FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Prediction of Stress-Strain Curves in Multi-Phase Materials Under Tension Using Machine Learning

Luís Guilherme de Campos Tavares Fernandes



Masters in Mechanical Engineering

Supervisor: João Ferreira

Co-supervisor: Igor Lopes

October 28, 2022

Prediction of Stress-Strain Curves in Multi-Phase Materials Under Tension Using Machine Learning

Luís Guilherme de Campos Tavares Fernandes

Masters in Mechanical Engineering

October 28, 2022

Abstract

In this work, is described the process used in the creation of a framework capable of training, testing and evaluating multiple neural networks with the intent to predict the stress-strain curves of dual-phase policrystiline materials. This encompassed the generation of a new dataset, as well as multiple tests with both conventional and less traditional configurations of layers to build a rough architecture, followed by an extensive automated tuning stage and developing of data augmentation technics.

The dataset contained both representative volume elements and respective curves. The microstructures were generated using tessellation and then discretized into 16×16 meshes. These were then analyzed using coupled multi-scale finite elements software, outputting the homogenized stresses at multiple strain points. The process was optimized and alternatives to tessellation were explored, in an attempt to reduce processing time. Once all curves were computed and compiled, the dataset was reviewed, and some particular cases identified.

In the machine learning front, the benefits of using convolutional neural networks over fully connected architectures were demonstrated. Multiple combinations of layers and respective hyperparameters were tested, resulting in a model with some precision. Further along, this architecture was optimized using tuning software, identifying and solving the bottlenecks in it. There were two tuning stages, separated by the implementation of a mixture of online and offline data augmentation technics, that justified further adjustments to the hyperparameters.

In the end, the final model accuracy was measured and compared to the ones that preceded it during the development of this work. Some samples that proved challenging to the model were also isolated. The final results proved very promising for future implementations of artificial intelligence in computational mechanics, saving considerable amounts of time with little trade off in accuracy.

Keywords: artificial intelligence, computational mechanics, convolutional neural networks, finite element method, machine learning, multiscale analysis, representative volume elements, stress-strain curve

Resumo

Neste trabalho, é descrito o processo utilizado na criação de um *framework* capaz de treinar, testar e avaliar múltiplas redes neurais com a intenção de prever as curvas de tensão-deformação de materiais policristalinos bifásicos. Isto abrange a geração de um novo *dataset*, bem como múltiplos testes de configurações convencionais, como menos tradicionais, para construir uma arquitectura inicial. De seguida, ocorreu uma extensa fase de ajustes automatizados e desenvolvimento de técnicas de *data augmentation*.

O *dataset* continha tanto os elementos representativos do volume como as respectivas curvas. As microestruturas foram geradas usando tesselação e depois discretizadas em malhas de dimensão 16×16. Estas foram então estudadas utilizando *software* de análise multi escala por elementos finitos, resultando nas tensões homogeneizadas em múltiplos pontos de deformação. Numa tentativa de reduzir o tempo de processamento, este foi otimizado e foram exploradas alternativas à tesselação. Uma vez calculadas e compiladas todas as curvas, o conjunto de dados foi revisto, e alguns casos particulares identificados.

Na parte de machine learning, foram demonstrados os benefícios da utilização de redes neurais convolucionais e foram testadas múltiplas combinações de camadas e respectivos hiperparâmetros, resultando num modelo com alguma precisão. De seguida, esta arquitectura foi otimizada utilizando *software* de *tuning*, identificando e resolvendo os pontos fracos desta. Houve duas fases de afinação, separadas pela implementação de uma mistura de técnicas de *data augmentation online* e *offline*, que justificaram mais outra onda de ajustes.

Por fim, a precisão do modelo final foi medida e comparada com os que o precederam durante o desenvolvimento deste trabalho. Algumas amostras que se revelaram problemáticas para o modelo foram também isoladas. Os resultados finais revelaram-se muito promissores para futuras implementações de inteligência artificial em mecânica computacional, poupando quantidades consideráveis de tempo com pouca perda de precisão.

Keywords: análises multiescala, curvas de tensão-deformação, elementos representativos do volume, inteligência artificial, machine learning, mecânica computacional, método dos elementos finitos, redes neuronais convolucionais

Acknowledgements

My gratitude to my supervisors, João Ferreira and Igor Lopes, for the great patience they showed me over the last months, and for being always present. For your support over the first few months when results were scarce as the framework was beginning to take shape, to the last few weeks when this report was being finalized. Without your guidance, this work would never have gotten so far. I should also thank the CM2S research group for the help with setting up the initial data generation and computational resources made available.

I cannot go without thanking my mother and my sisters. To the many friends I've made over the past five years, for putting up to my never ending dad jokes.

Finally, this work is mostly dedicated to my nieces and nephews, for always looking up to me and never letting me down, and to Rafael and Celso, for being my brothers and best influences I've ever had. Without you, I wouldn't be where I am today.

Guilherme Tavares

"A great leap forward often requires taking two steps back."

— Obi-Wan Kenobi

"And sometimes, all it requires is the will to jump."

— Anakin Skywalker

Contents

1 Introduction **1**
- 1.1 Context . . . 2
- 1.2 Objectives . . . 2
- 1.3 Methodology . . . 2

2 Review of Literature **5**
- 2.1 Continuum Mechanics . . . 5
  - 2.1.1 Kinematics of Deformation . . . 6
  - 2.1.2 Deformation Gradient . . . 7
  - 2.1.3 Strain Measures . . . 9
  - 2.1.4 Stress Measures . . . 9
  - 2.1.5 Laws of Thermodynamics on Deformation . . . 11
- 2.2 Finite Element Method . . . 11
  - 2.2.1 Weak Form of the Equilibrium Equation . . . 12
  - 2.2.2 Space Discretization . . . 12
- 2.3 Multi-Scale Modelling Based in Computational Homogenization . . . 14
  - 2.3.1 Representative Volume Element, the Micro Scale Domain . . . 15
  - 2.3.2 Homogenized Deformation Gradient . . . 16
  - 2.3.3 Multi-Scale Kinematics . . . 16
  - 2.3.4 The Hill-Mandel Macro-Homogeneity Condition . . . 17
  - 2.3.5 Homogenized Stress Tensor . . . 18
  - 2.3.6 Micro-Scale Boundary Conditions . . . 19
- 2.4 Machine Learning . . . 20
  - 2.4.1 Artificial Neural Networks and Dense Layers . . . 21
  - 2.4.2 Training . . . 23
  - 2.4.3 Convolutional Neural Networks . . . 26

3 Initial Dataset, generation and considerations **29**
- 3.1 Material Choice . . . 29
- 3.2 RVE and Mesh Generation - Neper . . . 30
  - 3.2.1 Operating Neper . . . 31
- 3.3 FE^2 Analysis - Links2 . . . 33
  - 3.3.1 FE^2 Modeling . . . 33
  - 3.3.2 Operating Links2 . . . 35
- 3.4 Automation . . . 35
  - 3.4.1 Running neper_by_phase.sh and Links2 . . . 36
  - 3.4.2 Separating Useful Files . . . 37
  - 3.4.3 Compile All Curves and Microstructures . . . 37

3.5 Analysis . . . 38
3.5.1 Curves Distribution . . . 39
3.5.2 Outliers . . . 41
3.6 Substitutes to Neper . . . 46
3.6.1 Random RVEs . . . 46
3.6.2 Cluster-based, Pseudo-Random Distribution . . . 48

4 Model Selection **51**
4.1 Establishing Input Shape and First Model . . . 51
4.1.1 Simple Fully Connected Model . . . 52
4.1.2 Simple CNN Model . . . 54
4.2 Model Design . . . 56
4.2.1 Increasing Depth . . . 58
4.2.2 Purely Convolutional CNN Model . . . 60
4.2.3 Implementation of Average Pooling . . . 63
4.2.4 Replacing Flatten with Global Average Pooling . . . 67

5 Model Refinement and Data Augmentation **71**
5.1 Model Tuning . . . 71
5.1.1 Implementing Optuna . . . 72
5.1.2 Analysis 1: evaluate model's tendencies . . . 73
5.1.3 Analysis 2: implementing Dropout . . . 75
5.1.4 Analysis 3: early drop rates . . . 77
5.1.5 Analysis 4: testing blocks' repeats vs Dropout . . . 79
5.1.6 Analysis 5: final Dropout rates . . . 81
5.1.7 Analysis 6: Learning Rate . . . 83
5.1.8 Tuned Model - Performance and Comparison . . . 85
5.2 New Samples and Data Augmentation . . . 87
5.2.1 Increasing the Dataset . . . 88
5.2.2 Rotating Meshes . . . 88
5.2.3 RVE mirroring . . . 90
5.2.4 Shifting RVEs . . . 90
5.2.5 Model Performance with Data Augmentation . . . 91
5.3 Model Expansion . . . 94
5.3.1 Study 1 - Tweaking Drop Rates . . . 94
5.3.2 Study 2 - Increasing the number of kernels . . . 95
5.3.3 Study 3 - Eliminating Dropout 1 . . . 96
5.3.4 Studies 4 and 5 - Increasing blocks repeats . . . 97
5.3.5 Studies 6 and 7 - Fixing Learning Rate . . . 100
5.4 Final Model Performance . . . 102

6 Conclusions **105**
6.1 Future Work . . . 106

A Bash for RVE Generation **109**

B Bash for Separating Useful Files **111**

C Python for Compiling Dataframe **113**

D Maps for Elements ID in Neper and Python **115**

References **117**

List of Figures

2.1 Rigid Translation and Rotation. 7
2.2 Motion φ on the body $\mathscr{B}$. 8
2.3 Space Discretization of the domain Ω. 13
2.4 Periodic microstructure and two possible RVEs. 15
2.5 Three concurrent dense layers. 21
2.6 Schematic of an artificial neuron. 22
2.7 Plot of Sigmoid, ReLu, Leaky ReLu and Linear activation functions. 24
2.8 Plot of MSE, MAE and Huber with direct comparison. 25
2.9 Schematic of a convolutional layer. 27
2.10 Max Pooling of a 3×3 2D Matrix, with a 2×2 filter and stride 1. 28

3.1 2D RVE generated by Neper included in the Dataset, and respective mesh. 31
3.2 FE^2 model. 34
3.3 Schematic for Dataset Assembly. 36
3.4 Plot of all 100 000 curves in the initial dataset. 39
3.5 Area under the curve distribution. 40
3.6 Median curve and the interval at both 10%, 5%, 2.5% and 1%. 41
3.7 RVEs with horizontal fiber-like formations. 42
3.8 RVEs with vertical beam-like formations. 43
3.9 Rigid Beam perpendicular to the loading direction in a Soft Matrix. 44
3.10 Crushed Elements, highlighted. 44
3.11 RVEs with crushed elements. 45
3.12 Randomly generated RVEs. 47
3.13 Plot of 5 000 curves from randomly generated RVEs. 47
3.14 Comparing Neper and randomly generated RVEs. 48
3.15 First stage in pseudo-random RVE generation. 49
3.16 Last stage in pseudo-random RVE generation. 49
3.17 Multiple RVEs from the same 4×4. 49

4.1 Mesh as a matrix and respective sequence in Links2 and used in the Python Code. 52
4.2 Loaded microstructure and respective Mesh and Python matrix representation. . . 53
4.3 Schematic for K-Fold Cross Validation. 53
4.4 Losses across Folds for FullCon and SimpleCNN. 57
4.5 Illustration of a Stratified K-Fold Cross-Validation for 3 Folds. 58
4.6 Losses across Folds for DeeperCNN. 60
4.7 Relative Histogram of the MAPE for the Test of DeeperCNN. 61
4.8 Huber losses across Folds for PureCNN. 62
4.9 Relative Histogram of the MAPE for the Test of PureCNN. 63

4.10 Representation of both pooling strategies. 64
4.11 Comparison of loss of information between Max Pooling and Average Pooling. . 65
4.12 Huber losses across Folds for AvgPoolCNN. 66
4.13 Relative Histogram of the MAPE for the Test of AvgPooling. 67
4.14 Global Average Pooling of three 4×4 feature maps. 68
4.15 Huber losses across Folds for GlobAvgPoolComp. 70

5.1 Schematic of the workings of Optuna's Hyper-tuning. 72
5.2 Huber Loss by Learning Rate during Analysis 6. 84
5.3 Huber losses across Folds for TunedCNN. 86
5.4 Evolution of Parameters and Losses over Model Development. 87
5.5 Example of data augmentation on an image. 87
5.6 Original and Rotated RVEs with respective stress-strain curves. 89
5.7 Mirrored RVEs and respective curve. 90
5.8 Representation of equivalent RVEs on their periodic arrangement and the process to obtain the shifted one. 91
5.9 Changes in loss with data augmentation. 93
5.10 Variation in Huber Loss for Drop Rates in Study 1. 94
5.11 Variation of drop rates and Huber loss over trials on Study 1. 95
5.12 Variation in Huber Loss for Drop Rates in Study 2. 96
5.13 Variation of drop rates and Huber loss over trials on Study 2. 96
5.14 Variation in Huber Loss for Drop Rate 2 in Study 3. 97
5.15 Variation of drop rate 2 and Huber loss over trials on Study 3. 97
5.16 Variation in Huber Loss for Block Repeats in Study 4. 98
5.17 Variation of block repeats and Huber loss over trials on Study 4. 98
5.18 Variation in Huber Loss for Block Repeats in Study 5. 99
5.19 Variation of block repeats and Huber loss over trials on Study 5. 99
5.20 Batch Size variance in Study 5. 100
5.21 Huber Loss per Learning Rate. 101
5.22 Learning Rate variance in Study 7. 101
5.23 Huber losses across Folds for FinalModel. 102
5.24 Evolution of Parameters and Losses up to Final Model. 103
5.25 Relation between model's prediction MAPE and area under the curve. 103

6.1 Schematic for a possible non-sequential neural network. 107

D.1 Mapping of Element ID in Neper/Link2 and Python Code. 115

List of Tables

3.1 Results of Parallelization Test. 37
3.2 "Mean Absolute Percentage Displacement from Median" for the initial Dataset. . 40

4.1 K-Fold Cross Validation Results for FullCon. 54
4.2 Losses and Percentages of Test RVEs with MAPE bellow 10% and 5% for SimpleCNN. 55
4.3 Comparisson of Losses for FullCon and SimpleCNN. 56
4.4 Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for DeeperCNN. 60
4.5 Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for PureCNN. 62
4.6 Average Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for PureCNN. 63
4.7 Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for AvgPooling. 67
4.8 Average Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for AvgPoolCNN. 68
4.9 Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for GlobAvgPoolCNN. 70
4.10 Average Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for GlobAvgPoolCNN. 70

5.1 Variable hyperparameters in Analysis 1. 73
5.2 Best results for each Study in Analysis 1. 74
5.3 Variable hyperparameters in Analysis 2. 76
5.4 Best results for each Study in Analysis 2. 77
5.5 Variable hyperparameters in Analysis 3. 79
5.6 Best results for each Study in Analysis 3. 79
5.7 Variable hyperparameters in Analysis 4. 80
5.8 Best results for each Study in Analysis 4. 81
5.9 Variable hyperparameters in Analysis 5. 83
5.10 Best results for each Study in Analysis 5. 83
5.11 Variable hyperparameters in Analysis 6. 85
5.12 Best results for each Study in Analysis 6. 85
5.13 Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for TunedCNN. 85
5.14 Average Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for TunedCNN. 86

5.15 Parameters evolution from GlobAvgPoolCNN to TunedCNN. 86
5.16 Calculation of final dataset size. 92
5.17 Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for TunedCNN for 300 000 RVEs. 92
5.18 Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for TunedCNN for Augmented Dataset. 93
5.19 Variable hyperparameters in Study 1. 94
5.20 Variable hyperparameters in Study 2. 95
5.21 Variable hyperparameters in Study 3. 96
5.22 Variable hyperparameters in Study 4. 97
5.23 Variable hyperparameters in Study 5. 99
5.24 Variable hyperparameters in Study 6. 100
5.25 Variable hyperparameters in Study 7. 101
5.26 Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for FinalCNN. 102

List of Symbols

Abbreviations

2D	Two Dimentional
3D	Three Dimentional
AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CV	Cross Validation
dof	Degree of Freedom
FE	Finite Element
FEM	Finite Element Method
FNN	Feedforward Neural Network
GUI	Graphic User Interface
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
ML	Machine Learning
MSE	Mean Square Error
PNG	Portable Network Graphics
ReLu	Rectified Linear Unit
REV	Representative Elementary Volume
RVE	Representative Volume Element
SGD	Stochastic Gradient Descent
SIANN	Space Invariant Artificial Neural Networks

Convention

$(\cdot)^d$	Discretized equivalent of $(\cdot)$
$(\cdot)_\mu$	Micro-scale equivalent of $(\cdot)$
$\mathbf{a}$,$\mathbf{A}$	Tensors
a	Vectors
$\mathscr{A}$	Spaces

Variables

α	Number of Kernels
β	Repeats of Convolutional Block
γ	Repeats of Convolutional & Pooling Block
δ	Size of Dense Layers

ε	Drop rate of first dropout layer
$\boldsymbol{\varepsilon}^{(m)}$	Eulerian strain tensor
ζ	Drop rate of second dropout layer
ρ	Specific mass in the deformed state
$\boldsymbol{\sigma}$	Cauchy stress tensor
$\boldsymbol{\tau}$	Kirchhoff stress tensor
φ	Deformation mapping function
Ω	Domain
Ω^d	Discretized domain
$\partial\Omega$	Boarder of domain
A_t	Actual value in error calculation
$\mathbf{B}$	Discrete symmetric gradient operator
$\boldsymbol{B}$	Left Cauchy-Green strain tensor
$\boldsymbol{b}$	Body force
$\mathscr{B}$	Body occupying a domain in reference position
$\boldsymbol{C}$	Right Cauchy-Green strain tensor
$\boldsymbol{E}^{(m)}$	Lagrangian strain tensors
$\mathscr{E}$	Euclidean space
$\mathbf{f}$	Nodal force vector
$\boldsymbol{F}$	Deformation gradient
$\dot{\boldsymbol{F}}$	Virtual deformation gradient
$F_{i,j}$	Cartesian component of the deformation gradient
F_t	Forecast value in error calculation
g	Activation function
J	Determinant of deformation gradient
L	Length of body
l	Characteristic length
$\mathbf{N}$	Interpolation function matrix
$\boldsymbol{n}$	Vector normal to surface
N_i	Nodal interpolation function
$\boldsymbol{P}$	First Piola-Kirchhoff stress tensor
$\boldsymbol{p}$	Position of a generic particle in the reference configuration
$\boldsymbol{q}$	Center of rotation
$\boldsymbol{R}$	Orthogonal tensor
$\boldsymbol{S}$	Second Piola-Kirchhoff stress tensor
s_{RVE}	Representative volume element size
t	Time variable
$\boldsymbol{t}$	Surface force
$\boldsymbol{U}$	Right stretch tensor
$\boldsymbol{u}$	Displacement
$\tilde{\boldsymbol{u}}$	Displacement fluctuations
$\delta\boldsymbol{u}$	Virtual displacements vector
$\delta\mathbf{u}$	Nodal virtual displacements vector
u_i^j	Nodal displacement of the node j along the degree of freedom i
$\boldsymbol{W}$	Skew tensor
$w_{k,i}$	Weight to multiply input from neuron i in neuron k
x_i	Input from neuron i on neuron k
$\boldsymbol{x}$	Position of a generic particle in the deformed configuration

$\dot{\boldsymbol{x}}$	Velocity of a particle
y_k	Output of neuron k
$\boldsymbol{V}$	Left stretch tensor
$\boldsymbol{v}$	Spatial velocity of a particle

File Types

`.dat`	File containing information for FE^2 Analysis
`.out`	Output file from Links2
`.pkl`	Pickle file, file containing a Pandas DataFrame
`.png`	Portable Network Graphics (PNG) image file
`.rve`	RVE file containing mesh and all parameters for Micro Scale FEM Analysis
`.vtk`	Visualization file of FE^2 Analysis
`.zip`	Archive file containing compressed files

Chapter 1

Introduction

The last decades have completely revolutionized engineering. The invention and spread of the digital computer in the past century marked the beginning of the evolution of computational methods to solve specific problems. Using numerical analysis, machines could solve mathematical equations faster than any human. These matured into the basis of today's engineering across most fields. Technologies such as the Finite Element Method (FEM) became some of the essential tools for designing structures and components. Each year, said methods only become faster and more precise as the processing power of each computer increases.

Another field that has seen significant improvement over the last few years is the field of Artificial Intelligence (AI), more specifically, Machine Learning (ML). Alongside the increasing processing power, the datasets needed to create these AIs are growing incredibly in many fields. These factors have given rise to various versatile models in the most varied application areas over the last few decades. These are, in the end, complex mathematical models, adjusted so that they can capture every feature that composes its input and each associate with a possible output. Although designing and changing them is a very complex and time-consuming task, they can be speedy and capable of consistent results once finished.

With all of this in mind, it is only natural to try and join these robust numerical methods with the compelling relationship establishing machine learning algorithms. And that leads to the core question of this work: can these algorithms capture all necessary details and connections to predict the mechanical behavior of complex microstructural arrangements? But, of course, it gets much more complicated than this. Machine learning is an ever-expanding and complex field, with many different classes, methods, and nuances whose effectiveness in each case can only be truly estimated by trial and error. And even then, it would be unrealistic to expect 100% accuracy from these algorithms. But then again, the numerical methods used in continuum mechanics are not that exact either.

1.1 Context

The FEM by itself is very capable of predicting the behavior of complex structures when under load, but for this, it requires the mechanical properties of what it is analyzing. These can be easily obtained for isotropic materials, as they keep their characteristics independently of loading direction. This, however, becomes a problem when the underlying components that forms them, for some reason, grants them anisotropic behavior. This happens, for instance, when a microstructure is composed of multiple phases with very different material responses. Some arrangements may cause significantly anisotropic properties. As such, the overall response of the microstructure must be taken for every iteration of FEM, and their respective elements refreshed. Another occurrence would be when subjecting a material to extreme strains, as then events at a microscopic scale can compromise its overall properties. With this, rises the need to link the events at a macro scale to their influence in the microstructures, and vice-versa. One of the ways this can be achieved is through the multiscale analysis using the computational homogenization implemented in this work. These use multiple FEM models of the microstructures running inside each integration point of the model containing the macro mesh.

