding of structural safety is crucial.

To address this, a combination of machine learning and FEM could establish a productive synergy. The program could leverage machine learning techniques for general analysis while incorporating FEM solvers to conduct verification by running random samples, or by having a *control with FEM-* button. This hybrid approach would provide the benefits of machine learning's efficiency and flexibility while maintaining the robustness and transparency of FEM.

Investing time and resources in machine learning for structural analysis requires courage, but it also holds the potential for significant strategic advantages within the industry. By further developing the Velociraptor framework and incorporating machine learning directly, engineers could enhance their design capabilities, improve efficiency, and explore new possibilities in structural engineering.

6.4 Real Time Health Monitoring

Real-time health monitoring through the use of digital twins is a potential application of the Velociraptor framework and graph neural networks in general. The concept involves creating a digital twin of a structure and employing a graph neural network representation to continuously predict its safety and potential for damage.

The process begins with modelling the structure in Grasshopper and then using Velociraptor to convert the model into a graph representation. This representation can be linked to real-time data from the actual building, including external and internal forces and structural responses. Forces, obtained from weather reports and physical measurements, are treated as input parameters, while the corresponding structural responses serve as labels. The forces and responses can be collected using various devices such as strain gauges, weather stations, and accelerometers. By automatically generating data points, the graph representation can be continuously trained on the actual structural behaviour, enabling it to predict future load scenarios.

The question of why this approach is useful arises. Global warming has multiple negative effects on the planet, one effect being more extreme weather conditions. More frequent heavy rainfall, snowfalls, landslides, and stronger winds pose a significant challenge to structures built based on past and even current standards. By employing a digital twin, more realistic simulations can be conducted, helping to ensure the structural integrity of buildings in potential scenarios. This proactive approach allows for the timely identification of risks and the implementation of necessary counteractions. As demonstrated by the ArchNN and TrussNN models, the predictions, although not exact, were reasonably accurate, sometimes even for geometries outside the trained field. This approach is not limited to the effects of global warming but also applies to other natural phenomena such as earthquakes, which can have devastating consequences.

This real-time health monitoring can be particularly relevant for cultural heritage buildings, which are often very old and may have incomplete documentation. Traditional analysis techniques, such as Rayleigh damping described in Section 2.4.2, are an approximation and do not accurately capture the behaviour of these structures. By employing a GNN trained directly on cause-and-effect relationships, the structural health of heritage buildings can be closely monitored, allowing for better preparation and preservation for the future. This strategy is not just confined to heritage buildings, it can also be effectively utilized for monitoring the structural health of various other buildings and infrastructures.

7 Conclusion

Through this thesis, Finite Element Method and Graph Neural Network theory were exploited to develop VelociraptorFEA and VelociraptorAI, with the research goal displayed below in mind.

The goal of this thesis is to explore the possibility of combining FEM and AI in a structural analysis framework within a parametric environment.

In this thesis, it was demonstrated how static and dynamic structural analysis tools can be implemented in Grasshopper using FEM with 2D and 3D Euler-Bernoulli beam elements. The Grasshopper plugins Velociraptor2D (V2D) and Velociraptor3D (V3D) exhibited robust performance in benchmark tests against the FEA software Robot Structural Analysis and the Grasshopper plugin Karamba3D, both for the FEA solver, Eigenvalue problem solver and the Newmark average acceleration method.

To explore the potential of applying GNNs in structural analysis, ArchNN and TrussNN were developed. ArchNN accurately predicts nodal moments for 20-node 2D structural arches subjected to a distributed load, with a precision of under 3% when the lengths and heights of the arches are within a specified range. ArchNN exemplifies the potential of GNNs in structural analysis. TrussNN modifies and applies the ArchNN architecture to predict z-directional time history functions for 32-node truss bridges subjected to two static, centred point loads. The TrussNN clearly showed promising performance when compared to the Velociraptor3D dynamic solver. An improvement in the procedure of fitting the time history displacements given by V3D could improve TrussNN's learning process. Larger datasets, wider ranges, and longer training processes would improve both ArchNN's and TrussNN's predictive accuracy and versatility, which would be necessary before including them in the Velociraptor structural analysis framework. The workflow of this framework, expressed in the research goal, is depicted in 7.1.

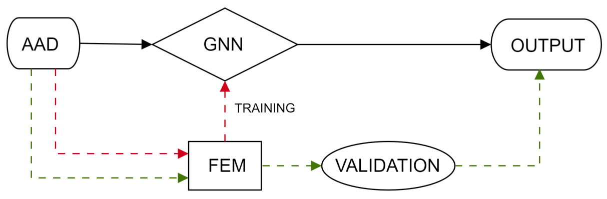


Figure 7.1: Velociraptor structural analysis framework workflow.