1.2 Objectives

The main objective of this work is to design an ML architecture capable of predicting the homogenized mechanical properties of a double-phase, polycristal microstructure, more specifically, its stress-strain curve. For this, is necessary to create the entire framework capable of developing those microstructures, analyzing them using numerical methods to obtain their homogenized stress tensors and using them to train and optimize the ML models.

1.3 Methodology

For this work, a new framework had to be developed, using Python and Bash. This had to be designed to be able to generate the data, process and clean it. It also had to be able to train models, evaluate them and adjust the hyperparameters accordingly.

The first stage was to generate the data necessary to feed the machine learning algorithms. For this, each sample had its microstructure generated and analyzed, resulting in a stress-strain curve. When this was concluded for all entries, the curves and microstructures had to be compiled into a centralized file.

While the initial dataset was generated, the first machine learning models were designed and tested, marking the beginning of the manual tuning process. Once the layers were selected and there was some rough idea of a working architecture, a software to oversee tuning was implemented.

After obtained a tuned architecture, some data augmentation technics were implemented to improve accuracy. Post these implementations, to extract better results, the model entered a second stage of overseen tuning, concluding the design process.

Chapter 2

Review of Literature

The development of this thesis encompassed multiple areas of study, from the more traditional mechanical engineering fields of continuum mechanics and finite element method, which form the basis of most structural analysis in this work, to the rapidly evolving environment of Machine Learning.

In this chapter, a brief introduction to these fields is presented, focusing on the major aspects of each. More detailed explanations can be found in the references mentioned.

2.1 Continuum Mechanics

Most common mechanical analysis, be it of solid or fluid mediums, are made based on the principles of continuum mechanics and thermodynamics. This allows for a simplification of what would be, otherwise, impractical to calculate, as if one tried to take into account the contribution of every micro and nano constituents of a body at every moment, there would be far too many variables to compute.

The main reason we can normally assume a material as continuous and homogeneous, is due to the massive difference between the scale of the micro and nanostructure and the entire system. A very simple example that allows a visualization of this principle, is to consider a perfect monocrystalline body, as, for instance, a gemstone or a silicon monocrystal, used in electronics. Even though said crystal is constituted by countless atoms, arranged in a crystal structure, to the naked eye, and even to the microscope, there is only a solid body, and we are unable to distinguish an atom from its neighbors. Then, we can consider the body as continuous solid body, occupying a determined volume in space.

This same logic can apply to metals and other conventional poly-crystalline or amorphous materials, as every crystal or other micro constituent is still much smaller than the sum of their parts.

In this section are explained some basic principles and equations for the analysis of continuum mediums, needed for the formulation of the finite element method and multiscale analyses.

2.1.1 Kinematics of Deformation

The deformation of a body is characterized by its variation in size, shape and location in order of time (t). Let there be a body which occupies a region Ω of a three-dimensional Euclidean space $\mathscr{E}$ as its reference configuration, with boundary $\partial\Omega$. A deformation of said body is defined as a function φ such that:

$$\varphi : \Omega \rightarrow \mathscr{E}.$$

This function maps each point in the body to a position, characterizing the deformed configuration. This way, for every point ($\boldsymbol{p}$) in the domain Ω there is a deformed equivalent $\boldsymbol{x}$, such that:

$$\boldsymbol{x} = \varphi(\boldsymbol{p}). \tag{2.1}$$

One may also define the displacement $\boldsymbol{u}(\boldsymbol{p})$ as the difference between a point's original and deformed configuration:

$$\boldsymbol{u}(\boldsymbol{p}) = \varphi(\boldsymbol{p}) - \boldsymbol{p}, \tag{2.2}$$

thus:

$$\boldsymbol{x} = \boldsymbol{p} + \boldsymbol{u}(\boldsymbol{p}). \tag{2.3}$$

Some deformations conserve the distance between each material point, being considered rigid deformations of the body. These can be either rigid translation or rigid rotation.

A rigid deformation can be defined by a constant displacement vector $\boldsymbol{u}$, independent of $\boldsymbol{p}$:

$$\varphi(\boldsymbol{p}) = \boldsymbol{p} + \boldsymbol{u}, \tag{2.4}$$

while a rigid rotation pivoting around a point $\boldsymbol{q}$ can be defined by:

$$\varphi(\boldsymbol{p}) = \boldsymbol{q} + \boldsymbol{R}(\boldsymbol{p} - \boldsymbol{q}), \tag{2.5}$$

where $\boldsymbol{R}$ is an orthonormal tensor, which defines the rotation of the body.

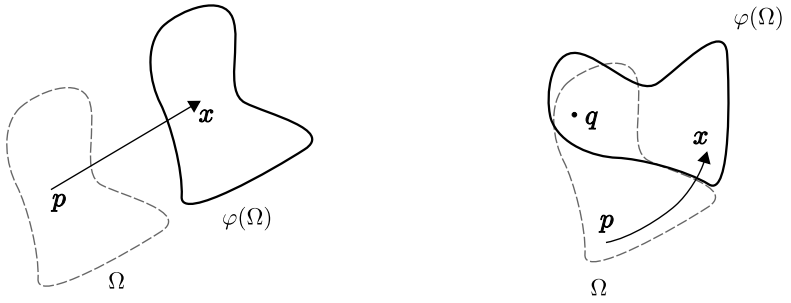
If both rigid translation and rotation occurs, then the resulting rigid deformation is given by:

$$\varphi(\boldsymbol{p}) = \varphi(\boldsymbol{q}) + \boldsymbol{R}(\boldsymbol{p} - \boldsymbol{q}), \tag{2.6}$$

and a deformation can only be considered rigid if it behaves according to Equation (2.6).

If a deformation is time-dependent, then it is considered a motion, being defined by a function

$$\varphi : \Omega \times \mathscr{R} \rightarrow \mathscr{E},$$



(a) (b)

Figure 2.1: Rigid Translation (a) and Rotation (b).

so for every instant $t \in \mathscr{R}$, corresponds a deformed state $\varphi(\Omega,t)$, characterizing a motion of the body.

That said, and introducing time dependency in Equation (2.1) and combining Equation (2.3), one obtains:

$$\boldsymbol{x}(\boldsymbol{p},t) = \varphi(\boldsymbol{p},t) = \boldsymbol{p} + u(\boldsymbol{p},t), \tag{2.7}$$

from which we can describe the velocity of the material particle $\boldsymbol{p}$ as:

$$\dot{\boldsymbol{x}}(\boldsymbol{p},t) = \frac{\partial \boldsymbol{x}(\boldsymbol{p},t)}{\partial t} = \frac{\partial \boldsymbol{u}(\boldsymbol{p},t)}{\partial t}. \tag{2.8}$$

Similarly, one may define the spatial velocity as the velocity of the material particle at $\boldsymbol{x}$ at time t, given by:

$$\boldsymbol{v}(\boldsymbol{x},t) = \dot{\boldsymbol{x}}(\varphi^{-1}(\boldsymbol{x},t),t). \tag{2.9}$$

Also, from Equation (2.6) we can conclude that a rigid deformation is given by:

$$\boldsymbol{v}(\boldsymbol{x},t) = \boldsymbol{v}(\boldsymbol{y},t) + \boldsymbol{W}(t)(\boldsymbol{x}-\boldsymbol{y}) \ , \ \boldsymbol{x},\boldsymbol{y} \in \varphi(\Omega,t) \tag{2.10}$$

being $\boldsymbol{W}(t)$ a skew tensor. This states that the instantaneous velocity of $\boldsymbol{x}$ is given by the velocity of the pivot point ($\boldsymbol{y}$) and a superimposed rotation, whose axis is parallel to the vector associated to $\boldsymbol{W}$.

2.1.2 Deformation Gradient

One of the most important attributes to characterize a body's motion and stress state and to predict its ability to withstand increased loads, is its variation in shape, and how that variation changes along $\varphi(\Omega)$. With this in mind, it's clear the need to be able to deduce this value from the motion φ, as this one, by itself, is unable to provide us this information. This can be achieved by analyzing

the vector separating two infinitesimal neighboring particles, $\boldsymbol{p}$ and $\boldsymbol{p}+d\boldsymbol{p}$, both contained in Ω, and how said vector changes after applying φ (Figure 2.2).

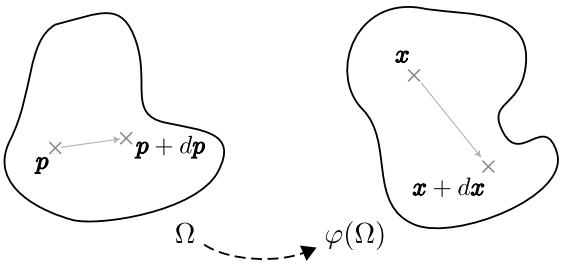


Figure 2.2: Motion φ on the body $\mathscr{B}$.

Being $\boldsymbol{x}=\varphi(\boldsymbol{p})$ and $\boldsymbol{x}+d\boldsymbol{x}=\varphi(\boldsymbol{p}+d\boldsymbol{p})$, it is possible to establish $\boldsymbol{F}$ as the relation between both $d\boldsymbol{x}$ and $d\boldsymbol{p}$ as they represent the same vector in different spatial configurations, such that:

$$d\boldsymbol{x}=\boldsymbol{F}d\boldsymbol{p}, \tag{2.11}$$

where the deformation gradient ($\boldsymbol{F}$) of a motion φ is the mathematical representation of the changes in shape a body is subjected to from the space Ω to $\varphi(\Omega)$. It is given by the second-order tensor $\boldsymbol{F}$ such that:

$$\boldsymbol{F}(\boldsymbol{p},t)=\nabla_p\varphi(\boldsymbol{p},t)=\frac{\partial \boldsymbol{x}_t}{\partial \boldsymbol{p}}, \tag{2.12}$$

and can also be rewritten as:

$$\boldsymbol{F}=\boldsymbol{I}+\nabla_p\boldsymbol{u}, \tag{2.13}$$

where $\boldsymbol{I}$ is the identity matrix.

The Cartesian components of $\boldsymbol{F}$ then are given by:

$$\begin{aligned} F_{ij} &= \frac{\partial x_i}{\partial p_j} \\ &= \delta_{ij}+\frac{\partial u_i}{\partial p_j}, \end{aligned} \tag{2.14}$$

where x_i denote the components of $\boldsymbol{x}_t$ and δ_{ij} being the *Kronecker delta*:

$$\delta_{ij}=\begin{cases} 1, & \text{for } i=j, \\ 0, & \text{for } i\neq j. \end{cases} \tag{2.15}$$

One may also write:

$$\boldsymbol{F}(\boldsymbol{x},t)=[\nabla_x\varphi^{-1}(\boldsymbol{x},t)]^{-1}=[\boldsymbol{I}-\nabla_x\boldsymbol{u}]^{-1}. \tag{2.16}$$

Analyzing an infinitesimal volume of space dV contained in Ω and its variation when applied φ is possible to associate its variation with the determinant of $\boldsymbol{F}$:

$$dv = JdV, \tag{2.17}$$

where

$$J = \det \boldsymbol{F}. \tag{2.18}$$

2.1.3 Strain Measures

There are many possible ways to measure a body's deformation, as multiple tensors have been proposed throughout the years, all of them derived from the deformation gradient. The right and left *Cauchy-Green* ($\boldsymbol{C}$ and $\boldsymbol{B}$) strain tensors are some of the most commonly used. Although other options, such as the right and left stretch tensors ($\boldsymbol{U}$ and $\boldsymbol{V}$) are available, $\boldsymbol{C}$ and $\boldsymbol{B}$ are easier to implement, making them more viable options for many calculations. They can be expressed as:

$$\boldsymbol{C} = \boldsymbol{F}^T \boldsymbol{F}, \tag{2.19}$$

$$\boldsymbol{B} = \boldsymbol{F}\boldsymbol{F}^T. \tag{2.20}$$

Of these two, the most convenient quantity for solid analysis is the right *Cauchy-Green* tensor, $\boldsymbol{C}$, as it is symmetric and is defined at the reference configuration, contrary to $\boldsymbol{B}$, defined at the deformed configuration. They can be related to the Lagrangian strain tensors ($\boldsymbol{E}^{(m)}$) by:

$$\boldsymbol{E}^2 = \frac{1}{2}(\boldsymbol{C} - \boldsymbol{I}). \tag{2.21}$$

It is also possible to express $\boldsymbol{E}^{(m)}$ in relation to the right stretch tensor $\boldsymbol{U}$:

$$\boldsymbol{E}^{(m)} = \begin{cases} \frac{1}{m}(\boldsymbol{U}^m - \boldsymbol{I}), & \text{if } m \neq 0, \\ \ln(\boldsymbol{U}), & \text{if } m = 0, \end{cases} \tag{2.22}$$

with a similar relation between the Eulerian strain tensor ($\boldsymbol{\varepsilon}^{(m)}$) and the left stretch tensor $\boldsymbol{V}$:

$$\boldsymbol{\varepsilon}^{(m)} = \begin{cases} \frac{1}{m}(\boldsymbol{V}^m - \boldsymbol{I}), & \text{if } m \neq 0, \\ \ln(\boldsymbol{V}), & \text{if } m = 0. \end{cases} \tag{2.23}$$

$\boldsymbol{E}^{(m)}$ and $\boldsymbol{\varepsilon}^{(m)}$ are also known as the generalized strain tensors, in the material and spatial configuration, respectively.

2.1.4 Stress Measures

A motion happens due to the presence of unbalanced forces. This might deform the body, of simply move it from its original position. There are two different kinds of forces that might be

applied in the body:

- Surface force: these are applied in the boundaries of the material and propagate along it, for example, pushing or pulling an object;
- Body force: these actuate in the body's mass itself, being dependent on the volume of it, like gravitational or magnetic pulls.

Stress is a measure to evaluate how much force is being applied in the material, and is calculated as force over an area of the section of interest.

Similarly to strains, there are multiple ways to measure them, being some of the most important listed bellow.

Cauchy Stress Tensor

This is the most common measure for stresses, also called the true stress.

Being $\boldsymbol{t}$ the surface traction following the outwards normal vector $\boldsymbol{n}$ to the outside surface of a volume, the Cauch stress tensor is given by:

$$\boldsymbol{t}(\boldsymbol{n},t) = \boldsymbol{\sigma} \cdot \boldsymbol{n}. \tag{2.24}$$

where $\boldsymbol{\sigma}$ is symmetric and is composed of the deviatoric stress tensor ($\boldsymbol{s}$) and a hydrostatic component ($p = 1/3 \times \mathrm{tr}\boldsymbol{\sigma}$)

Kirchhoff Stress Tensor

Also known as the weighted Cauchy stress tensor (τ), this one is widely used when there is no change in volume during plastic deformation. Is given by:

$$\boldsymbol{\tau} = J\boldsymbol{\sigma}, \tag{2.25}$$

recalling Equation (2.18):

$$J = \det(\boldsymbol{F}).$$

First and Second Piola-Kirchhoff Stress Tensors

The first Piola-Kirchhoff stress tensor ($\boldsymbol{P}$) does not represent a material property, as is used only for its convenience. On the other hand, the second stress tensor ($\boldsymbol{S}$) relates the true reference surface traction.

$$\boldsymbol{P} = J\boldsymbol{\sigma}\boldsymbol{F}^{-T}, \tag{2.26}$$

$$\boldsymbol{S} = \boldsymbol{F}^{-1} \cdot \boldsymbol{P}. \tag{2.27}$$

2.1.5 Laws of Thermodynamics on Deformation

Beside the continuum mechanics covered thus far, another set of rules that govern the deformation of a body $\mathscr{B}$ are the laws of thermodynamics. Of those, the most important in the context of this work are the conservation of mass and momentum equilibrium.

Conservation of Mass

The principle of conservation of mass states that the mass of a body after deformation must be the same as prior to it. It can be formulated as:

$$\dot{\rho} + \rho \ \mathrm{div}_x(\dot{\boldsymbol{u}}) = 0, \tag{2.28}$$

where ρ is the specific mass in the deformed state and $\mathrm{div}_x(\cdot)$ is the spatial divergence.

Momentum Equilibrium

Let $\mathscr{B}$ be a body with deformed domain $\varphi(\Omega)$ and boundary $\varphi(\partial\Omega)$. This body is under the forces $\boldsymbol{t}(\boldsymbol{n},\boldsymbol{x})$ and $\boldsymbol{b}(\boldsymbol{x})$ where $\boldsymbol{b}$ stands for the body forces. Balancing the effects of inertia and the acceleration field $\ddot{\boldsymbol{u}}$:

$$\int_{\varphi(\partial\Omega)} \boldsymbol{t}(\boldsymbol{n},\boldsymbol{p})dA + \int_{\varphi(\Omega)} \rho(\boldsymbol{p})\boldsymbol{b}(\boldsymbol{p})dV = \int_{\varphi(\Omega)} \rho(\boldsymbol{p})\ddot{\boldsymbol{u}}(\boldsymbol{p})dV, \tag{2.29}$$

$$\int_{\varphi(\partial\Omega)} \boldsymbol{p}\times\boldsymbol{t}(\boldsymbol{n},\boldsymbol{p})dA + \int_{\varphi(\Omega)} \boldsymbol{p}\times\rho(\boldsymbol{p})\boldsymbol{b}(\boldsymbol{p})dV = \int_{\varphi(\Omega)} \boldsymbol{p}\times\rho(\boldsymbol{p})\ddot{\boldsymbol{u}}(\boldsymbol{p})dV, \tag{2.30}$$

which can be re-written as:

$$\begin{cases} \mathrm{div}_x(\boldsymbol{\sigma}) + \boldsymbol{b} = \rho\ddot{\boldsymbol{u}} & \text{in } \varphi(\Omega), \\ \boldsymbol{t} = \boldsymbol{\sigma}\boldsymbol{n} & \text{in } \varphi(\partial\Omega). \end{cases} \tag{2.31}$$

These are known as the strong form equations, and relate the displacements with the forces applied on the body.

2.2 Finite Element Method

Over the decades, the processes used by engineers to determine the loading capabilities of components and structures have evolved, and never have they developed as much as with the advancements of computers and their respective processing power. Whist before all calculations were done using pen and paper, and resorting to massive approximations in order to be executable in working time, nowadays, as we improved our understanding of how to harness said power to modernize our methodologies, allowing them to become more complex, precise and reliable, which caused a surge in more efficient engineering processes than the ones common 60 to 70 years ago.

One of these processes, that in the past 50 years has established itself as one of the most important and versatile tools for engineers is the Finite Element Method, which consists of discretizing the part or region of space at hand into multiple elements to be analyzed both separately and as a bundle, and to approximate the unknown variables as functions within each one. Whist this concept was not revolutionary and engineers have been breaking complex problems into more manageable chunks for many years prior, the technological development of the 60s allowed the automation of these tasks, with smaller elements and more accurate numerical analysis than ever before.

Said discretization consisted of specifying a finite number of nodes and elements in space, discretizing the domain volume, in a process called meshing. For each node it's possible to establish boundary conditions, representative of the model intended, as well as loads that may apply to the element. Then, for each one, the equilibrium conditions are calculated, applying numerical solutions to the equilibrium equations, which outputs an approximation of the variables on each node. Then, applying the interpolation equations to obtain the variation within each element, and combining the results for each one, it is possible to easily evaluate the approximate status of the entire domain for each load applied. This allowed engineers to analyze complex problems with increased accuracy, on a shorter time frame.

Currently, multiple companies develop commercial software capable for performing finite element analysis whether they allow more control over the every parameter, such as ABAQUS and Ansys, or in a more user-friendly interface, but restraining control over the analysis framework, as in Solidworks.

In this section there is a brief overview of the process, starting in the weak form of the differential equation, all the way to the space discretization that defines the method. A complete analysis of this method and its formulation can be found on the work by Oñate [1].

2.2.1 Weak Form of the Equilibrium Equation

The strong form equation (2.31) can be hard to solve in numerical implementations. For this very reason, there is a need to deduct a different form with easier numerical solutions. This new form (weak form) relaxes some continuity and differentiability requirements on potential solutions, making it easier to deduct possible solutions for this form.

This can be obtained by the application of the Virtual Work Principle on (2.31), resulting in:

$$\int_{\varphi(\Omega)} [\boldsymbol{\sigma} : \nabla_x \delta\mathbf{u} - \mathbf{b} \cdot \delta\mathbf{u}]\, dV - \int_{\varphi(\partial\Omega)} \boldsymbol{t} \cdot \delta\mathbf{u} dA = 0, \quad \forall \delta\mathbf{u} \in \mathscr{V}, \tag{2.32}$$

where $\delta\mathbf{u}$ are the virtual displacements on the admissible space $\mathscr{V}$.

2.2.2 Space Discretization

The basic concept of FEM is to separate the domain Ω into smaller chunks (n_{elem}), called finite elements, each with its own domain $\Omega^{(e)}$. These, when combined, form an approximation of

the actual domain (Ω^d), and, if the spatial discretization was properly done, with the appropriate number and form of elements (Figure 2.3), this should be similar to Ω:

$$\Omega \approx \Omega^d, \tag{2.33}$$

where $(\cdot)^d$ corresponds to the discretization equivalent of $(\cdot)$.

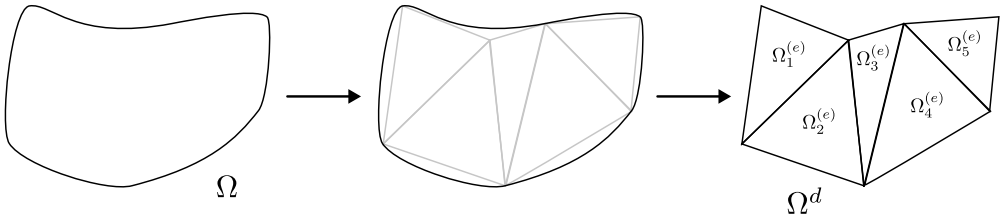


Figure 2.3: Space Discretization of the domain Ω.

The method itself works by determining the interest values, usually displacements, at the n_{points} that form the mesh with n_{elem} elements. As there are n_{dof} Degrees of Freedom (dof) for each node point, the interest vector containing all displacements in the mesh is given by:

$$\mathbf{u} = \{u_1^1,\ u_2^1,\ ...,\ u_{n_{dof}}^1,\ u_1^2,\ ...,\ u_{n_{dof}}^{n_{points}}\}^T, \tag{2.34}$$

totaling $n_{dof} \times n_{points}$ interest variables.

To compute these, it's first necessary to establish the interpolation functions ($N_i^e(\boldsymbol{x})$) that will be used later on to establish the interest field inside the element, based on the values at each node:

$$\boldsymbol{u}(\boldsymbol{x}) = \sum_{i=1}^{n_{nodes}} N_i^e(\boldsymbol{x})\mathbf{u}_i, \quad \boldsymbol{x} \in \Omega^{(e)}, \tag{2.35}$$

where n_{nodes} is the number of nodes forming the element. It is also possible to define N_i for Ω^d as:

$$\boldsymbol{u}(\boldsymbol{x}) = \sum_{i=1}^{n_{points}} N_i(\boldsymbol{x})\mathbf{u}_i, \quad \boldsymbol{x} \in \Omega^d. \tag{2.36}$$

The following deduction depends on the dimension of the problem at hand. For the two-dimensional (2D) analyzes such as the ones done in this work, one can establish a matrix $\mathbf{N}$ such that:

$$\mathbf{N} = \begin{bmatrix} N_1(\boldsymbol{x}) & 0 & N_2(\boldsymbol{x}) & 0 & \cdots & N_{n_{points}}(\boldsymbol{x}) & 0 \\ 0 & N_1(\boldsymbol{x}) & 0 & N_2(\boldsymbol{x}) & \cdots & 0 & N_{n_{points}}(\boldsymbol{x}) \end{bmatrix}. \tag{2.37}$$

Once established, one may also define the discrete symmetric gradient operator ($\mathbf{B}$) and discrete spatial gradient operator ($\mathbf{G}$) from the derivatives of N_i:

$$\mathbf{B} = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial x} & 0 & \cdots & \frac{\partial N_{n_{points}}}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial y} & \cdots & 0 & \frac{\partial N_{n_{points}}}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & \cdots & \frac{\partial N_{n_{points}}}{\partial y} & \frac{\partial N_{n_{points}}}{\partial x} \end{bmatrix}, \tag{2.38}$$

Once the discretization is all defined, and looking back to the weak formulation of the equilibrium equation (2.32), one may re-adapt this to the new domain, such that:

$$\left\{ \int_{\varphi(\Omega^d)} \left[(\mathbf{B})^T \boldsymbol{\sigma} - \mathbf{N}^T \mathbf{b} \right] dV - \int_{\varphi(\partial\Omega^d)} \mathbf{N}^T \mathbf{t} dA \right\}^T \cdot \delta\mathbf{u} = 0, \quad \forall \delta\mathbf{u} \in \mathscr{V}^d, \tag{2.39}$$

this must be possible for all possible $\delta\mathbf{u}$ contained in the discretized virtual displacement space, thus:

$$\mathbf{f}^{int} - \mathbf{f}^{ext} = 0, \tag{2.40}$$

where:

$$\mathbf{f}^{int} = \int_{\varphi(^h\Omega)} (\mathbf{B}^g)^T \boldsymbol{\sigma} dV, \tag{2.41}$$

$$\mathbf{f}^{ext} = \int_{\varphi(^h\Omega)} \mathbf{N}^T \mathbf{b} dV + \int_{\varphi(\partial\, ^h\Omega)} \mathbf{N}^T \mathbf{t} dA. \tag{2.42}$$

These calculations can be done element wise, where the global $\mathbf{f}^{int}$ and $\mathbf{f}^{ext}$ are then assembled from their element counterparts.

2.3 Multi-Scale Modelling Based in Computational Homogenization

Although many solid mechanics analysis can assume homogeneous materials, this might not always be the case. Sometimes, variation on composition or treatment of some areas may result in different properties along the domain Ω, or it can be anisotropic due to some phase changes or inclusions in the microstructure. This can happen on classic materials, such as steel or aluminum alloys, when subjected to surface of heat treatments, resulting in different properties with depth. Another occurrence of these phenomena is the ever more common composites, such reinforced polymers, whose direction and layout grant the overall anisotropic characteristics of the material. In some applications, it might be of interest to vary these characteristics in the same component, further more complicating calculations.

To take into account these variations in the microstructure (or micro scale) onto the material's behavior, its often necessary to establish an association between them. This can be done utilizing multi-scale models based on computational homogenization, that combine representative volume elements (RVEs) of the microstructure and the finite element method to calculate the macro stress field, depending on each RVE's response to displacements. The models used in this work are

Coupled Multi-Scale Finite Element Analysis, or FE^2, as both macro and micro scale calculations are done concurrently. In a decoupled analysis, these would be executed separately.

In this method, the displacement field is calculated on the macro-scale, and forced as a homogenized displacement field to each RVE, located in the elements' integration points. Then, once the microscale displacement field is estimated, the stress field is calculated, from which a homogenized stress can be computed through the formulation described in this section. Once this process is repeated for every integration point, the macro scale stress field can be assembled and the analysis can move to the next iteration.

In this section, the main concepts of FE^2 analyzes are presented, from the formulation needed to associate both scales, to the basics of its numerical implementation, and it was build by studying the works by Souza Neto et al [2] and Rodrigues Lopes [3].

2.3.1 Representative Volume Element, the Micro Scale Domain

A Representative Volume Element (RVE), or Representative Elementary Volume (REV), is a statistical representation of the microstructure, and it is composed of the same phases, has the same formations and represents the overall material's properties. These can be defined as "the smallest volume over which a measurement can be made that will yield a value representative of the whole" and "a material volume which is statistically representative of the infinitesimal material neighborhood of that material point".

Establishing the size of this volume can be very intricate. For crystalline microstructures and similarly random materials, it must be large enough to represent the overall volume, but it must be as small as possible to reduce computing costs. This is based on the Scale Separation Principle, that states that the size of the RVE (s_{RVE}) must be much larger than those that constitute it (characteristic length l) but much smaller than the macroscopic continuum:

$$l << s_{RVE} << L. \tag{2.43}$$

The size section is much simpler on periodic materials (where its structure is periodic), as it only needs to be the periodic unit itself (Figure 2.4).

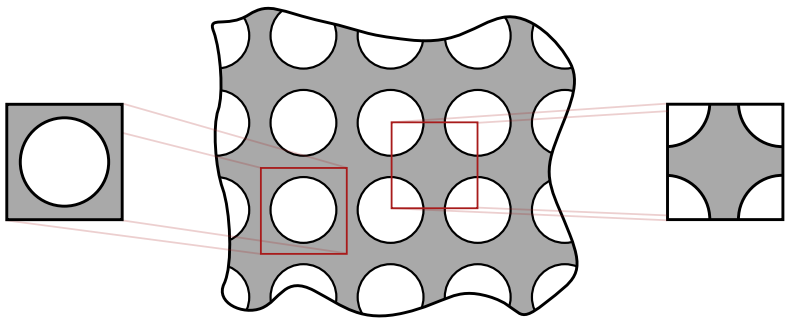


Figure 2.4: Periodic microstructure and two possible RVEs.

Worth noting that in this document the term RVE is loosely used to refer to a mesh of a RVE and its respective entry in the datasets.

2.3.2 Homogenized Deformation Gradient

As briefly explained before, the relation between scales is established as the deformation gradient at a RVE's integration point is passed down to the microscale, in the form of a homogenized deformation gradient ($\boldsymbol{F}$), equal to its macro equivalent.

Considering that $(\cdot)_\mu$ denotes the micro scale variant of the unit $(\cdot)$, one can declare $\boldsymbol{p}_\mu$ as being part of the RVE's domain Ω_μ, with undeformed volume $v_{0,\mu}$. It is then possible to establish the relation between $\boldsymbol{F}$ and $\boldsymbol{F}_\mu$ as

$$\begin{aligned}\boldsymbol{F}(\boldsymbol{x},t) &= \frac{1}{v_{0,\mu}}\int_{\Omega_\mu}\boldsymbol{F}_\mu(\boldsymbol{p}_\mu,t)\mathrm{d}v \\ &= \boldsymbol{I}+\frac{1}{v_{0,\mu}}\int_{\Omega_\mu}\nabla_{\boldsymbol{p}}[\boldsymbol{u}_\mu(\boldsymbol{p}_\mu,t)]\mathrm{d}v.\end{aligned} \tag{2.44}$$

Applying the Gauss Theorem:

$$\boldsymbol{F}(\boldsymbol{x},t) = \boldsymbol{I}+\frac{1}{v_{0,\mu}}\int_{\Omega_\mu}\boldsymbol{u}_\mu(\boldsymbol{p}_\mu,t)\otimes\boldsymbol{n}_{0,\mu}(\boldsymbol{p}_\mu)\mathrm{d}a, \tag{2.45}$$

where $\boldsymbol{n}_{0,\mu}$ is the unit vector normal to the boundary of Ω_μ, pointing outwards.

2.3.3 Multi-Scale Kinematics

As the $\boldsymbol{F}(\boldsymbol{x},t)$ is imposed on the equations (2.44) and (2.45), this limits the possible $\boldsymbol{u}_\mu$. As such, it is possible to establish a minimally constrained set of kinematically admissible microscopic displacements ($\mathscr{K}^*_\mu$), such that

$$\boldsymbol{u}_\mu \in \mathscr{K}^*_\mu, \tag{2.46}$$

where

$$\mathscr{K}^*_\mu \equiv \left\{\boldsymbol{u}_\mu, \text{sufficiently regular}\middle|\int_{\Omega_\mu}\boldsymbol{u}_\mu(\boldsymbol{p}_\mu,t)\otimes\boldsymbol{n}_{0,\mu}(\boldsymbol{p}_\mu)\mathrm{d}a = v_{0,\mu}[\boldsymbol{F}(\boldsymbol{x},t)-\boldsymbol{I}]\right\}. \tag{2.47}$$

The linear displacement may be obtained from the decomposition of $\boldsymbol{u}_\mu$ into its linear component:

$$\boldsymbol{u}_\mu(\boldsymbol{p}_\mu,t) = [\boldsymbol{F}(\boldsymbol{x},t)-\boldsymbol{I}]\cdot\boldsymbol{p}_\mu+\tilde{\boldsymbol{u}}_\mu(\boldsymbol{p}_\mu,t), \tag{2.48}$$

where $\tilde{\boldsymbol{u}}_\mu$ is the displacement fluctuation, and is the only variable, as $\boldsymbol{F}(\boldsymbol{x},t)$ is imposed and $\boldsymbol{u}_\mu(\boldsymbol{p}_\mu,t)$ is known.

Is then possible to establish $\boldsymbol{F}_\mu$ in function of $\tilde{\boldsymbol{u}}_\mu$ such that

$$\begin{aligned}\boldsymbol{F}_\mu(\boldsymbol{x},t) &= \boldsymbol{F}(\boldsymbol{x},t) + \nabla_{\boldsymbol{p}}[\tilde{\boldsymbol{u}}_\mu(\boldsymbol{p}_\mu,t)] \\ &= \boldsymbol{F}(\boldsymbol{x},t) + \frac{1}{v_{0,\mu}}\int_{\Omega_\mu} \nabla_{\boldsymbol{p}}[\tilde{\boldsymbol{u}}_\mu(\boldsymbol{x},t)]\mathrm{d}v.\end{aligned} \tag{2.49}$$

By reorganizing, canceling $\boldsymbol{F}_\mu(\boldsymbol{x},t)$ and applying the Gauss Theorem isolates the minimal kinematically admissible constraint:

$$\begin{aligned}\int_{\Omega_\mu} \nabla_{\boldsymbol{p}}[\tilde{\boldsymbol{u}}_\mu(\boldsymbol{x},t)]\mathrm{d}v = \boldsymbol{0} \\ \int_{\Omega_\mu} \tilde{\boldsymbol{u}}_\mu(\boldsymbol{x},t)\otimes\boldsymbol{n}_{0,\mu}(\boldsymbol{p}_\mu)\mathrm{d}a = \boldsymbol{0}\end{aligned} \tag{2.50}$$

Form this, is possible to assemble the minimally constrained space of displacement fluctuations:

$$\mathscr{K}_\mu^* \equiv \left\{\tilde{\boldsymbol{u}}_\mu, \text{sufficiently regular} \middle| \int_{\Omega_\mu} \tilde{\boldsymbol{u}}_\mu(\boldsymbol{x},t)\otimes\boldsymbol{n}_{0,\mu}(\boldsymbol{p}_\mu)\mathrm{d}a = \boldsymbol{0}\right\}, \tag{2.51}$$

however, as some other constraints may apply as a result of boundary conditions, it should be defined:

$$\tilde{\boldsymbol{u}}_\mu \in \mathscr{K}_\mu \subset \mathscr{K}_\mu^*, \tag{2.52}$$

where $\mathscr{K}_\mu$ is the resulting actual set of kinematically admissible fluctuation field, once all is considered.

2.3.4 The Hill-Mandel Macro-Homogeneity Condition

The Hill-Mandel Macro-Homogeneity Condition is one of the most important equations for scale bridging which states that the virtual work on the entire micro domain Ω_μ must be equal to the associated integration point in the macro model.

The mathematical representation of this is given by:

$$\boldsymbol{P}(\boldsymbol{x},t) : \dot{\boldsymbol{F}}(\boldsymbol{x},t) = \frac{1}{v_{o,\mu}}\int_{\Omega_\mu} \boldsymbol{P}(\boldsymbol{p}_\mu,t) : \dot{\boldsymbol{F}}(\boldsymbol{p}_\mu,t), \tag{2.53}$$

with:

$$\dot{\boldsymbol{F}}_\mu(\boldsymbol{x},t) = \dot{\boldsymbol{F}}(\boldsymbol{x},t) + \nabla_{\boldsymbol{p}}[\dot{\tilde{\boldsymbol{u}}}_\mu(\boldsymbol{p}_\mu,t)], \tag{2.54}$$

where $\dot{\boldsymbol{F}}$ and $\dot{\boldsymbol{F}}_\mu$ are the deformation gradient rates, or virtual deformation gradients. Combining (2.53) and (2.54):

$$\boldsymbol{P}(\boldsymbol{x},t):\dot{\boldsymbol{F}}(\boldsymbol{x},t)=\boldsymbol{P}(\boldsymbol{x},t):\dot{\boldsymbol{F}}(\boldsymbol{x},t)+\frac{1}{v_{o,\mu}}\int_{\Omega_\mu}\boldsymbol{P}_\mu(\boldsymbol{p}_\mu,t):\nabla_{\boldsymbol{p}}[\dot{\tilde{\boldsymbol{u}}}_\mu(\boldsymbol{p}_\mu,t)]\mathrm{d}v. \tag{2.55}$$

Reorganizing:

$$\int_{\Omega_\mu}\boldsymbol{P}_\mu(\boldsymbol{p}_\mu,t):\nabla_{\boldsymbol{p}}[\dot{\tilde{\boldsymbol{u}}}_\mu(\boldsymbol{p}_\mu,t)]\mathrm{d}v=0,\ \forall\dot{\tilde{\boldsymbol{u}}}_\mu\in\mathscr{V}_\mu, \tag{2.56}$$

where $\mathscr{V}_\mu$ corresponds to the space of virtual kinematically admissible displacements, that

$$\mathscr{V}_\mu\equiv\tilde{\mathscr{K}_\mu}. \tag{2.57}$$

From this is possible to obtain the strong form of micro scale equilibrium:

$$\begin{aligned}\int_{\Omega_\mu}\boldsymbol{P}_\mu(\boldsymbol{p}_\mu,t):\nabla_{\boldsymbol{p}}[\dot{\tilde{\boldsymbol{u}}}_\mu(\boldsymbol{p}_\mu,t)]\mathrm{d}v&=\int_{\partial\Omega_\mu}\boldsymbol{t}_{0,\mu}(\boldsymbol{p}_\mu,t)\cdot\dot{\tilde{\boldsymbol{u}}}_\mu\mathrm{d}a+\int_{\Omega^s_\mu}\boldsymbol{b}_{0,\mu}(\boldsymbol{p}_\mu,t)\cdot\dot{\tilde{\boldsymbol{u}}}_\mu\mathrm{d}v\\0&=\int_{\partial\Omega_\mu}\boldsymbol{t}_{0,\mu}(\boldsymbol{p}_\mu,t)\cdot\dot{\tilde{\boldsymbol{u}}}_\mu\mathrm{d}a+\int_{\Omega^s_\mu}\boldsymbol{b}_{0,\mu}(\boldsymbol{p}_\mu,t)\cdot\dot{\tilde{\boldsymbol{u}}}_\mu\mathrm{d}v\end{aligned} \tag{2.58}$$

From this, it's possible to conclude that the Hill-Mandel condition is only verified if and only if:

$$\int_{\partial\Omega_\mu}\boldsymbol{t}_{0,\mu}(\boldsymbol{p}_\mu,t)\cdot\dot{\tilde{\boldsymbol{u}}}_\mu\mathrm{d}a=0\quad\wedge\quad\int_{\Omega^s_\mu}\boldsymbol{b}_{0,\mu}(\boldsymbol{p}_\mu,t)\cdot\dot{\tilde{\boldsymbol{u}}}_\mu\mathrm{d}v=0,\quad\forall\dot{\tilde{\boldsymbol{u}}}_\mu\in\mathscr{V}_\mu. \tag{2.59}$$

2.3.5 Homogenized Stress Tensor

Once the micro displacement field is calculated, it's then possible to compute the micro stress field and to homogenize it, obtaining the value relevant for the macro model.

Knowing that:

$$\boldsymbol{P}(\boldsymbol{x},t)=\frac{1}{v_{0,\mu}}\int_{\Omega_\mu}\boldsymbol{P}_\mu(\boldsymbol{p}_\mu,t)\mathrm{d}v, \tag{2.60}$$

integrating by parts the right side and applying the strong form of equilibrium:

$$\boldsymbol{P}(\boldsymbol{x},t)=\frac{1}{v_{0,\mu}}\left(\int_{\partial\Omega_\mu}\boldsymbol{t}_{0,\mu}\otimes\boldsymbol{p}_\mu\mathrm{d}a-\int_{\Omega_\mu}\boldsymbol{b}_{0,\mu}\otimes\boldsymbol{p}_\mu\mathrm{d}v\right). \tag{2.61}$$

Finally, applying to the equation above the already demonstrated

$$\boldsymbol{P}=J\boldsymbol{\sigma}\boldsymbol{F}^{-T}, \tag{2.62}$$

its possible to obtain the final relation between $\boldsymbol{\sigma}$ and the homogeneised deformation gradient:

$$\boldsymbol{\sigma}=\frac{1}{\det(\boldsymbol{F})}\boldsymbol{P}\boldsymbol{F}^T. \tag{2.63}$$

2.3.6 Micro-Scale Boundary Conditions

Lastly, it is necessary to impose Kinematic constraints in the RVE. These govern the behavior of the micro model and affect how the homogenized properties will respond, so it is necessary to chose one accordingly.

Although there are many boundary conditions available, in this work the periodic boundary condition was tuilized, but will be covered. Those are:

- Taylor boundary condition (${}^{Tay}\tilde{\mathscr{K}}_\mu$);
- Periodic boundary condition (${}^{Per}\tilde{\mathscr{K}}_\mu$);
- Uniform traction boundary condition (${}^{Tay}\tilde{\mathscr{K}}_\mu$).

Although there are many boundary conditions available, only the periodic boundary condition was used in this work, and will be explained side by side with the most and least restrictive boundary conditions, Taylor and Uniform Traction, respectively.

Taylor Boundary Condition

This one is the most restrictive, stating that the displacement function is null over the entire domain:

$$\tilde{\boldsymbol{u}}_\mu(\boldsymbol{p}_\mu,t)=0, \quad \forall \boldsymbol{p}_\mu \in \Omega_\mu \tag{2.64}$$

This results in a displacement $\boldsymbol{u}$ that will vary linearly with $\boldsymbol{p}$ all over Ω_μ.

The actual set of kinematically admissible fluctuation field is then given by:

$$\tilde{\mathscr{K}}_\mu = {}^{Tay}\tilde{\mathscr{K}}_\mu \equiv \left\{\tilde{\boldsymbol{u}}_\mu, \text{ sufficiently regular} \mid \tilde{\boldsymbol{u}}_\mu = \boldsymbol{0}, \quad \forall \boldsymbol{p}_\mu \in \Omega^s_\mu\right\}. \tag{2.65}$$

Periodic Boundary Condition

The periodic boundary condition makes sure that the displacements on opposite sides of the RVE are equal, which essentially signifies that the RVE is repeated periodically (hence the name) as a mosaic. This is particularly useful for periodic microstructures, but also applicable for other formations.

This can be expressed such that opposite sides of the RVE have similar displacements, thus:

$$\tilde{\boldsymbol{u}}_\mu(\boldsymbol{x}^+_\mu,t) \equiv \tilde{\boldsymbol{u}}_\mu(\boldsymbol{x}^-_\mu,t)\,, \quad \forall i \wedge \forall \partial\Omega_{\mu,i} \wedge \forall \{\boldsymbol{x}^+_\mu,\boldsymbol{x}^-_\mu\} \in \partial\Omega_{\mu,i}, \tag{2.66}$$

where $\partial\Omega_{\mu,i}$ is the pair i composed of $\partial\Omega^+_{\mu,i}$ and $\partial\Omega^-_{\mu,i}$, for which $(\cdot)^+$ and $(\cdot)^-$ denote opposing sides of the RVE.