This thesis demonstrated how GNNs can be trained and tested using FEA plugins in Grasshopper, and how GNNs can make accurate structural analysis predictions. To implement GNN models as computational tools in VelociraptorAI Grasshopper plugins, dynamic GNN architectures would have to be implemented to handle the variety of input shapes, and mirror the adaptability of FEA. By doing this, developing a structural analysis framework within the parametric environment containing FEA components and GNN components could introduce a new, effective way of computational modelling.

Bibliography

[1] AnalystPrep. *Overfitting and Methods of adressing it*. 2021. URL: https://analystprep.com/study-notes/cfa-level-2/quantitative-method/overfitting-methods-addressing/.

[2] Robert McNeel & Associates. *Rhino 7*. Accessed: 06-06-2023. 2023. URL: https://www.rhino3d.com/.

[3] Autodesk. *Robot Structural Analysis*. Accessed: 06-06-2023. 2023. URL: https://www.autodesk.com/products/robot-structural-analysis/overview?term=1-YEAR&tab=subscription.

[4] Kolbein Bell. *An engineering approach to FINITE ELEMENT ANALYSIS of linear structure mechanics problems*. Fagbokforlaget, 2014. ISBN: 9788232102686.

[5] Jason Browniee. 2021. URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/.

[6] Vitaly Bushaev. *How do we 'train' neural networks ?* 2017. URL: https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73.

[7] Anil K. Chopra. *Dynamics of Structures*. Pearson, 2017. ISBN: 9780134555126.

[8] The Scipy community. *Optimization and root finding (scipy.optimize)*. Accessed: 06-06-2023. URL: https://docs.scipy.org/doc/scipy/reference/optimize.html.

[9] Robert D. Cook, David S. Malkus and Michael E. Plesha. *Concepts and Applications of Finite Element Analysis*. John Wiley & Sons, Inc., 1989. ISBN: 0471847887.

[10] Stephanie Glen. *Damped Sine Wave: Definition, Example, Formula*. Accessed: 06-06-2023. URL: https://www.statisticshowto.com/calculus-definitions/damped-sine-wave/.

[11] Edem Gold. *The History of Artificial Intelligence from the 1950s to Today*. 2023. URL: https://www.freecodecamp.org/news/the-history-of-ai/#the-ai-winter-of-the-1980s.

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

[13] IBM. *What is machine learning?* Acessed: 06-06-2023. URL: https://www.ibm.com/topics/machine-learning.

[14] Karamba3D. Accessed: 06-06-2023. 2023. URL: https://karamba3d.com/.

[15] Ayoosh Kathuria. 'Intro to optimization in deep learning: Gradient Descent'. In: (2018). URL: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/.

[16] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. eprint: 1412.6980.

[17] Arnd Koeppe. *Deep Learning in the Finite Element Method*. Tech. rep. Report No.: IAM-11. Aachen, Germany: RWTH Aachen University, 2021. URL: https://publications.rwth-aachen.de/record/819355/files/819355.pdf.

[18] Kenneth Leung. *The Dying ReLU Problem, Clearly Explained*. 2021. URL: https://towardsdatascience.com/the-dying-relu-problem-clearly-explained-42d0c54e0d24.

[19] Yuxi Li. *Deep Reinforcement Learning: An Overview*. 2018. arXiv: 1701.07274. URL: https://arxiv.org/abs/1701.07274.

[20] Kjell Magne Mathisen. *Lecture 1: The Finite Element Method*. 2021.

[21] Microsoft. *Visual Studio 2022*. Accessed: 06-06-2023. 2023.

[22] Microsoft. *Visual Studio Code*. Accessed: 06-06-2023. 2023.

[23] Ravindra Parmar. *Common Loss functions in machine learning*. 2018. URL: https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23.

[24] The-Crankshaft Publishing. *FEM for Frames (Finite Element Method) Part 1*. Accessed: 06-06-2023. URL: http://what-when-how.com/the-finite-element-method/fem-for-frames-finite-element-method-part-1/.

[25] PyTorch. *Linear*. Accessed: 06-06-2023. 2023. URL: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html.

[26] PyTorch. *PyTorch*. Accessed: 06-06-2023. URL: https://pytorch.org/.

[27] PyTorch. *torch.nn*. Accessed: 06-06-2023. 2023. URL: https://pytorch.org/docs/stable/nn.html#loss-functions.

[28] PyTorch. *Training With PyTorch*. Accessed: 06-06-2023. 2023. URL: https://pytorch.org/tutorials/beginner/introyt/trainingyt.html.

[29] Pranoy Radhakrishnan. *What are Hyperparameters? and How to tune the Hyperparameters in a Deep Neural Network?* 2017. URL: https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a.

[30] Lovely Sabat and Chinmay Kumar Kundu. 'History of Finite Element Method: A Review'. In: *Recent Developments in Sustainable Infrastructure*. Ed. by Bibhuti Bhusan Das et al. Singapore: Springer Singapore, 2021, pp. 395–404. ISBN: 978-981-15-4577-1.