The resulting actual set of kinematically admissible fluctuation field is

$$\tilde{\mathscr{K}}_\mu \equiv^{Per} \tilde{\mathscr{K}}_\mu \equiv \big\{\tilde{\boldsymbol{u}}_\mu, \text{ sufficiently regular } | \\ \tilde{\boldsymbol{u}}_\mu(\boldsymbol{p}_\mu^+,t) = \tilde{\boldsymbol{u}}_\mu(\boldsymbol{p}_\mu^-,t)\,, \quad \forall i \wedge \forall \partial\Omega_{\mu,i} \wedge \forall\{\boldsymbol{x}_\mu^+,\boldsymbol{x}_\mu^-\} \in \partial\Omega_{\mu,i}\big\}\,. \quad (2.67)$$

Uniform Traction Boundary Condition

This is the less limiting boundary condition set and corresponds directly to the minimally constrained space of displacement fluctuations

$$\tilde{\mathscr{K}}_\mu \equiv^{Uni} \tilde{\mathscr{K}}_\mu \equiv \left\{\tilde{\boldsymbol{u}}_\mu, \text{ sufficiently regular } | \int_{\Omega_\mu} \tilde{\boldsymbol{u}}_\mu(\boldsymbol{x},t) \otimes \boldsymbol{n}_{0,\mu}(\boldsymbol{p}_\mu)\mathrm{d}a = \boldsymbol{0}\right\} \equiv \tilde{\mathscr{K}}_\mu^*. \quad (2.68)$$

2.4 Machine Learning

Machine Learning (ML) is a subset of Artificial Intelligence (AI), and is based on the concept of training an algorithm with data for it to perform a certain task. This draws a parallel to how humans learn, by gathering information, identifying patterns and memorizing those relations, as the ML algorithm works in the same way: it is fed data, it identifies patterns from within said data and learns to associate each pattern with the intended output.

Although the bases of ML are already known for many decades, as the first successful applications of these concepts date back to the 1950s, when Alan Turing theorizes a machine that could learn and Arthur Samuel creates the first checkers playing AI, in the past decade this technology has massively advanced. Advancements in statistics and computer science have provided constant improvements to these algorithms, alongside the development in the datasets used for training, resulting in ever more capable machine learning models.

Nowadays, this technology has become an integral part of society, even when not in plain sight. ML models have been developed to predict traffic [4], translate sign language into text or speech [5], handwriting recognition [6], and even to identify target audience for a product [7]. It also has been successfully used to identify cancer in x-rays better than some expert doctors [8] and to distinguish marine life in underwater sonars. Is easy to understand that this technology is extremely powerful and versatile, as successful implementations are endless.

During this project, all implementations of machine learning were made using Tensorflow [9] and Keras [10], using also some functionalities of the scikit-learn package [11]. The theoretical revision in this section is based on the book by Kinsley and Kukieła [12].

When preparing an implementation of these algorithms, it is necessary to understand the two main categories they fall into.

Supervised learning

These are the most common implementations of ML algorithms, as it is very easy to tailor the model to the intended output.

The data used to train the models consists of both the intended inputs the AI will receive, and the expected result. Then, it is up to the algorithm to identify patterns in the data and to refine its ability to recognize these every time it sees a new example.

Most of the applications above rely on supervised learning.

All models trained in this work were through supervised learning, so the focus of this review will be on this methodology.

Unsupervised learning

Far less used than supervised learning, the models are trained with no specific output in mind. One of the main functions of these AIs is to identify similarities between data entries, and grouping them accordingly (clustering).

2.4.1 Artificial Neural Networks and Dense Layers

Many ML models are built upon Artificial Neural Networks (ANNs), inspired by the function of the brain of animals. An ANN is composed of many artificial neurons, each with its own set of inputs and outputs. Usually, these neurons are assembled into layers and connect to the ones in the adjacent ones, resulting in a structure capable of identifying multiple patterns and associate to each a specific output (Figure 2.5).

Layers themselves may have different layouts, resulting in diverse behaviors and pattern recognition capacity. The most easy to understand layer is the so-called dense layer, composed of an array of simple neurons.

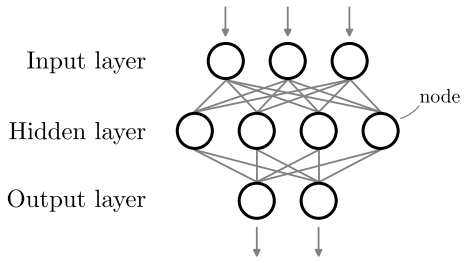


Figure 2.5: Three concurrent dense layers.

2.4.1.1 Artificial Neuron

At its core, an artificial neuron is a mathematical function. It sums the weighted inputs (outputs from the neurons in the previous layer) and passes them through an activation function, generally non-linear, and outputs the resulting value.

Being y_k the output of the neuron k, it can be described as:

$$y_k = g\left(\sum_{j=0}^{m} w_{kj} x_j\right), \tag{2.69}$$

where w_{ki} is the weight in the k neuron to multiply by the output x_j of the i neuron. The function g is the activation function, to be explored next.

The context of this calculation is better understood by consulting the Figure 2.6, representing the k neuron receiving the input from all m neurons from the previous layer and outputs to the next.

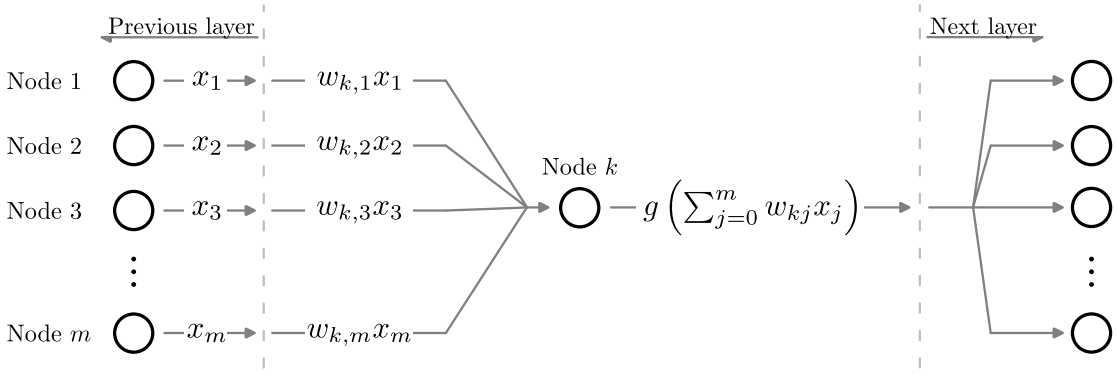


Figure 2.6: Schematic of an artificial neuron.

2.4.1.2 Activation Functions

On ANNs, the usage of non-linear activation functions throughout the nodes allows the neural-networks to compute nontrivial problems, with a limited number of nodes, making it easier for the model to generalize. However, not all activation functions are non-linear, as is possible to bypass this step using a pass-through function. These may also help to limit a node output, by using functions with tailored a codomain.

There are a multitude of activation curves to choose from, with the option of selecting a specific one according to individual need. Some of the most relevant are listed bellow, and plotted on Figure 2.7.

Sigmoid

The logistic or sigmoid function (Figure 2.7a) is a commonly used activation function in ANNs, as it has an output between 0 and 1, making it specially useful when calculating a probability.

$$g(x) = \frac{1}{1+e^{-x}}. \tag{2.70}$$

Rectified Linear Unit

The rectified linear unit (ReLu) (Figure 2.7b) is one of the most used activation functions currently, and is the one used on most layers throughout this work. It sees vast implementation in image processing networks, specially on hidden layers. It has linear behavior over 0, but is null when under it:

$$g(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}. \tag{2.71}$$

Leaky ReLu

This activation function is used to combat one of the main problems with the ReLu function, the "dying ReLu". As its derivative is null under 0, this will negatively affect the backpropagation in the training process. To solve this, it adds a slight slope to the ReLu function in the negative domain (Figure 2.7c):

$$g(x) = \begin{cases} \delta x, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}, \tag{2.72}$$

where most common δ is around 0.1.

Pass-Through

The pass-through, or, simply, linear activation function (Figure 2.7d), does not affect the weighted sum. It is sometimes used in the output layer of a regression neural-networks, such as the ones described in this work.

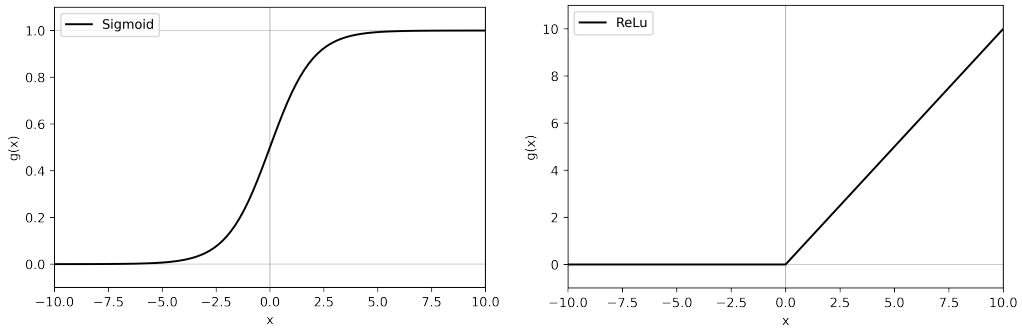
$$g(x) = x \tag{2.73}$$

2.4.2 Training

Training is the process when the algorithm goes through the dataset and optimizes the weights and biases of the model to adapt to the training data. This is repeated for almost all the dataset multiple times, with the objective of getting the model to get as close of a prediction as possible.

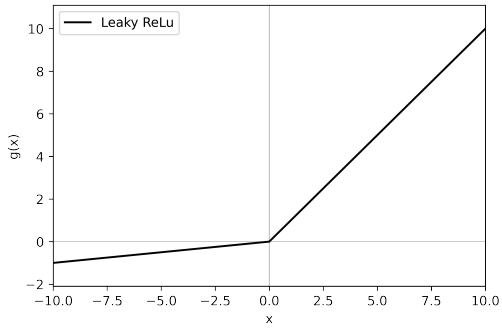
A prediction is evaluated using a loss function, serving as a metric of the model accuracy. The higher the loss, the less precise the model. The algorithm uses it for each entry in the dataset to adjust the parameters accordingly.

There are many different loss functions, but the three most used are:

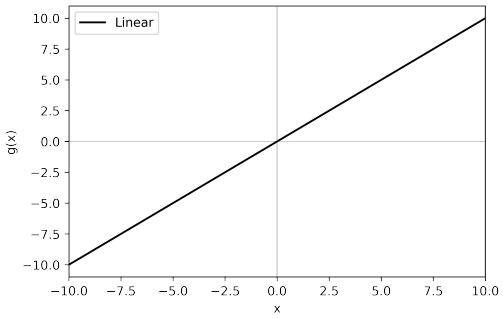


(a) Sigmoid

(b) ReLu



(c) Leaky ReLu



(d) Linear

Figure 2.7: Plot of Sigmoid (a), ReLu (b), Leaky ReLu (c) and Linear (d) activation functions.

Mean Square Error Loss

The Mean Square Error (MSE) (Figure 2.8a) is one of the most popular loss functions, as it is very simple to understand and is sensitive to outliers. However, this is also its main disadvantage, because it amplifies any possible error in the dataset. It can be calculated by:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(A_{t,i} - F_{t,i})^2, \tag{2.74}$$

where A_t is the actual value and F_t is the forecast, both with N components.

Mean Absolute Error Loss

Even though the Mean Absolute Error (MAE) (Figure 2.8b) formulation is very similar to MSE's:

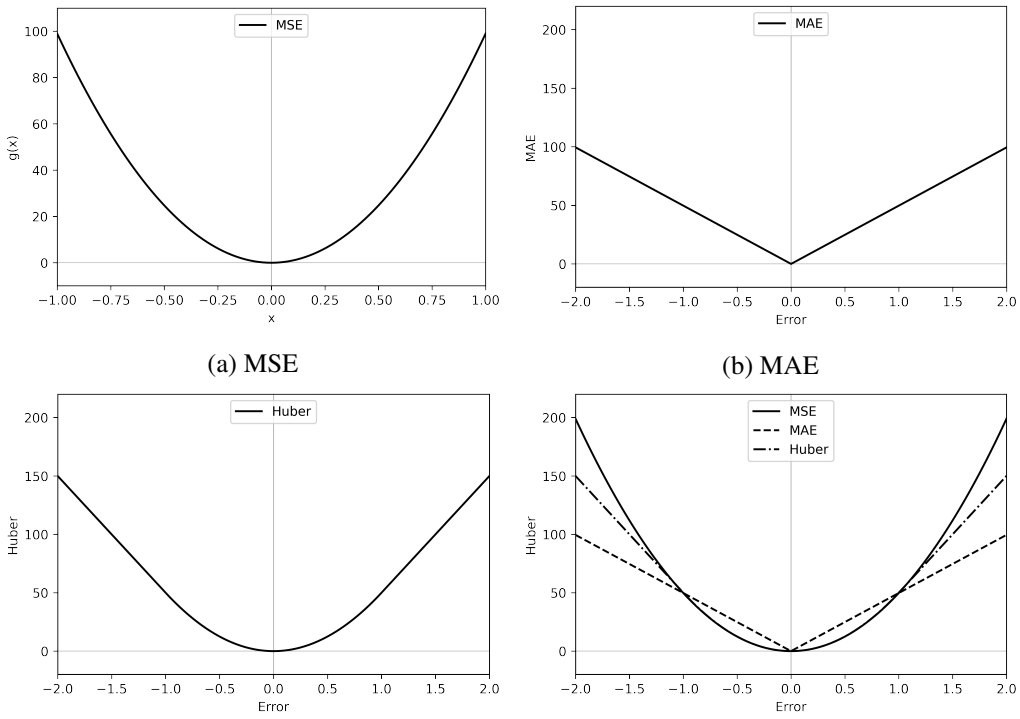
$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|A_t - F_t|, \tag{2.75}$$

it has the exact opposite effect, being not sensitive to outliers in the dataset. This might be useful when dealing with unclean data, where only the predictions for the general model are intended.

Huber Loss

This loss function sits somewhere in the middle of the two previously explained. It simply states that, if an absolute error is higher than a certain δ, it uses MAE, otherwise, uses MSE (Figure 2.8c and 2.8d). This results in a balanced evaluation criteria, allowing the outliers to impact the model, but not destabilize it.

$$L_i = \begin{cases} 0.5(y_i - \hat{y}_i)^2, & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta|y_i - \hat{y}_i| - 0.5\delta^2, & \text{if } |y_i - \hat{y}_i| < \delta \end{cases}, \tag{2.76}$$



(a) MSE

(b) MAE

(c) Huber

(d) Comparisson

Figure 2.8: Plot of MSE (a), MAE (b) and Huber (c) with direct comparison (d).

2.4.2.1 Stochastic Gradient Descent

The method utilized by most ML algorithms to optimize the biases and weights on the data is called the stochastic gradient descent (SGD), which is an iterative method to minimize an objective function. For ML, the objective function is:

$$Q(w) = \frac{1}{n}\sum_{i=1}^{n} Q_i(w), \tag{2.77}$$

where the parameter w is the one intended and each summand function Q_i is associated with the loss at the i observation of the dataset.

When trying to optimize the objective function, one could try to use batch gradient descent:

$$w := w - \eta \nabla Q(w) = w - \eta \frac{1}{n} \sum_{i=1}^{n} \nabla Q_i(w), \tag{2.78}$$

where η is the learning rate.

In stochastic gradient descent, an approximation of $\nabla Q(w)$ is made to:

$$w := w - \eta \nabla Q_i(w). \tag{2.79}$$

This is performed for each entry in the training set, and repeats the process (epochs) until convergence.

In some cases, a learning rate optimizer, such as Adam [13] can be implemented, reducing η as epochs go by, to help with convergence.

2.4.2.2 Difference between parameters and hyperparameters

Lastly, to avoid any confusion, is worth to make clear the difference in nomenclature between parameters and hyperparameters.

Parameters are the values within the model that change during training, i.e. the weights for every node. These are the variables to determine every training to obtain the best results.

Hyperparameters are the values that do not change during training, such as batch sizes, number of layers, number of nodes per layer, etc. These cannot be optimized numerically the same way parameters are, and are harder to tune, as most of the process is manual, by trial and error.

2.4.3 Convolutional Neural Networks

Convolutional Neural Networks (CNN) or Space Invariant Artificial Neural Networks (SIANN) are a class of ANNs mostly used for image analysis.

These are built around the convolution kernels or filters, capable of analyzing data and mounting feature maps. Those are capable of identifying the features that constitute each image, and this allows a post convolutional fully connected neural network to capture the remaining relations to the final output.

The two processes that characterize the CNN class are the convolutional and pooling layers.

2.4.3.1 Convolutional Layers

Convolutional layers receive a 2D dimensional matrix of data distributed by n channels, and compute a number of kernels, where the same element though all of them is responsible for analyzing same area in the input matrix (Figure 2.9).

This establishes a number of feature maps, each capable of identifying the patterns in the input matrix. This is a cost-effective way of capturing relations in table-like arrays of data. This happens because each neuron only receives input from a selective area in from the previous layer, instead of the full layer. To maintain dimensions, this requires that, when dealing with elements close to

the edges of the input matrix, so that the filtered matrix itself overlays the margin, some additional elements need be considered. This is what is called padding, and can be configured so that said additional elements are considered all equal, the same as the edge element on the original matrix or the ones on the other side of it, as it is displayed in a periodic arrangement.

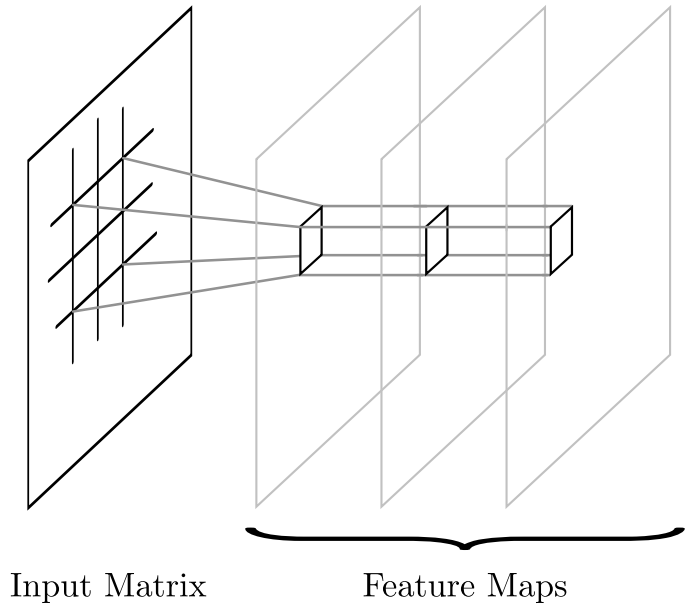


Figure 2.9: Schematic of a convolutional layer.

2.4.3.2 Pooling Process

The pooling process is the responsible for reducing the dimension of data along the neural network. This process works by applying a 2×2 filter to the input matrix, which isolates a specific area. The output variable, usually, contains the maximum value amongst the filtered elements. The filter is then moved by 1 element in the input, and the process is repeated, until the entire matrix is run through the pooling algorithm. The output elements from all filters are then assembled into a new matrix, with dimension reduced by one (Figure 2.10). The filter size and its movement may vary according to architecture, but this configuration is the most common.

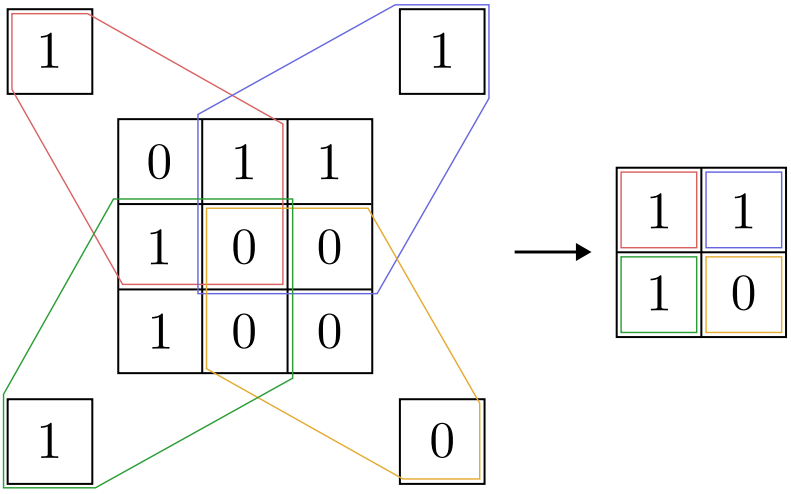


Figure 2.10: Max Pooling of a 3×3 2D Matrix, with a 2×2 filter and stride 1.

Chapter 3

Initial Dataset, generation and considerations

As explained in Chapter 2, in order to obtain a capable Machine Learning AI one needs an extensive dataset. This would allow the model to extract the relationship between the features that compose its inputs and their respective outputs. The first step in creating one for this purpose is generating the RVEs necessary to form the foundation of it, followed by a FE^2 analysis of the same microstructures to extract their curves. Finally, the relevant information is extracted and compiled. In total, 100 000 unique RVEs were generated and analyzed for this initial dataset.

3.1 Material Choice

For the material used in this dataset, it was chosen a soft matrix of aluminum, with 20% volume of boron particles. This was chosen as it would allow the dataset to have multiple behaviors in the curve.

Phase 1 - Soft

This is the dominant phase in the analysis, representing about 80% of the RVE's elements (this value varies slightly from RVE to RVE).

For its modeling, it was defined as a Von Misses material, with an elastic regime defined by its stiffness and Poisson' ratio:

$$E_1 = 72 \text{ GPa},$$

$$\varepsilon_1 = 0.333.$$

Its strain hardening was defined by its matrix:

$$\begin{bmatrix} 0.0 & 352 \\ 1.0 & 752 \end{bmatrix}.$$

Phase 2 - Hard

This phase is inserted in between the elements of the soft one, and is significantly harder than the first. As such, it can be modelled as an elastic material, given only by a certain point in its linear elastic deformation:

$$E_2 = 400 \text{ GPa},$$

$$\varepsilon_2 = 0.3.$$

3.2 RVE and Mesh Generation - Neper

One of the hardest parameters to establish starting out is how large does the dataset need to be. An architecture trained with a dataset too small won't be able to associate the features composing each input with the respective output. On the other hand, it is not desirable to work with massive amounts of data, as these have increased computing costs and slow down both model design and training substantially.

Even though is not yet very common to see neural networks applied in this kind of research, and publications that go into detail about dataset size and generation are scarce, Yang et al. [14][15] was successful in obtaining acceptable results to the engineering curve of pre-fissured microstructures resorting to CNN models with a dataset of 100 000 RVEs and respective curves. Although the problem itself is considerably different, being that these microstructures did not include a fissure and were generated based on crystalline growth, while theirs were composed of a random distribution of stiff elements on a soft matrix, it shined a light on how massive this dataset would become.

For this application, it was not possible to obtain the data through to experimental analysis, that is, to observe the microstructure of a sample and to model it, one by one. This process is, itself, very expensive and time-consuming, even for small quantities. The only way to generate the necessary data, in a fast and reliable way, was to turn to computer generated microstructures and to use them as foundation for the dataset.

For this purpose, the software used was Neper [16], an open-source package, capable of polycrystal generation and meshing. The version used was 3.4.0 and was run on Linux based systems.

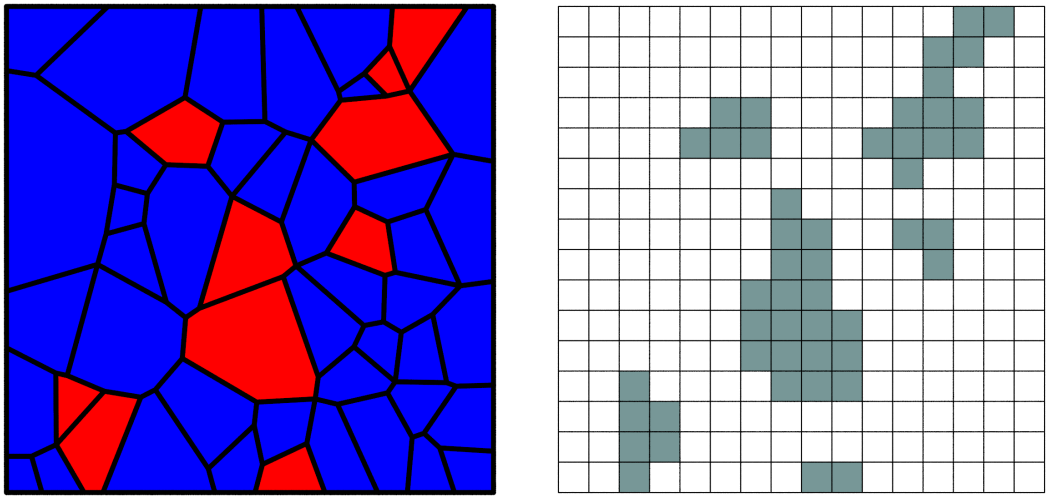
Neper was responsible for not only for generating the microstructure itself, but to mesh it as well, outputting most of the information regarding the RVEs necessary for FE^2 analysis. It also served as a support software, capable of generating visualizations for some key microstructures, although for most of them, to save computing resources, said figures were not created.

This software has three different modules, each of them responsible for a specific task.

Tessellation Module (-T)

This is the main module of Neper, and is responsible for generating the polyphase structure itself. It uses Voronoi tessellation, which completely fills the space with multiple, non overlapping

entities, followed by a Regularisation step, which removes small edges, facilitating the meshing process [17][18].



(a) Microstructure (b) Mesh

Figure 3.1: 2D RVE generated by Neper included in the Dataset (a), and respective mesh (b). The red crystals represent the hard phase, the blue ones represent the soft.

In Figure 3.1a is represented a 2D RVE generated for this dataset, by the method described above.

Meshing Module (-M)

This module generates meshes based on the microstructures obtained with -T. It's possible to change all parameters regarding meshing, such as element type and size.

Figure 3.1b represents the mesh obtained when applying -M to the microstructure in Figure 3.1a.

Visualization Module (-V)

This module allows the visualization of output files from the other modules, in form of PNG images. Both images in Figure 3.1 where generated using it.

3.2.1 Operating Neper

Neper does not have a Graphic User Interface (GUI), and is operated from Terminal. To do so, the program is called, followed by the intended module and the necessary arguments to run it:

```
>> $  neper <module_name> <module_arguments>
```

Two steps are necessary to create the RVE to be fed into both the FE^2 software and the machine learning algorithms:

1. Generate the microstructure itself, using the -T module to fill a 2D space with the two different phases;
2. Mesh the microstructure, using -M.

Starting by the Tesselation step, it was necessary to establish some parameters that would be the arguments called alongside the module, those being:

- Number of grains (`-n`): 50;
- Seeds (`-id`): for the initial dataset of 100 000 RVEs, the seeds used were integers between 1 and slightly over 100 000, as will be explained later;
- Crystal Orientation (`-ori` and `-oricrysym`): specifies how Neper will distribute crystals, e.g. either random or uniform, and it's symmetry, e.g. cubic;
- Regularization (`-regularization` and `-mloop`): turns on and off the regularization step post tessellation, as well as defining maximum number of loops;
- Phase's diameter distribution (`-morpho`): calculated from the volume fractions of each phase;
- Dimension (`-dim`): if the microstructure to generate is 2D or 3D;

It is worth noting that each seed is unique, that is, using the same seed, with the same parameters, in the same machine, Neper outputs the same microstructure every time. This does change with different computers, however.

The final command to generate the microstructure as per the parameters above would be:

```
>> $ neper -rcfile none -T -n 50 -id <seed> -dim 2 -morpho '"$MOR
   PHO"' -ori random -oricrysym cubic -regularization 1 -mloop
   100 -o 2-phase_50-grain_qua8_23-seed -format tess,ori,geo -o
   ridescriptor e -statcell modeid,area
```

With this, Neper will start the tessellation process, followed by the regularization step and outputs the final microstructure. Once finished, the meshing process it-self can start, and for this the -M module is called, followed by all mesh-related parameters:

- Element Type (`-eltype`): choose between available element types, such as tri, quad, tet or hex;
- Characteristic Length (`-cl` or `-rcl`): select characteristic length or relative characteristic length;
- Element Order (`-order`): select either linear or quadratic elements.

Resulting in the command:

```
>> $ neper -rcfile none -M -tmp $XDG_RUNTIME_DIR 2-phase_50-grain
   _qua8_<seed>-seed.tess -elttype quad -cl 0.0625 -dim 2 -orde
   r 2 -format msh -statelt modeid,area
```

This creates the final mesh, ready to be converted into the file type `.rve`. This is one of the two files to be read by the FE^2 software and later by python to create the `.pkl` database.

To facilitate this step, Neper was always run using a modified version of a bash script provided by the CM2S research group, named `neper_by_phase.sh`, which acts like a user interface for Neper, assembling the commands above, facilitating the process. This script calls both the tessellation and meshing modules, alongside generating multiple `.png` files representing the microstructure. It also builds the `.rve` file, containing the mesh, as well as material properties, loads and steps. Some of these parameters are explained in the following Section, as they are related to Links2 and it's workings.

The main modifications done to this script were adding a toggle to the -V modules, creating an option to skip `.png` generation, as these were unnecessary for the formation of the dataset, saving computing resources; and reading the seeds selected in a different, purpose built, bash script, named `mainbash.sh`, to be covered in Section 3.4.

3.3 FE^2 Analysis - Links2

Links is an in-house Finite Element Analysis program, developed by the CM2S research group, which allows the performing of multiscale analyzes based on computational homogenisation. The version used for the dataset generation was Links2, capable of performing FE^2 calculations, as each analysis has a macro and microscale component.

Similarly to Neper, this software does not include a GUI, and has to be run from Terminal by the following entry:

```
>> $ mpiexec -n 2 <Links2_location> <.dat_file_location>
```

3.3.1 FE^2 Modeling

The main objective of this FE^2 model is to simulate a component under stress, in similar conditions to a test coupon in a traction test. For this effect, and bearing in mind that each model is meant to only contain one RVE, a single macro element was needed. This element was subjected to the loading conditions emulating those of a traction test, so that:

$$\varepsilon_0 = 0 \quad \text{and} \quad \varepsilon_{final} = 0.3.$$

This element is then tied to the RVE intended to study (Figure 3.2), and its properties are re-calculated at each increment.

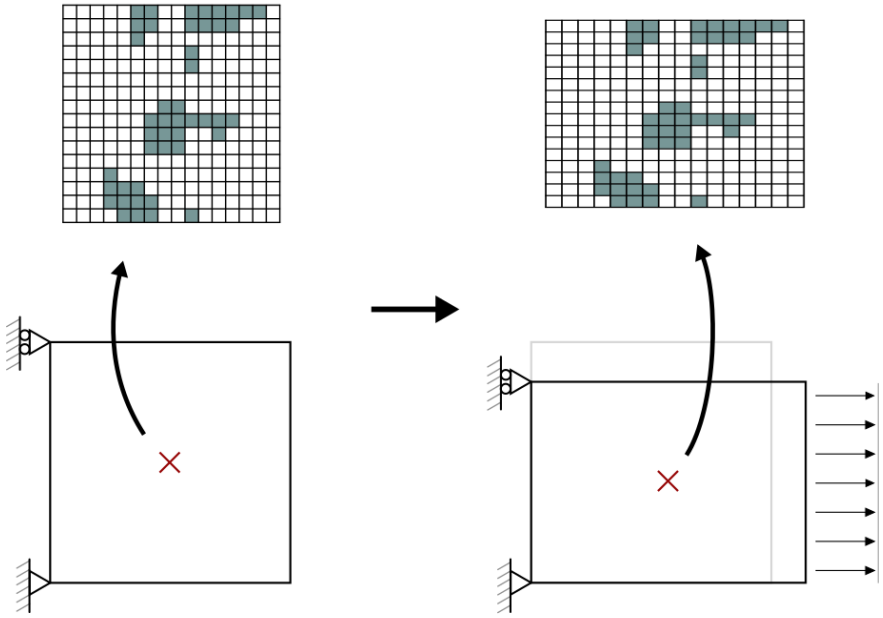


Figure 3.2: FE^2 model.

For the microscale boundary conditions, the chosen was the periodic boundary condition. This establishes that the strain is equal on opposing sides of the RVE, simulating what would happen if the element was placed in a matrix as part of a repeating mosaic.

3.3.1.1 Macro Scale

As the object of interest for each entry in the dataset are relative to the microscale analyses, all settings for macro scale calculations were constant for all RVEs.

The loading applied was in the form of prescribed horizontal displacements for both nodes in the left of the element, which, paired with a constant number of increments, should provide the homogenized horizontal stress for each RVE for the same set of strains. Despite this, not all samples had equal increments. Some needed additional sub-increments to complete the analyses, whist other, incremented at different intervals. The prior were still included in the dataset, but the sub-increments were filtered off as all curves must share strain points to be included in the machine learning algorithm. The former, however, proven inviable to include in the dataset. As these also represented a very small portion of the samples generated (circa 0.2%), the cause for this discrepancy was not very thoroughly investigated, and extra RVEs were generated to take their place instead. The maximum strain of 0.3 in the macro scale was chosen as, even tho it is a significantly high value for most loaded structures, it can be locally reached in some concentration spots in these, justifying the need to evaluate the models up to this point.

The number of increments chosen was 10, resulting in an 11 point stress-strain curve, where the first element is the initial state ($\sigma_{xx} = 0$). This was a compromise between having a high

enough resolution in the curve to worth identifying and the added computational cost each increment would have, accumulated over thousands of RVEs. With the last version of the RVE generation framework developed during this thesis, it would be possible to increase considerably this number, but this decision was taken with an older, unoptimized version in mind. To increase the number of elements, it would be necessary to re-generate the entire dataset, and wasn't viable in the timeframe available.

All the information regarding the single element mesh and loading of the macro scale model were contained into a `.dat` file, being this the file called by Links2. The `.dat` itself redirects to the `.rve` file in the same directory, containing the microscale parameters.

3.3.1.2 Micro Scale

The `.rve` is one of the most important files when generating the dataset, and contained not only the RVE itself, but also all parameters regarding the microscale component of the FE^2 analyzes. These were selected in `neper_by_phase.sh` and were included in the file when it was created.

Each RVE is composed of 256 quadratic quadrilateral elements (QUAD8), totaling 833 nodes. Each element's properties are tied to the respective material.

3.3.2 Operating Links2

Links was operated in similar manner to Neper, and a bash scrip was programmed to ease its execution. The program itself has to be compiled for the computer that will run it.

Once called Links2 and indicating the location of `.dat`, the FE^2 analysis commence, generating a folder containing all output files. In these files, there is a `.screen` file containing the terminal log for future reference, `.vtk` files and, most importantly, a `.out` document.

The `.vtk` files allow the inspecting of the FE^2 run in a visualization application such as Paraview. With this is possible to see the deformation of the RVE with every increment, in addition to the stress field, Material and Element ID. This allows a more detailed analysis of an individual RVE, and will be the one used every time there is need for a more careful inspection of a specific sample, due to any irregularity. However, most of the times, these files are simply archived.

The `.out` file regarding the only element in the macro scale model is the most important file of these, as it contains the homogenized stress at every increment, alongside other properties. This is the Links2 output file to be read by the Python framework.

3.4 Automation

Recapping, in order to obtain the data necessary to feed the AI, firstly one must generate the RVEs and their respective mesh, which get stored in `.rve` files. The following step is to run FE^2 analyzes on said meshes using Links2, generating `.out` files which contains, amongst other values, the homogenized stress and strain at every iteration. Lastly, all relevant information must

be selected from `.rve` and `.out`, treated and compiled into a single file containing all different microstructures generated (Figure 3.3).

As this process isn't humanly possible to repeat manually for all 100 000 RVEs, it was automated as much as possible, separating it in multiple steps.

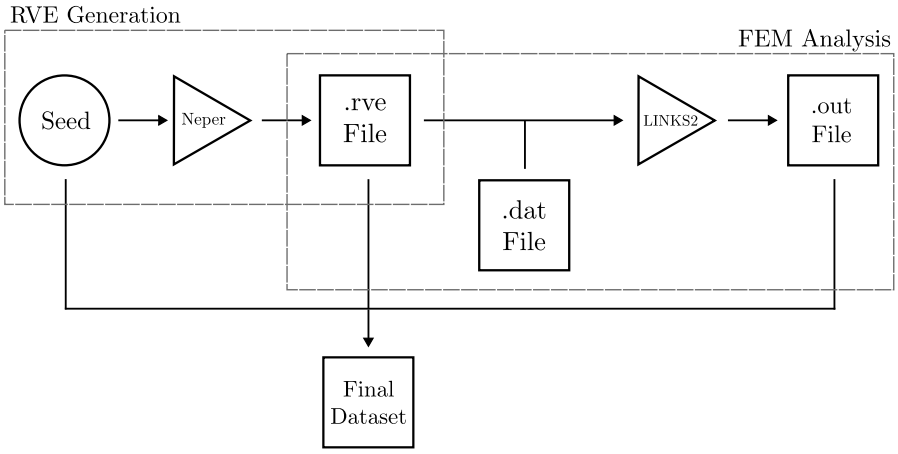


Figure 3.3: Schematic for Dataset Assembly.

3.4.1 Running neper_by_phase.sh and Links2

As both these steps are very time-consuming and computational intensive, it was crucial to fully automate these analyses so that they could run for days, or even weeks at a time without supervision.

The first step was to call Neper, by running `neper_by_phase.sh`, followed by Links2 for every microstructure. This was accomplished by a simple bash script (Algorithm 1 and Appendix A) which runs both in that order. This one also served as the main control script, where the seeds to be generated and analyzed where specified. Lastly, all output files from both programs were compressed into a `.zip` file, mostly to facilitate moving them between computers, as they total hundreds of gigabytes.

Algorithm 1 Running Neper and Links2.

```
seeds ← {s_1, s_2, ..., s_n}                    ▷ Seeds to generate
Bash neper_by_phase_modded.sh
for seed in seeds do
    Creates .dat file                            ▷ Needed to run Links for seed
end for
Run Links2 in parallel
Compress all files
```

It was also in the best interest to optimize these steps as much as possible, as faster analysis would allow for a larger dataset. As Neper, by default, uses all CPU cores when running, there was little to optimize on tessellation and meshing, but Links2, on the other hand, uses only 2

CPU cores per process. As such, it's justified to use parallelization, that is, running multiple FE^2 analysis simultaneously, using all cores. The machine creating these samples had an 8 core CPU, meaning that, theoretically, it's possible to run up to four processes, with little impact on individual calculation time. This would accelerate this step substantially.

A simple test was devised to verify if this method would actually reduce calculation time. For this, a sample of 25 random RVEs where selected. This sample was then analyzed using Links2 and the elapsed time registered. This was repeated for 2, 3 and 4 parallel processes, in addition to a control run with a single process. The sample was kept the same between runs. The results (Table 3.1) proven that running parallelization could vastly accelerate this process.

Table 3.1: Results of Parallelization Test.

Parallel Processes	Elapsed Time	% of time saved
1	4m03s35	—
2	2m04s77	48.7%
3	1m56s03	52.3%
4	1m45s07	56.7%

For this very reason, it was faster to generate all desired RVEs first and analyzing them after, instead of running Neper and Links2 interchangeably for each one.

3.4.2 Separating Useful Files

The first step in treating the data outputted by Links2 is to isolate the interest files, those being the `.rve` and `.out`. This was done mostly as storage management, as these files are around 35kB combined, whilst all files regarding a specific RVE are approximately 2MB. This means that the useful documents makeup about 7% of storage used per sample. While this difference for a single RVE is negligible, when a managing 100 000 runs, the storage saved by isolating the necessary files and compressing and moving the unnecessary ones to external storage, proved very valuable. This was accomplished using a very simple bash script (Appendix B).

3.4.3 Compile All Curves and Microstructures

The last step in treating the data was to convert it to a more usable format, such as spreadsheets or dataframes. This was done reading both `.rve` and `.out` files for each seed, isolating the microstructure and necessary information for plotting the stress-strain curve and adding it to a Pandas DataFrame.

This proved challenging as within each file there is a lot of information that needed to be filtered off, and the data desired had to be converted to more usable formats.

The information deemed necessary to include in the dataset was, for each entry:

- Seed: integer, as it was unique for each RVE, it served as identification;

- Microstructure: array of 256 integers, representing the microstructure;
- Stress: array of 11 floats, stress at every increment;
- Strain: array of 11 floats, strain at every increment;
- Area under the stress-strain curve: added later to the dataset, and calculated using the composed trapezoidal rule, proved useful for stratifying the data.

Once this process was completed for every seed, it was possible to save the dataframe as a single Pickle (`.pkl`) file, easily read by Python whenever needed.

This entire process was coded in Python (Append C) and followed Algorithm 2.

Algorithm 2 Compile all curves and microstructures.

```
df ← empty DataFrame
for seed in seeds do
    micro ← empty Array
    for each line in .rve do                          ▷ Reading Microstructure from .rve
        if line is from Microstructure then
            element ← part of line
            Append element to micro
        end if
    end for
    stresses ← empty Array
    strain ← empty Array
    for each line in .out do                          ▷ Reading Stress-Strain Curve from .out
        if line not first then
            stress ← part of line
            strain ← part of line
            Append stress to stresses
            Append strain to strains
        end if
    end for
    Drops excessive strains and stresses              ▷ Filters off sub-increments
    if number of remaining strains and stresses = 11 then
        Adds seed, micro, stresses and strains to df
    end if
end for
Saves df to .pkl
```

3.5 Analysis

After over two months of continuous data generation and analysis, the initial dataset of 100 000 RVEs was ready for model training. The files created for the entire dataset total 216 GB, but were compressed down to a 362 MB `.pkl` file containing all relevant information for each sample. This

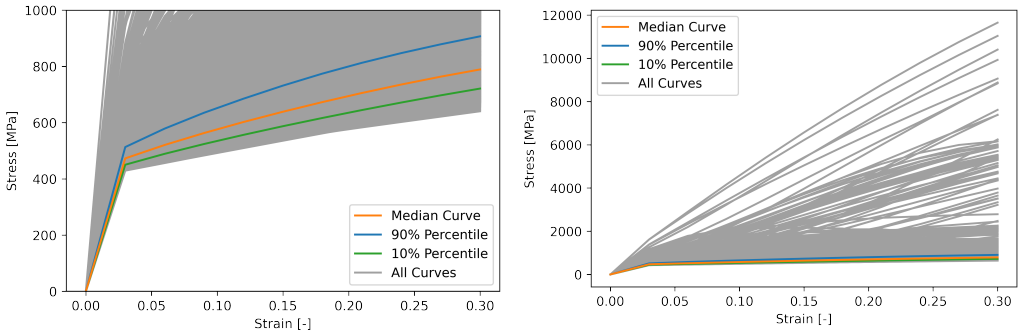
shows the advantage of file separation and using a space efficient file type such as pickle to store a dataset this size.

The last version of the RVE generation framework was capable of generating 50 000 RVEs per week, from seed selection, all the way to `.pkl`. The computer used for this purpose had an Intel Core i7-3770K processor with 16GB of RAM, without GPU acceleration.

This section contains a brief overview of the 100 000 curves and microstructures, analyzing the median curve, density and some cases that stand out with particular behaviors.

3.5.1 Curves Distribution

To understand the dataset itself, is necessary to understand the distribution of all stress-strain curves. In Figure 3.4 the curve obtained from the median value at each strain point along all samples is plotted, together with the 90% and 10% percentiles, obtained in the same manner. In the same figure, all 100 000 curves were drawn, granting a sense of how varied this dataset really is.



(a) Focused on median curve

(b) Global view

Figure 3.4: Plot of all 100 000 curves in the initial dataset.

From Figure 3.4a, it is easy to understand the median material behavior. There is an initial elastic stage over the first increment, followed by plastic deformation up to about 750 MPa at $\varepsilon = 0.3$. From the position of both percentiles is also obvious the high density of curves surrounding the median, with most RVEs having similar curves to this one.

However, the most impressive plot might be Figure 3.4b, as it fully shows every curve. This gives a scale of the entire dataset, and is shows how high some of these curves can reach, and how different behaviors the machine learning model will need to capture. Some of these particular curves will be explored on Section 3.5.2.

One parameter that proven useful to separate the most common curves from the outliers was the area under each one, as briefly mentioned before. The microstructures with abnormal behavior tend to cover larger areas, as they reach higher stresses. As such, is also quite valuable to understand the distribution of this parameter (Figure 3.5), as it illustrates the actual variation in the dataset, and the concentration of curves around the median value. From this, is expected that

the ones farther away from the most dense interval will prove to be the ones where the machine learning algorithm will have the worst performance.