[31] Benjamin Sanches-Lengeling et al. 'A Gentle Introduction to Graph Neural Networks'. In: *Distill* (2021). DOI: 10.23915/distill.00033. URL: https://distill.pub/2021/gnn-intro.

[32] Finn Harald Sandberg. 'Sleipner-havariet og Draugen'. In: (2018). URL: https://draugen.industriminne.no/nb/2018/05/14/sleipner-havariet-og-draugen/.

[33] Franco Scarselli et al. 'The graph neural network model'. In: (2009). URL: https://ro.uow.edu.au/cgi/viewcontent.cgi?article=10501&context=infopapers.

[34] SAGAR SHARMA. *Activation Functions in Neural Networks*. 2017. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[35] Siddharth Sharma, Simone Sharma and Anidhya Athaiya. 'ACTIVATION FUNCTIONS IN NEURAL NETWORKS'. In: *International Journal of Engineering Applied Sciences and Technology, 2020* 4.12 (2020), pp. 310–316.

[36] Standford. *Convolutional neural networks (cnns / convnets).* [online course, stanford University]. 2022. URL: https://cs231n.github.io/convolutional-networks/.

[37] PyG Team. *PyG Documentation.* Accessed: 06-06-2023. 2023. URL: https://pytorch-geometric.readthedocs.io/en/latest/.

[38] PyG Team. *Source code for torch$_g$eometric.nn.conv.feast$_c$onv.* Accessed: 06-06-2023. 2023. URL: https://pytorch-geometric.readthedocs.io/en/latest/_modules/torch_geometric/nn/conv/feast_conv.html.

[39] TensorFlow. *Introduction to Tensors.* Accessed: 06-06-2023. URL: https://www.tensorflow.org/guide/tensor.

[40] TIBCO. *What is a NEural Network.* Accessed: 06-06-2023. 2023. URL: https://www.tibco.com/reference-center/what-is-a-neural-network.

[41] Thornton Tomasetti. *Colibri (by COREstudio).* Accessed: 06-06-2023. 2022. URL: https://www.food4rhino.com/en/app/colibri.

[42] Jordi Torres. *Learning Process of a Deep Neural Network.* 2020. URL: https://towardsdatascience.com/learning-process-of-a-deep-neural-network-5a9768d7a651.

[43] Turing. *What Is the Necessity of Bias in Neural Networks?* Accessed: 06-06-2023. URL: https://www.turing.com/kb/necessity-of-bias-in-neural-networks.

[44] Turing.com. *Understanding NVIDIA CUDA: The Basics of GPU Parallel Computing.* Accessed: 06-06-2023. 2023. URL: https://www.turing.com/kb/understanding-nvidia-cuda.

[45] Nitika Verma, Edmond Boyer and Jakob Verbeek. *FeaStNet: Feature-Steered Graph Convolutions for 3D Shape Analysis.* 2018. eprint: 1706.05206.

[46] Thomas Lunde Villanger and Kristian Nikolai Åland. *Exploring Finite Element Analysis in a Parametric Environment.* Trondheim, Norway, 2021.

[47] Eamon Whalen and Caitlin Mueller. 'Towards Reusable Surrogate Models: Graph-Based Transfer Learning on Trusses'. In: (2023). URL: https://arxiv.org/pdf/2109.02689.pdf.

[48] Edward L. Wilson. *Static and Dynamic Analysis of Structures.* Computers and Structures, Inc., 2010. ISBN: 9780923907044.

[49] Harsh Yadav. *Dropout in Neural Networks.* 2022. URL: https://towardsdatascience.com/dropout-in-neural-networks-47a162d621d9.

[50] Jie Zhou et al. 'Graph neural networks: A review of methods and applications'. In: *AI Open* 1 (2020), pp. 57–81. ISSN: 2666-6510. DOI: https://doi.org/10.1016/j.aiopen.2021.01.001. URL: https://www.sciencedirect.com/science/article/pii/S2666651021000012.

[51] Victor Zhou. *Machine Learning for Beginners: An Introduction to Neural Networks.* Accessed 06-06-2023. 2022.

Appendix

A GitHub Repositories

Project	URL
Velociraptor2D	https://github.com/hermanliaboe/Velociraptor2D.git
Velociraptor3D	https://github.com/hermanliaboe/Velociraptor3D.git
ArchNN & TrussNN	https://github.com/hermanliaboe/VelociraptorAI.git

B Videos

Name	Description and URL
Beam Time History	Video showing how the 4-meter long beam vibrates subjected to an initial displacement https://youtu.be/_pVqGHr0OOI
Truss Bridge Time History	Video showing how the truss bridge vibrates subjected to an initial displacement https://youtu.be/D3zHz4L68sc
Truss Bridge Prediction BEST	Video showing the comparison between the Velociraptor3Ds calculated model and TrussNN predicted model https://youtu.be/JO1AGABxf-k
Truss Bridge Prediction WORST	Video showing the comparison between the Velociraptor3Ds calculated model and TrussNN predicted model https://youtu.be/fB3I0_bwx-k