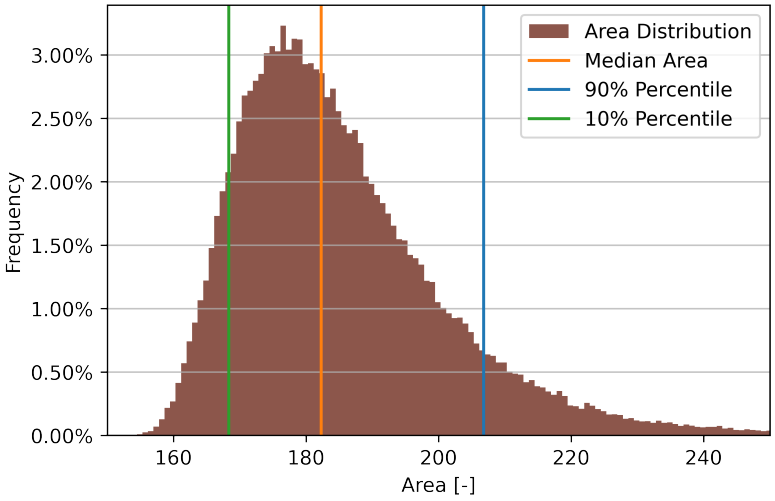


Figure 3.5: Area under the curve distribution.

Over the course of this thesis, to aid neural network evaluation, the Mean Absolute Percentage Error (MAPE) between model predictions and actual curves will be frequently used, in both histograms and percentages model's predictions with MAPE bellow certain thresholds.

To give a sense of scale for those values, and set a point of origin moving forward, a "mean absolute percentage displacement from median" was calculated for each RVE, using the same equation as MAPE (Equation (3.1)) were the median curve took the place of the actual value (A_t) and each sample the forecast value (F_t). These values (Table 3.2) show that the curves are still sufficiently distributed for the training to make sense, and to illustrate what these differences signify curve-wise, on Figure 3.6 are plotted both the median curve, and deviations of in by these percentages.

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{A_t - F_t}{A_t} \right| \tag{3.1}$$

Table 3.2: "Mean Absolute Percentage Displacement from Median" for the initial Dataset.

Mean Displacement from Median	10%	5%	2.5%	1%
Percentage bellow	82.1%	4.1%	2.1%	0.8%

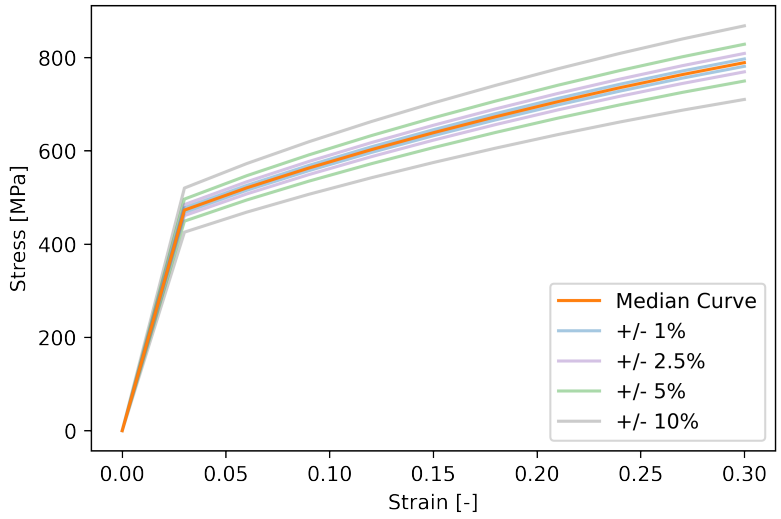


Figure 3.6: Median curve and the interval at both 10%, 5%, 2.5% and 1%.

3.5.2 Outliers

Whist most samples are still in the bounds of the median curve, and have similar responses on a concentrated area, some RVEs have very peculiar responses. It's important to understand why that might be, to help interpret how the AI model will behave and how it can be improved later on.

From the analysis done to the dataset it's possible to identify two main reasons to why, for the same percentages of the same phases, some geometries have such vastly different performances. Some have particular RVEs have structures within them, allowing the rigid phase to absorb more energy, while others have deformations outright questionable, resulting in unexpected variation of homogenized stresses along the curve.

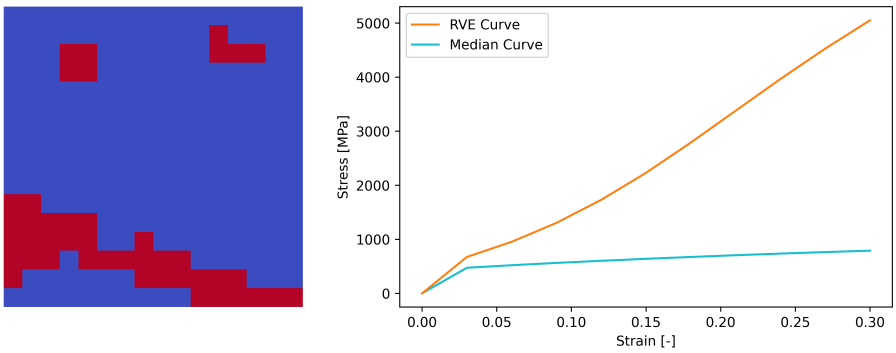
3.5.2.1 Beam and Fiber-like structures

Some of the most important technological advances in the past decades in the field of materials for mechanical applications have been in composite materials. From satellites and airplanes to road cars, fiber glass and carbon fiber have become common substitutes for more classical mechanical alloys whenever strength-to-weight ratio is a critical factor. In essence, these are simply a fabric or, in some applications, just the fiber itself, immersed in a soft polymeric matrix. When under load, the fibers along the loading axis absorb most of the deformation energy.[19][20]

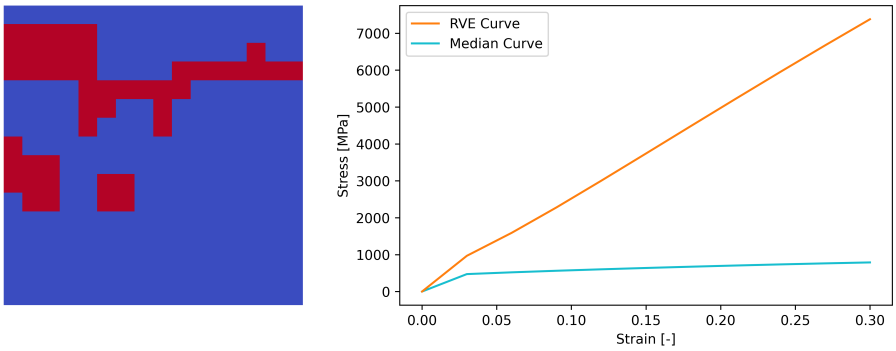
As this dataset was generated based on microstructures with a pseudo-random distribution of 50 grains, converted into a 16×16 2D mesh, it is expected that in some cases, the rigid elements might align side-by-side along the entire RVE. In these samples, the rigid structure inside the soft matrix will result in properties vastly different from the median of the dataset, composed of a less organized distribution of the same elements.

Whenever this structure would form in the horizontal, parallel to the loading of the macro element, it would act similarly to a composite material, where the rigid phase wouldn't allow the

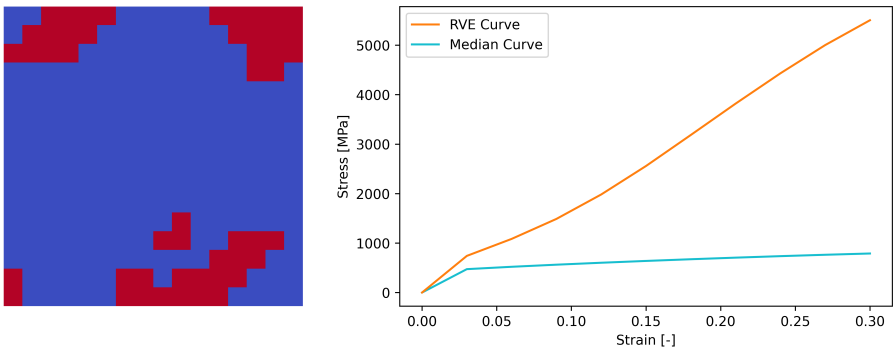
RVE to deform as much along that axis. On Figure 3.7 are presented some examples of these RVEs, alongside their respective stress-strain curves. Worth noting that, although Figure 3.7c may not look like a horizontal alignment of the rigid elements, due to the periodic boundary condition applied on the micro scale, the top and bottom elements act like they are next to each other, forming a continuous line across the entire RVE.



(a)



(b)

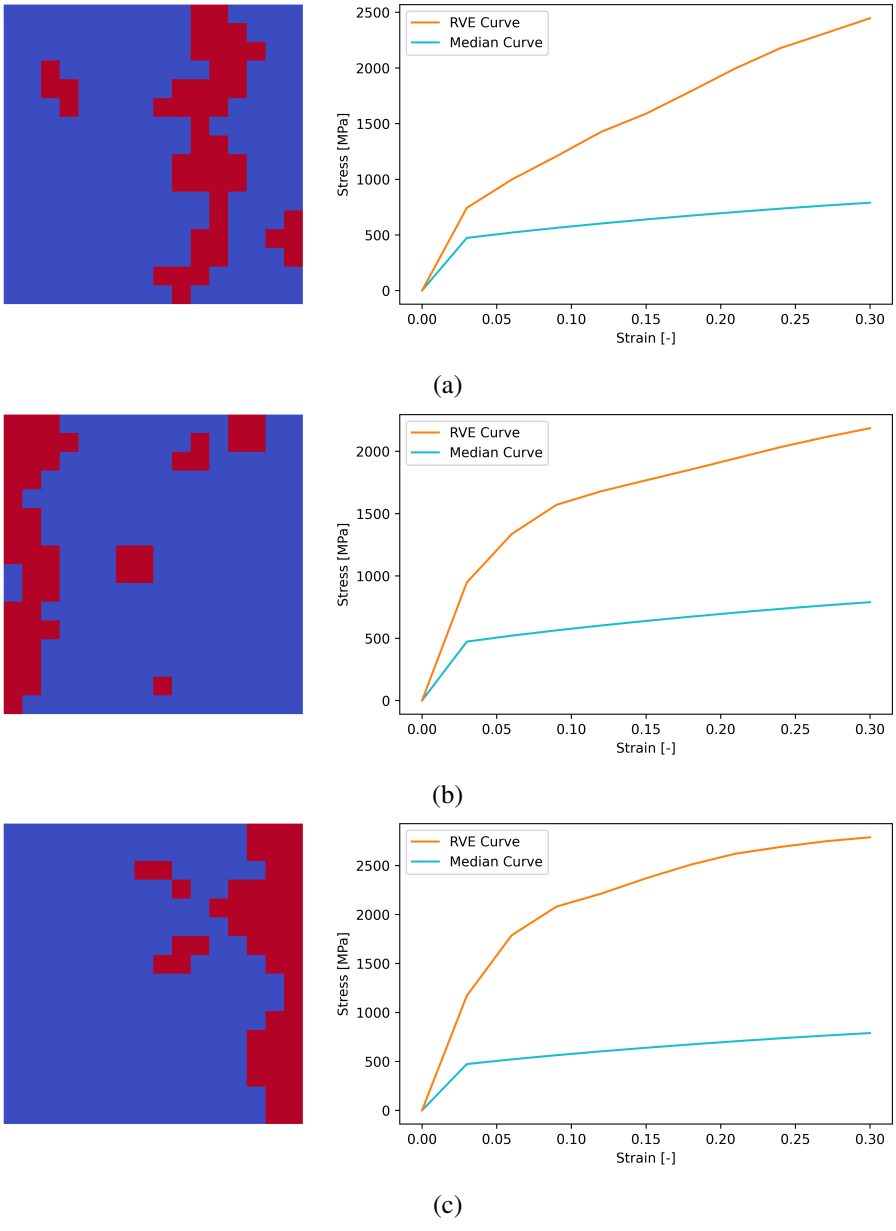


(c)

Figure 3.7: RVEs with horizontal fiber-like formations.

If the rigid elements would align perpendicular to the loading axis, the stresses would still be vastly superior for any given strain than the median, but the same parallel as before doesn't apply, as a composite shouldn't be loaded perpendicular to all its fibers. The explanation for this behavior lies in the Poisson effect, that is, as the RVE elongates in the loading direction, it must

contract in other directions to compensate. If the material isn't allowed to freely constrict along the other directions, it would result in a considerable increase in stress for any given strain in the same direction as the load is applied. This is precisely what happens in these scenarios, as the rigid elements are under considerable vertical stress for any given horizontal homogenized strain, acting as a rigid beam under compression, impeding the RVE from thinning. This phenomenon is illustrated in Figure 3.9. Some curves for these RVEs are presented on Figure 3.8.



(a)

(b)

(c)

Figure 3.8: RVEs with vertical beam-like formations.

Worth noting that these could not be considered statistical representations of the material studied, as the likelihood of the entire hard phase being aligned along an entire component is close to none. As such, in a rigorous physical model, these should be discarded. However, as the focus

of this work is to determine the capability of the machine learning models to adapt to the various hard phase arrangements and how these affect the overall curve, these were kept in, to study if the models could fit these rare cases.

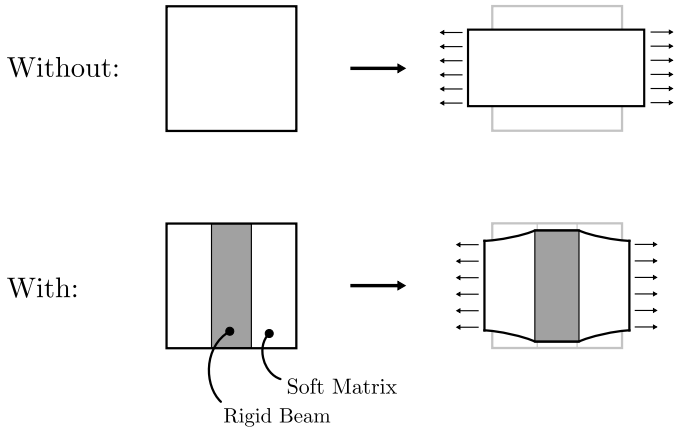
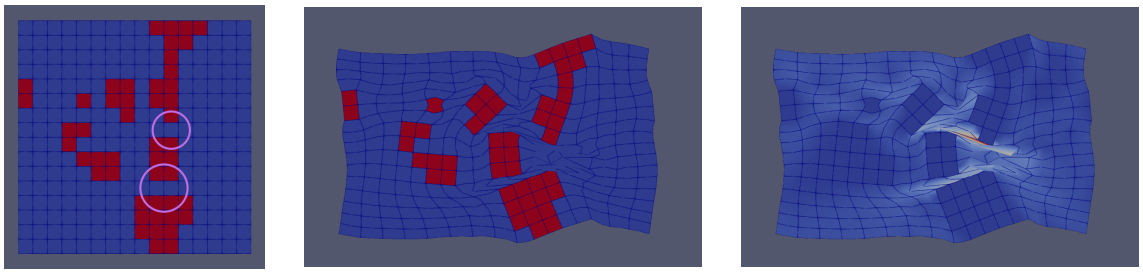


Figure 3.9: Rigid Beam perpendicular to the loading direction in a Soft Matrix.

3.5.2.2 Crushed Elements

Most RVEs have predictable behavior when under loading, resulting in fairly regular traction curves. However, some samples from the generated dataset stood out as they had peculiar appearance. Some of these, such as the one seen on Figure 3.10, after analyzing the mesh in Paraview, it was possible to identify a soft element that, during traction, resulted in crushing of said element. This would then be so deformed in such a manner that it shouldn't even be considered a physical representation of actual material.



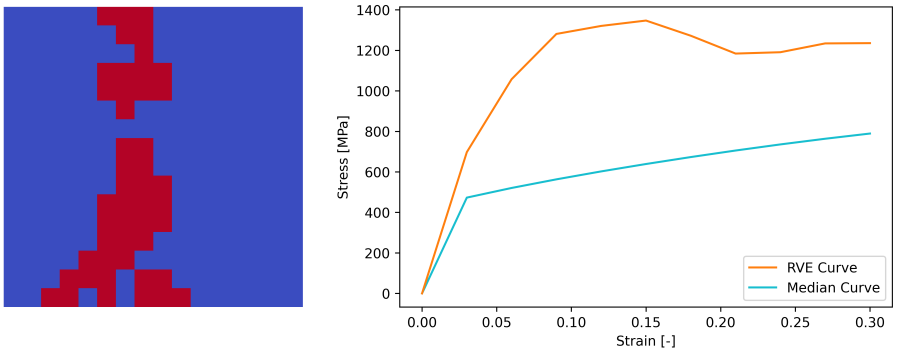
(a) Undeformed (b) Deformed (c) Von Misses Stress

Figure 3.10: Crushed Elements, highlighted in (a).

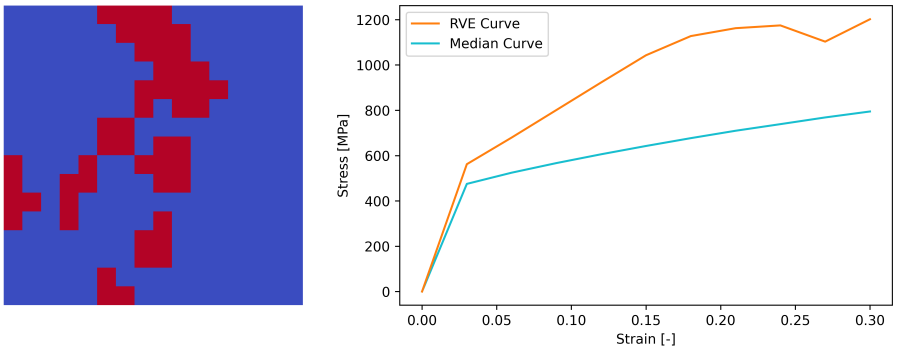
Arguments can be made that these samples should be removed from the dataset, as they will just be noise and not actually contribute to model accuracy. However, it wasn't possible in the timeframe available to code a filtration algorithm that could identify and remove these entries without endangering useful data. The first method that came to mind was to remove any curve that had a dip in stress along it, as this was unexpected behavior but common throughout all these RVEs. This would have a fairly straight forward execution, yet there was no guarantee that this drop in stress wouldn't happen in any accurate sample. If it did, important data could be removed

from the dataset, so this method was discarded. Another approach would be to eliminate any mesh that had element that was deformed beyond a specific threshold, but this raises the question of where to trace said threshold. It would also have been very coding-intensive for an apparently small percentage of the dataset and with way too little time for implementation, with no guarantees of success. Obviously, due to dataset size, manual filtration was not an option.

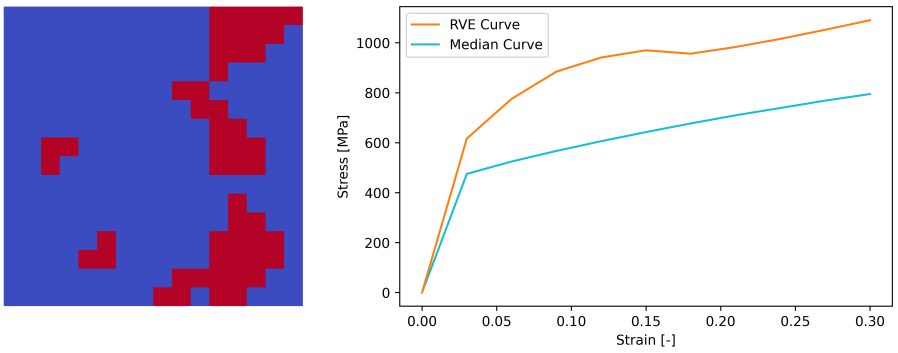
It was not possible to estimate how many of *crushed RVEs* are still part of the dataset, but as these were first discovered close to the end of framework development, after manually analyzing many curves and model predictions, it is safe to assume that these are few and far in between. They shouldn't introduce much noise into model training, yet is something that should be tackled in the future.



(a)



(b)



(c)

Figure 3.11: RVEs with crushed elements.

3.6 Substitutes to Neper

When creating a dataset this massive, a significant concert is the speed of data generation. The process itself must be as optimized as possible, as a sight delay, over thousands of cycles, could save days, or even weeks of computing time.

During the early stages of this project, each RVE could take anywhere from forty seconds, up to a minute of processing. This, obviously, wasn't doable when considered the side of the intended dataset. For this very reason, the process had to be as optimized as possible.

The most time-consuming step in the entire process was tessellation, as this could take up to thirty seconds per RVE, that then had to be meshed. The second most time-consuming step was the FE^2 analysis, but there was no way to circumvent that, as it was the core of the entire analysis. Yet, tessellation seemed as a big waste, as it spent thousands of iterations creating a complex microstructure, that would then be simplified into a 16×16 grid. For this reason, two attempts were made to create Neper-like meshes, to accelerate data generation.

3.6.1 Random RVEs

The most immediate concept for generating a mesh was to simply do a random distribution of rigid elements in the soft matrix. This seemed as a simple, fast and easy to implement way to create new RVEs.

However, this was not as straight forward as it seems. Whilst creating a 16×16 zeros matrix with a random distribution of around 20% "1"s was very simple program-wise, for it to be analyzed by Link2, a `.rve` file had to be created. This was not a simple task, as these files are fairly complex, with a lot of information regarding mesh and material properties. And this had to be done exactly like Neper, otherwise the FE^2 software would not recognize it. This was eventually accomplished by breaking apart a Neper-made `.rve` file, and, bit by bit, reassembling it, but replacing the Material IDs of each element with the corresponding one from the random matrix.

On Figure 3.12 are presented some RVEs generated by this method, and in Figure 3.13 is plotted the curves of 5 000 of these meshes.

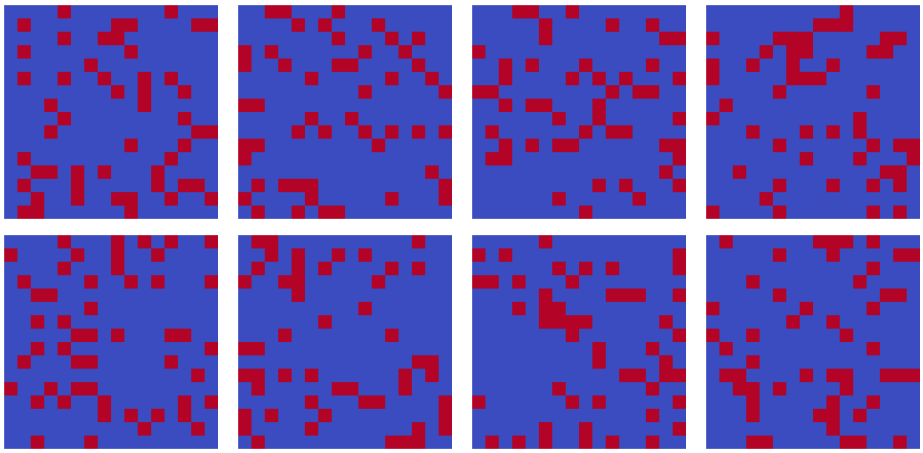
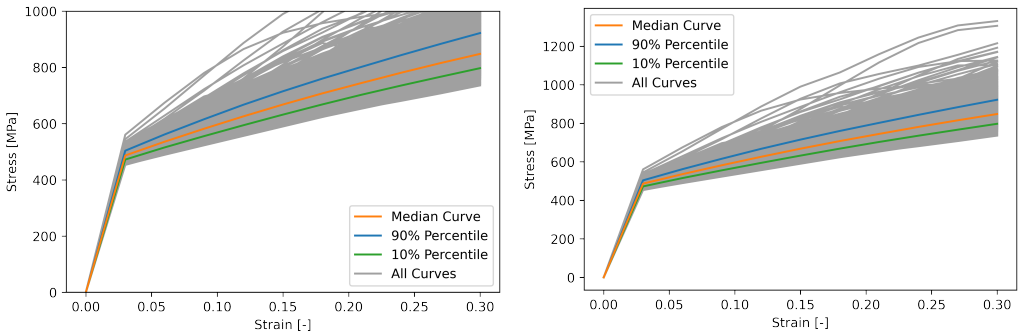


Figure 3.12: Randomly generated RVEs.

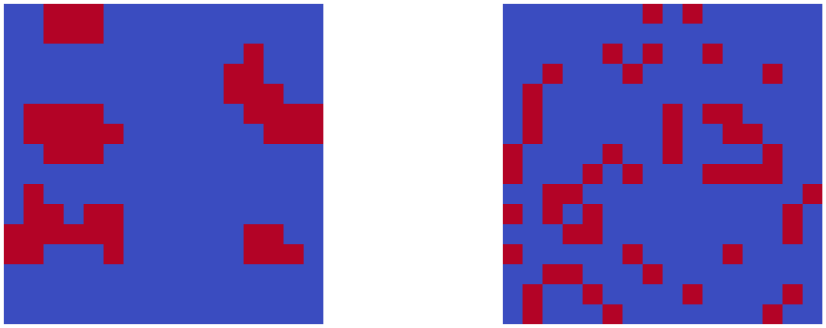


(a) Focused on median curve

(b) Global view

Figure 3.13: Plot of 5 000 curves from randomly generated RVEs.

This process ended up discarded as the RVEs generated did not resemble the ones created by Neper (Figure 3.14). The curves generated by it were also too regular when compared to the ones from Figure 3.4, as both the 10% and 90% percentiles where at similar distances from the median curve, when compared to Figure 3.4a, with the 90% curve being significantly higher than in the randomly generated ones.



(a) Neper Generated (b) Randomly Generated

Figure 3.14: Comparing Neper and randomly generated RVEs.

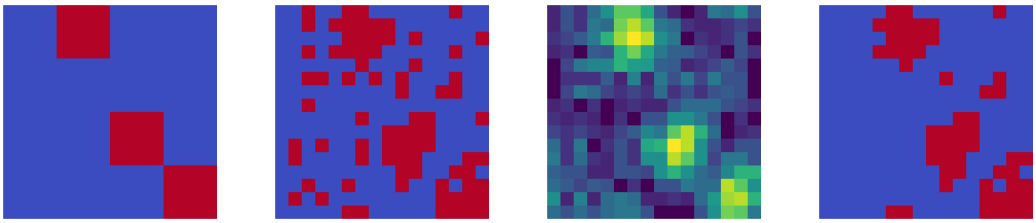
3.6.2 Cluster-based, Pseudo-Random Distribution

The second method developed was conceived with the biggest problem of the first in mind, that is, solve the lack of clustering of hard elements. The main idea was to use an initial support matrix to establish varying probabilities of each element being of the hard phase, depending on location. With this, create the first iteration of the RVE, and to modify it based on the density of hard elements of that region. This would cause the elements of the same type to stick together, solving the absence of these groups.

The support matrix would be a 4×4, and each element would have 20% chance of being of the hard phase (Figure 3.15a). Then, a 16×16 matrix would be created. Each element had an 80% chance of matching their overlapping 4×4 element (Figure 3.15b). The following step would be to calculate a weight associated with each element. This weight would determine the likelihood of a certain element changing phase. It was calculated for a given element with coordinates i, j by:

$$z_{i,j} = \sum_{n=1}^{8} e_n / d_n, \tag{3.2}$$

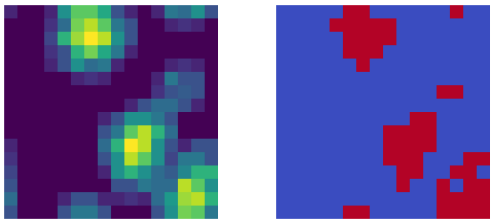
where n scrolls through all 8 surrounding elements, e_n is the value for that element (0 for soft, 1 for hard) and d_n would be the distance between the center of i, j and n (1 for the elements immediately beside, $\sqrt{2}$ for the ones in the diagonal). This would result in a map such as the one in Figure 3.15c. After that, if an element's weight was over or under certain thresholds, it would change phases (Figure 3.15d). This would remove standalone elements, and regulate slightly the rim of the hard clusters.



(a) Initial 4×4 (b) Initial 16×16 (c) First Weights (d) First Corrected

Figure 3.15: First stage in pseudo-random RVE generation.

The last step would be to recalculate $z_{i,j}$ and refine the percentage of each phase using the weights as guides, flipping the ones in the extremities (Figure 3.16).



(a) Second Weights (b) Final RVE

Figure 3.16: Last stage in pseudo-random RVE generation.

An advantage of this process is that it is possible to create many slightly different RVEs from the same initial 4×4 (Figure 3.17).

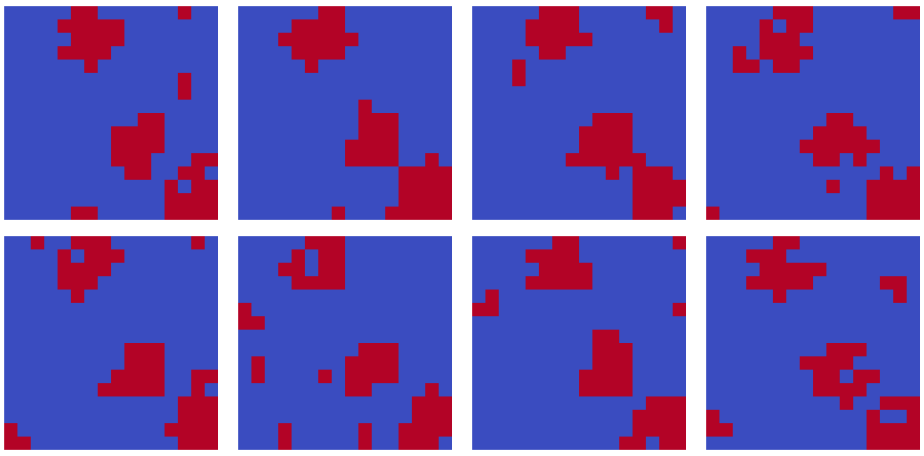


Figure 3.17: Multiple RVEs from the same 4×4.

These microstructures starting to look more like the ones outputted from Neper, but it was still possible to easily tell them apart. The next iteration of this methodology was intended to have some randomness added to the weight calculation, allowing some occasional small clusters to not be completely removed. However, this generation algorithm was never finished. A change

in equipment and the subsequent optimizations made Neper a viable option, discarding the need for these alternatives.

The best takeaway from these studies into mesh generation was the methodology to create a `.rve` file for a Python generated mesh. This would be useful later on for data augmentation, and sparked the idea for one of the system used.

Chapter 4

Model Selection

4.1 Establishing Input Shape and First Model

As introduced on Section 2.4, neural networks for machine learning can have multiple architectures, each one more appropriate for each intended application. The first step in designing one is choosing which of these architectures is the best to perform the analysis intended. For this, it's important to specify both inputs and outputs intended.

Input - Microstructure's Mesh

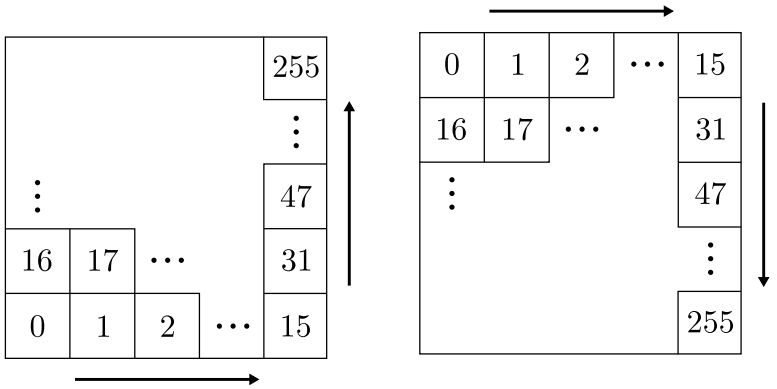
As explained previously, this was obtained from the `.rve` file, and contains the Material ID for each element, in a sequential order. This is stored in the dataframe as a 256-integer array. For each array element, corresponds a mesh element (Figure 4.1, complete maps in Appendix D), and can either take values 0 or 1:

- 0 represents soft phase;
- 1 represents hard phase.

However, depending on which architecture is chosen, it might be needed as a 16×16 matrix, resembling the mesh itself. Worth noting that, for the sake of coding simplicity, this representation is a mirrored version of the initial mesh along an axis following the loading direction. This mirroring does not void the FE^2 analysis on the RVE, as it still represents the same microstructure under the same strains. This is illustrated by Figures 4.1 and 4.2.

Output - Stress-Strain Curve

This was read from the `.out` file, and is an 11 element array containing the stresses at all 10 increments along the curve, as well as it's initial state ($\sigma = 0$ MPa). The curve does not need to be represented using both stresses and strains as the strain points are the same for all RVEs, defined by Link2 and filtered by the Algorithm 2. For all curves, the stress for $i = 0$ is always 0, as it represents the initial step.



(a) LINKS2 (b) Python Code

Figure 4.1: Mesh as a matrix and respective sequence in Links2 and used in the Python Code.

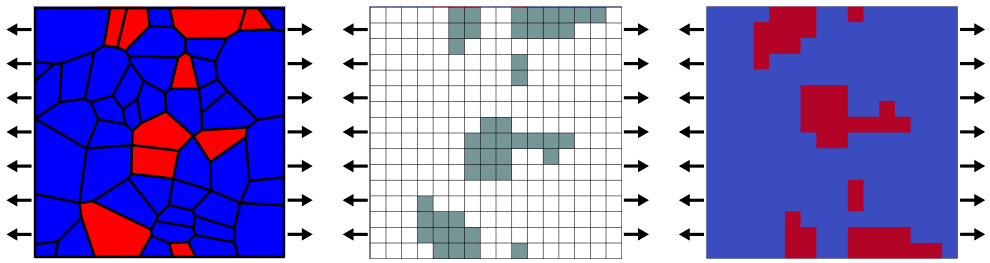
Once established both inputs and outputs, the following step consists of selecting an initial architecture. For this application, the ones worth analyzing are Fully Connected Neural Networks and Convolutional Neural Networks (CNN). Recalling Section 2.4, Fully Connected Neural Networks form the basis of neural networks, and are the basic their most implementation. CNNs are mostly tailored for image processing and classification, used in face recognition.

For this problem, at first, it makes much more sense to use CNNs, as our mesh can be viewed as an image, and the homogenized properties are dependent of the interaction and shapes formed by neighboring elements, whose relation is better captured by CNN models. Yet, these two different architectures where tested, using very simple models, for comparison purpose. For this test, a sample of 10% of the original dataset was prepared, consisting of 10 000 RVEs. It used for K-Fold Cross-Validation (CV) with 10 folds. Worth noting, these 10 Folds where the same for both models. This allows direct comparison between both models at each fold.

4.1.1 Simple Fully Connected Model

This model is a basic application of a fully connected architecture, and has its structure detailed in Model 1. It is composed of alternating Batch Normalization layers and three Dense Layers, being those smaller along the model, having 126, 64 and 32 nodes each. The Batch Normalization was applied as it helps stabilize training[21][22]. This results in a very simple model, but representative of the capacity of this architecture to extract the stress-strain curve out of the mesh.

The loss function selected for this analysis was the Huber Loss, as it is less sensitive to outliers in the data than the more commonly used MSE. This was chosen to minimize the effect of similar samples as the particular RVEs studied in Section 3.5.2. This lost function was kept the same for the majority of the models tested, and whenever not stated otherwise, the loss values mentioned will be referring to test loss.



(a) Microstructure (b) Mesh (c) Python matrix

Figure 4.2: Loaded microstructure and respective Mesh and Python matrix representation.

Model 1: FullCon - Simple Fully Connected Model

Input: Mesh (Shape: 256 integer array)

$1\times$

Dense Layer:
Number of Nodes: 128
Batch Normalization Layer
Dense Layer:
Number of Nodes: 64
Batch Normalization Layer
Dense Layer:
Number of Nodes: 32
Batch Normalization Layer
Dense Layer:
Number of Nodes: 11
Batch Normalization Layer

Output: Curve (Shape: 11 float array)

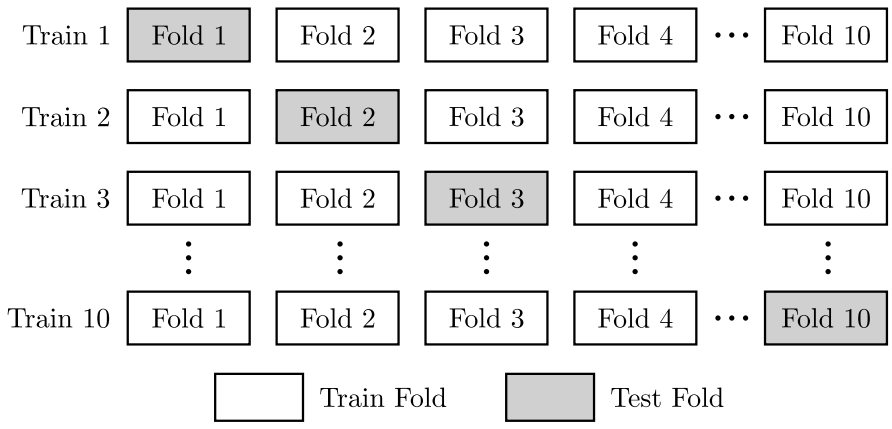


Figure 4.3: Schematic for K-Fold Cross Validation.

Model Evaluation

On Table 4.1 are presented the test losses for the ten trained SimpleFNN models, as well as the percentages of test samples with MAPE bellow 10% and 5%. From it, is possible to conclude that the model is fairly capable of predicting the behavior of the microstructure under stress, as for 49.1% of the test RVEs it predicted with a MAPE of 5% or better, being this a fairly decent estimate.

Table 4.1: K-Fold Cross Validation Results for FullCon.

	Huber Loss	MAPE Bellow 10%	MAPE Bellow 5%
Fold 1	45.14	79.8%	48.9%
Fold 2	43.47	83.1%	48.8%
Fold 3	42.16	83.1%	47.8%
Fold 4	42.53	82.9%	49.0%
Fold 5	41.07	82.3%	44.6%
Fold 6	38.81	81.8%	52.0%
Fold 7	42.56	85.1%	52.5%
Fold 8	36.04	83.9%	53.9%
Fold 9	40.46	81.2%	49.6%
Fold 10	42.41	81.2%	49.1%
Average	41.46	82.44%	49.6%

4.1.2 Simple CNN Model

Similar to the previously defined FullCon Model, this architecture is a simple CNN Model, solely to verify the capabilities of this architecture to predict the curve. This architecture was designed to have a similar number of trainable parameters to the previous one, so that they could be somewhat comparable.

The model chosen consists of 2 different blocks: the first had a sequence of 2D Convolutional Layers and Max Pooling Layers, and is repeated twice; the second block is composed of simply two dense layers of 64 and 11 nodes each, being the last one the output layer, as per detailed in Model 2.

Model 2: Simple CNN Model

Input: Mesh (Shape: 16×16 integer matrix)

2×
- **2D Convolution Layer:**
  - Number of Kernels: 4
  - Kernel Size: 3
  - Activation: ReLu
  - Stride: (1,1)
- **Batch Normalization Layer**
- **Max Pooling 2D Layer:**
  - Filter Size: (2,2)
  - Stride: (1,1)

1×
- **Dense Layer:**
  - Number of Nodes: 64
- **Batch Normalization Layer**
- **Dense Layer:**
  - Number of Nodes: 11

Output: Curve (Shape: 11 float array)

Model Evaluation

On Table 4.2 are presented the losses for the model trained for the 10 Folds.

Table 4.2: Losses and Percentages of Test RVEs with MAPE bellow 10% and 5% for SimpleCNN.

	Huber Loss	MAPE Bellow 10%	MAPE Bellow 5%
Fold 1	42.59	80.2%	48.6%
Fold 2	40.40	83.9%	54.4%
Fold 3	38.62	85.6%	53.7%
Fold 4	40.15	84.4%	53.7%
Fold 5	37.87	84.5%	53.3%
Fold 6	35.37	84.8%	57.8%
Fold 7	41.21	85.4%	54.9%
Fold 8	35.61	84.1%	54.6%
Fold 9	37.61	83.7%	51.5%
Fold 10	38.76	84.7%	55.8%
Average	38.82	84.1%	53.8%

On Table 4.3 and Figure 4.4 are represented the losses for each Fold by both FullCon and SimpleCNN Models. With the exception of Fold 8, where both models have very similar performance, SimpleCNN proved consistently better than FullCon. This demonstrated that considering the mesh as a 16×16 matrix and analyzing it as an image would result in better model performance, so this

Table 4.3: Comparisson of Losses for FullCon and SimpleCNN.

	FullCon	SimpleCNN	Improvement
Fold 1	45.14	42.59	5.65%
Fold 2	43.47	40.40	7.06%
Fold 3	42.16	38.62	8.40%
Fold 4	42.53	40.15	5.60%
Fold 5	41.07	37.87	7.79%
Fold 6	38.81	35.37	8.86%
Fold 7	42.56	41.21	3.17%
Fold 8	36.04	35.61	1.19%
Fold 9	40.46	37.61	7.04%
Fold 10	42.41	38.76	8.61%
Average	41.46	38.82	6.34%

architecture class was chosen moving forward. However, it shall be pointed out that there is still a small difference in the number of trainable parameters between the models. This results in a slight bias in the comparison, but this should not be significant enough to invalidate the conclusion from it.

Another takeaway from this comparison, especially evident in Figure 4.4, is the variance of performance between folds for both models. This can be seen as both have similar variations in performance, compared to the mean of all folds, in each one of them. For instance, both models performed terribly on Fold 1, with about average performance on Fold 5 and great results on Fold 6. This might indicate an imbalance when folding, resulting in some having easier to predict curves than the others. This may happen as some of them have a higher concentration of outliers than the others. Even so, this does not invalidate this analysis, as the resolution for which model is the best it's still very clear. However, this might indicate a need to more carefully generate the folds going forward, using Stratified K-Fold Cross Validation.

4.2 Model Design

Once the model type is chosen, it is necessary to go deeper into the model design. A neural networks' performance is hard to estimate without the necessary testing, as such, its respective layer types, quantities and respective parameters, all need to be chosen and tested.

During this process, the model went through many iterations, with changes to its layers types and arrangements. This was the most time-consuming step in the creation of the neural network, as it required constant supervision and research, with every tested change requiring a significant amount of processing time. This was also done in parallel to the dataset generation, with an increasing dataset along the process itself. The models where always tested using CV, keeping the same K-Folds across comparisons, to eliminate the influence caused by the variations in the test set.

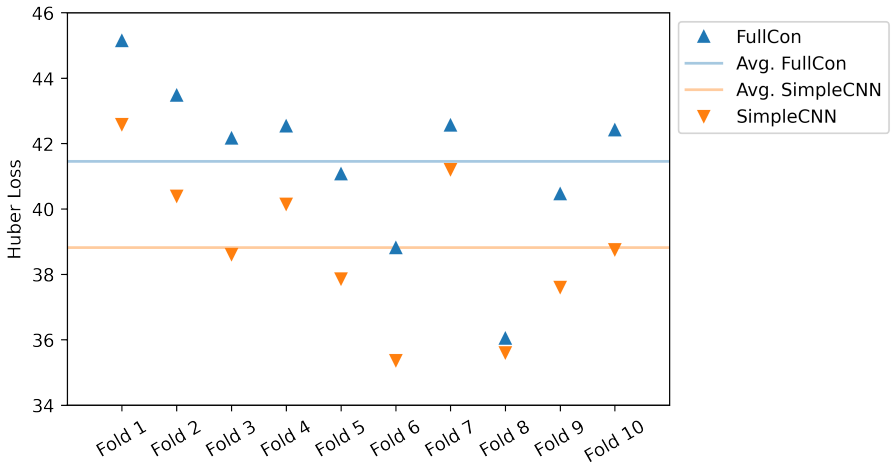


Figure 4.4: Losses across Folds for FullCon and SimpleCNN.

In this section, some architectures are presented. These will demonstrate key decisions taken during model design, justifying the layers chosen in the final neural network. Worth noting that these architectures were all re-run using Stratified K-Fold CV for the entire 100 000 RVE dataset, in order to allow a direct comparison between each version. All other aspects regarding model layout were equalized, to isolate each layer's impact on accuracy. Batch size was kept at 16 up until tuning, but learning rate was changed between iterations.

For the stratification process, the chosen criteria was the area under the curve, as this proven useful in the identification of the outliers in the dataset previously. As these outliers are expected to be the ones where the model would have the worst performance, it made sense to guarantee that they were evenly spread across all folds. This value was then used in an intermediate classification into multiple classes, each of them containing the same number of RVEs as the number of Folds, distribution them randomly across all folds. An illustration of this process is shown in Figure 4.5, and the step by step is detailed in Algorithm 3.

Algorithm 3 Code for Stratified K-Fold Cross-Validation.

$df \leftarrow$ dataframe with 100 000 RVEs
Creates empty column *Category* in df
$percent \leftarrow$ array with percentages ▷ Delimit the percentiles
$percentile \leftarrow$ array that links RVE to respective percentile
$Category \leftarrow percentile$
$skf \leftarrow$ StratifiedKFold function from the scikit-learn package
for each *train* and *test* in $skf.split$ **do** ▷ For each Fold
    $TRAINdf \leftarrow df[train]$ ▷ Creates Train Dataframe
    $TESTdf \leftarrow df[test]$ ▷ Creates Test Dataframe
    Save $TRAINdf$ to `.pkl` file
    Save $TESTdf$ to `.pkl` file
end for

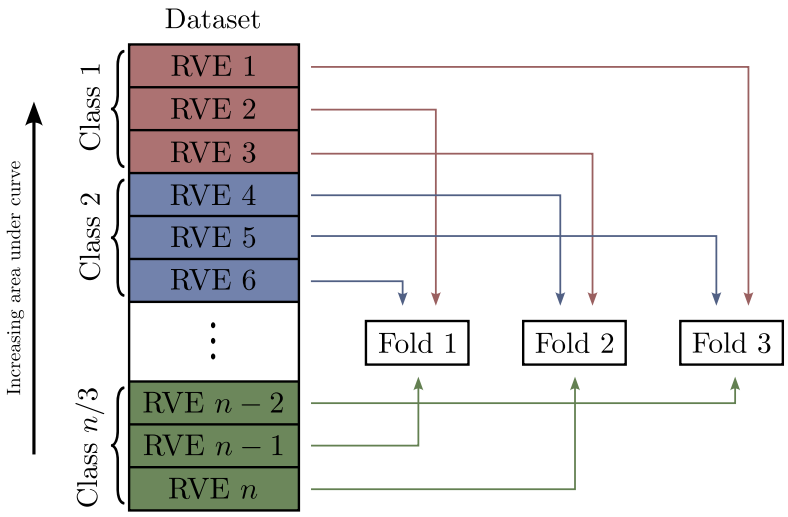


Figure 4.5: Illustration of a Stratified K-Fold Cross-Validation for 3 Folds.

4.2.1 Increasing Depth

The first obvious step to, usually, improve model performance is to increase its depth, that is, increase the amount of hidden layers. For this purpose, and based on SimpleCNN (Model 2), a new block was added, consisting solely of 2D Convolutional Layers. This would allow for some pre-processing of the images before the Max Pooling process, as this can only be repeated a certain number of times, as it reduces the matrices dimensions.

The rest of the model was also deepened, as the number of kernels per Convolutional Layer was increased and the Dense Block that caps the process was also expanded. This model is detailed in Model 3.

This deepening should allow the neural network to better capture details in the microstructure, resulting in a finer analysis. However, this increase in the number of trainable parameters may cause overfitting, reducing the accuracy of the model, as is verified later on.

Model Evaluation

On Table 4.4 are presented the losses of the model for the Stratified K-Fold CV. It should be mentioned that, even though this model has vastly outperformed the SimpleCNN, as this training had an increased dataset, from 10 000 to 100 000 RVEs, this improvement in performance is partially due to this change. As such, these results cannot be directly compared to the ones obtained previously.

Besides this, the effect of the Stratified K-Fold CV is still apparent, specially on Figure 4.6, which plots the losses for all folds, and it's mean value. The decrease in variation of performance during testing is obvious, especially when compared to relation between SimpleFNN and SimpleCNN to the respective folds (Figure 4.4).

Worth noting the significant increase in the percentage of RVEs with MAPE bellow 10% and 5%, due to both the deepening of the neural network and the augmented dataset. As these values start being very similar between models, from now on, also the percentage of test RVEs with MAPE bellow 2.5% is also provided, as a new metric to help visualize the AIs performance.

Model 3: DeeperCNN - Deeper CNN Model

Input: Mesh (Shape: 16×16 integer matrix)

2×	**2D Convolution Layer:** Number of Kernels: 8 Kernel Size: 3 Activation: ReLu Stride: (1,1) **Batch Normalization Layer**
4×	**2D Convolution Layer:** Number of Kernels: 8 Kernel Size: 3 Activation: ReLu Stride: (1,1) **Batch Normalization Layer** **Max Pooling 2D Layer:** Filter Size: (2,2) Stride: (1,1)
1×	**Flatten Layer**
2×	**Dense Layer:** Number of Nodes: 64 **Batch Normalization Layer**
1×	**Dense Layer:** Number of Nodes: 11

Output: Curve (Shape: 11 float array)

Table 4.4: Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for DeeperCNN.

	Huber Loss	MAPE Bellow 10%	MAPE Bellow 5%	MAPE Bellow 2.5%
Fold 1	22.21	96.8%	82.7%	49.4%
Fold 2	23.42	96.3%	81.3%	45.9%
Fold 3	22.41	97.5%	80.9%	45.2%
Fold 4	22.55	97.0%	82.3%	47.3%
Fold 5	22.24	97.3%	81.7%	47.6%
Fold 6	22.60	96.7%	81.9%	48.1%
Fold 7	22.86	97.0%	80.9%	46.2%
Fold 8	22.80	97.2%	81.2%	43.9%
Fold 9	23.00	96.3%	80.7%	47.3%
Fold 10	23.20	96.1%	81.0%	48.4%
Average	22.73	96.8%	81.5%	46.9%

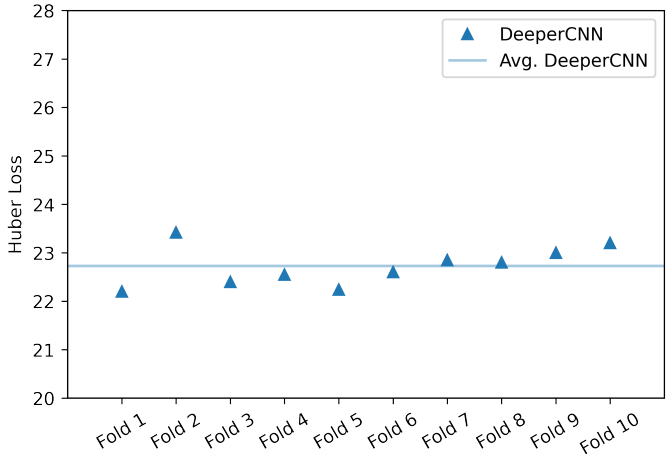


Figure 4.6: Losses across Folds for DeeperCNN.

4.2.2 Purely Convolutional CNN Model

Whist most CNN networks have some form of dense block in the end of its coevolutionary and pooling layers, in some applications it may be advantageous to eliminate this end hidden section, leaving solely a unique dense layer to serve as output for the neural network. Inspired by the work of Yang [23] on the prediction of mechanical properties of composite RVEs using Convolutional Neural Networks, a variation of DeeperCNN (Model 3) without this final block was tested.

This new architecture (Model 4) is similar of the one described previously, but eliminating the block of dense Layers in the end, keeping solely the one needed as the AI's output, an 11-node final dense layer.

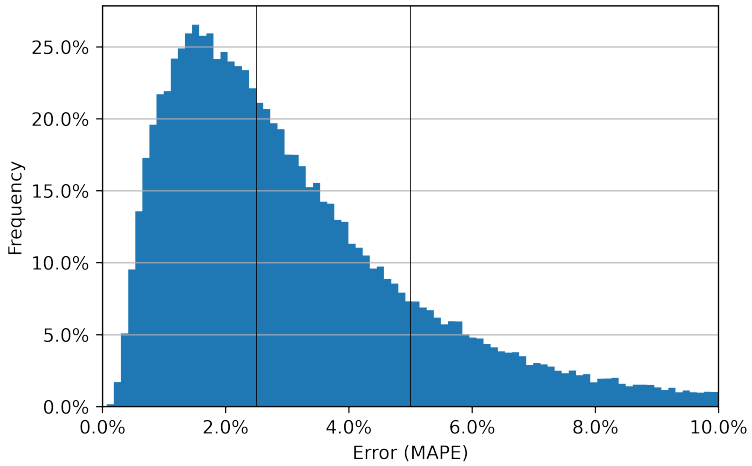


Figure 4.7: Relative Histogram of the MAPE for the Test of DeeperCNN.

Model 4: Purely Convolutional Model

Input: Mesh (Shape: 16×16 integer matrix)

2×	**2D Convolution Layer:** Number of Kernels: 8 Kernel Size: 3 Activation: ReLu Stride: (1,1) **Batch Normalization Layer**
4×	**2D Convolution Layer:** Number of Kernels: 8 Kernel Size: 3 Activation: ReLu Stride: (1,1) **Batch Normalization Layer** **Max Pooling 2D Layer:** Filter Size: (2,2) Stride: (1,1)
1×	**Flatten Layer** **Dense Layer:** Number of Nodes: 11

Output: Curve (Shape: 11 float array)

Worth noting that this model is not expected to vastly outperform the previous one. The intent of this analysis is to evaluate the influence of the dense block on the models' precision, and if said

influence is worth the added computing costs these layers require. If these models have similar performance, it may be worth to remove said end section.

Model Evaluation

The results of the training and testing of PureCNN using the Stratified K-Fold CV described previously are presented on Table 4.5. These folds remain the exact same as the ones used for DeeperCNN, allowing a direct comparison between both models' performance for each individual fold, besides their overall accuracy.

Table 4.5: Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for PureCNN.

	Huber Loss	MAPE Bellow 10%	MAPE Bellow 5%	MAPE Bellow 2.5%
Fold 1	24.23	96.1%	79.1%	42.9%
Fold 2	23.80	96.9%	79.0%	42.7%
Fold 3	26.63	95.3%	75.1%	36.4%
Fold 4	26.60	95.7%	76.6%	36.9%
Fold 5	25.01	96.1%	77.5%	39.7%
Fold 6	24.75	96.5%	78.2%	38.3%
Fold 7	26.22	95.8%	74.6%	37.5%
Fold 8	25.81	95.5%	75.8%	38.6%
Fold 9	23.70	96.7%	78.6%	43.4%
Fold 10	24.67	96.0%	77.7%	42.7%
Average	25.14	96.0%	77.2%	39.9%

Both models' testing Huber loss per fold are plotted on Figure 4.8, together with their mean across all tests. Is easily seen that the end block proved very valuable for the models' accuracy, as PureCNN did not surpass the DeeperCNN in any fold, resulting in an average loss considerably higher that the previous architecture.

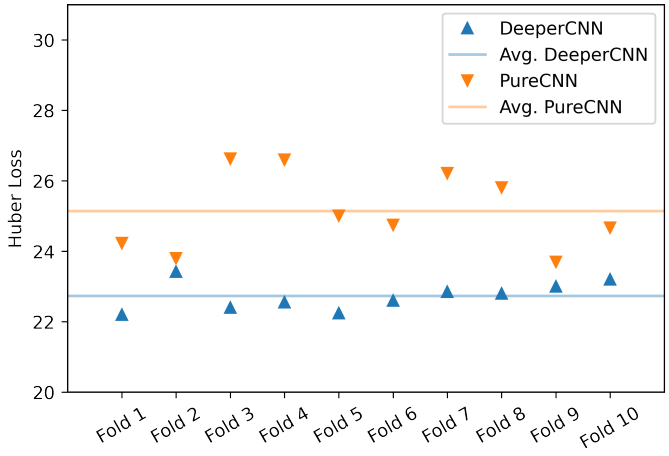


Figure 4.8: Huber losses across Folds for PureCNN.

This apparently small difference in Huber loss comes with a considerable decrease in percentages of RVEs with MAPE bellow 5% and 2.5%, with little impact on 10% (Table 4.6). This shows that even such a small variation in loss translates to a significant discrepancy in actual results, specially for lower MAPEs.

Table 4.6: Average Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for PureCNN.

	Huber Loss	MAPE b. 10%	MAPE b. 5%	MAPE b. 2.5%
DeeperCNN	22.73	96.8%	81.5%	46.9%
PureCNN	25.14	96.0%	77.2%	39.9%
Difference	+2.41(10.6%)	-0.8%	-4.3%	-7.0%

Another conclusion from this comparison is the high variability in losses between folds for PureCNN, evident on Figure 4.8. This reveals an unexpected benefit from the dense block, as this post-processing of the flatten matrix helped with stabilizing the architecture's testing performance.

Ultimately, it's easy to understand that PureCNN did not outperform, or even matched the previous results. The model was less stable during testing and had losses considerably worst then DeeperCNN, resulting in a decrease in the accuracy of the predicted curves. As such, this change was discarded, returning the dense block to the end of the architecture.

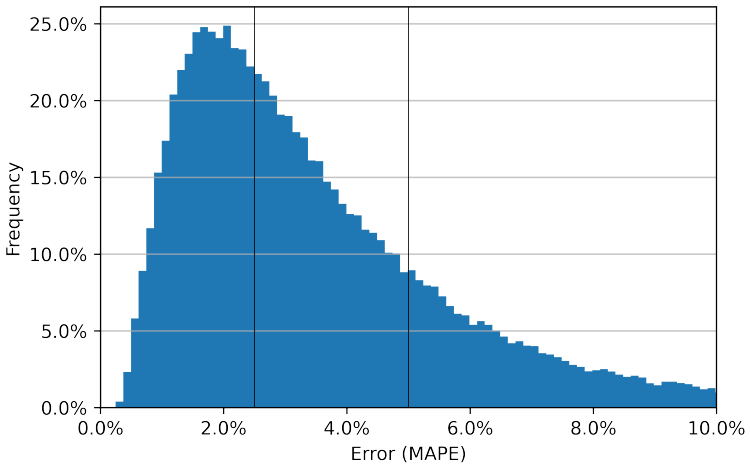
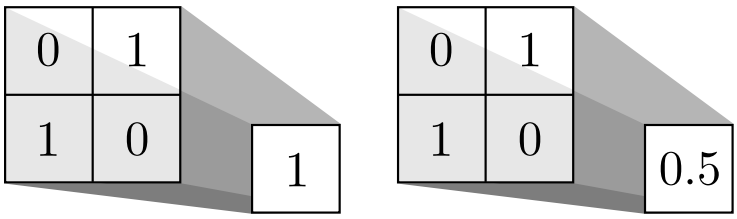


Figure 4.9: Relative Histogram of the MAPE for the Test of PureCNN.

4.2.3 Implementation of Average Pooling

As explained in Section 2.4, some of the most important layers in a Convolutional Neural Network are the Pooling layers, being those responsible for both reducing the matrices dimension, as well as helping to identify the characteristics of the image that may relate to it's output. These layers are usually intercalated with the convolutional layers, containing the kernels, responsible for most image processing, and usually, the last one, outputs into the flatten layer.

In most CNN for image classification and processing, the pooling process chosen is the max pooling, illustrated in Figure 4.10a. However, an alternative to max pooling is the average pooling, which works in similar ways to the previously described algorithm, being the key difference that the output element is calculated based on the average of the values inside the filter, instead of it's maximum (Figure 4.10b). This small difference may not seem very relevant in an isolated filter, but when applied to a full matrix, or even a 3×3, the differences in the output are very prominent. On Figure 4.11 are presented three different input matrices, alongside their graphic representation, where its respective values influences the elements color intensity, being 1 its maximum. Each of these matrices was then run through both max and average pooling, and their respective outputs are presented in similar fashion. In this set of examples, all distinct input matrices have equal max pooling outputs. However, all average pooling outputs are slightly different, indicating a higher loss of information by max pooling. This might prove valuable, as there is a lot of minutiae in each RVE, and average pooling might be able to better interpret these small details and their relation to the final output, that would, otherwise, be lost.



(a) Max Pooling (b) Average Pooling

Figure 4.10: Representation of both pooling strategies.

Another alternative to max and average pooling not explored in this work is factional max pooling, that allows a higher number of pooling layers. [24]

For this analysis, a model built based on DeeperCNN (Model 3) was prepared, called AvgPoolCNN (Model 5), with the only modification to its architecture being the change of max pooling to average pooling in the intermediate block, keeping the rest of the layout. Both filter size and stride where kept the same, at 2×2 and 1 respectively.

Model Evaluation

This revision presented a considerable breakthrough during model development, as seen on Figure 4.12, which compares the losses per fold for AvgPoolCNN (Tables 4.7 and 4.8) with the previous architectures. This improvement in accuracy may well be due to the small details in the RVE that the previous models had trouble identifying and analyzing, but better captured by this simple change.

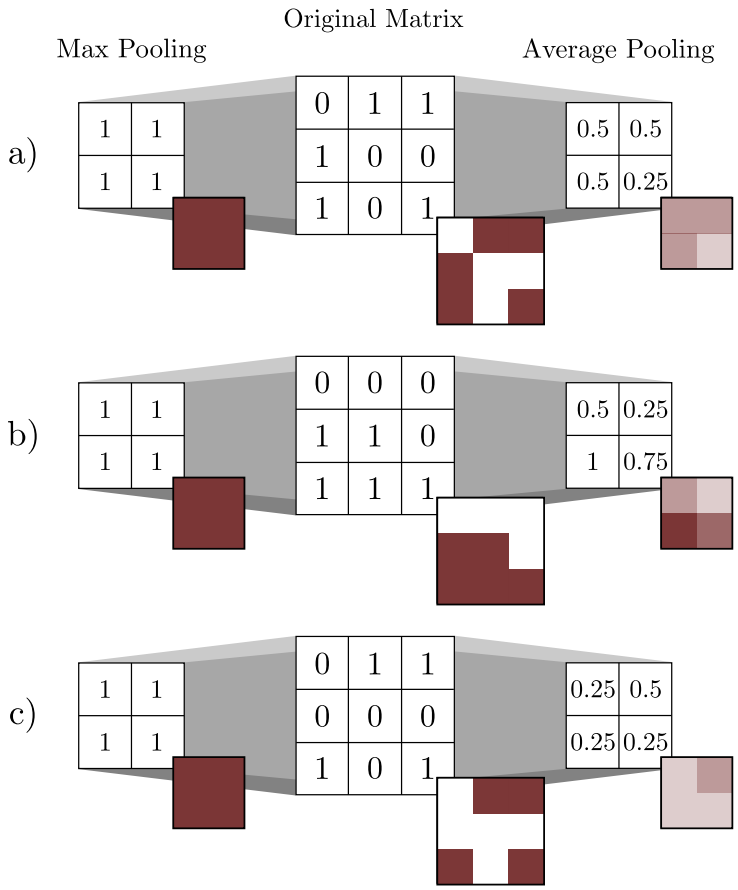


Figure 4.11: Comparison of loss of information between Max Pooling and Average Pooling.

Model 5: AvgPoolCNN - CNN with Average Pooling

Input: Mesh (Shape: 16×16 integer matrix)

2×	**2D Convolution Layer:** Number of Kernels: 8 Kernel Size: 3 Activation: ReLu Stride: (1,1) **Batch Normalization Layer**
4×	**2D Convolution Layer:** Number of Kernels: 8 Kernel Size: 3 Activation: ReLu Stride: (1,1) **Batch Normalization Layer** **Average Pooling 2D Layer:** Filter Size: (2,2) Stride: (1,1)
1×	**Flatten Layer**
2×	**Dense Layer:** Number of Nodes: 64 **Batch Normalization Layer**
1×	**Dense Layer:** Number of Nodes: 11

Output: Curve (Shape: 11 float array)

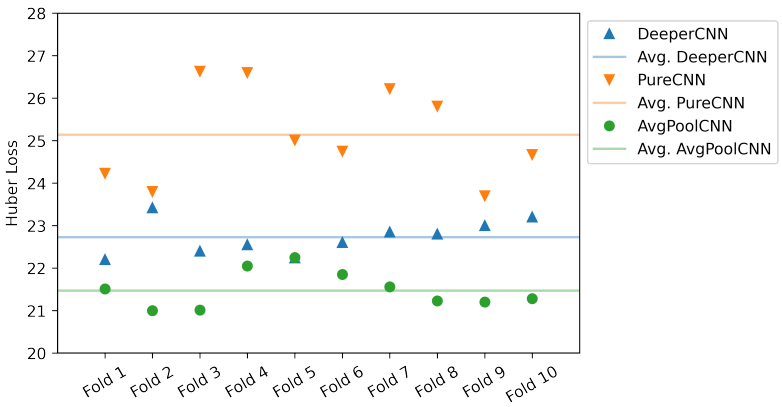


Figure 4.12: Huber losses across Folds for AvgPoolCNN.

Table 4.7: Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for AvgPooling.

	Huber Loss	MAPE Bellow 10%	MAPE Bellow 5%	MAPE Bellow 2.5%
Fold 1	21.51	97.6%	83.6%	49.1%
Fold 2	21.00	97.7%	84.5%	50.9%
Fold 3	21.01	97.6%	84.4%	50.5%
Fold 4	22.05	97.4%	82.6%	47.6%
Fold 5	22.25	96.9%	82.5%	48.2%
Fold 6	21.85	96.9%	83.3%	51.8%
Fold 7	21.56	97.2%	84.4%	52.1%
Fold 8	21.23	97.0%	84.2%	52.9%
Fold 9	21.20	97.7%	84.0%	48.0%
Fold 10	21.28	97.2%	84.6%	48.0%
Average	21.47	97.3%	83.8%	50.3%

Based on these arguments, and backed by these results, the pooling process selected for the model was the average pooling, as it had far better results than the max pooling.

After such an improvement in performance, the pooling process used for the model was changed to average pooling. It's increase in performance justify running this unconventional layout.

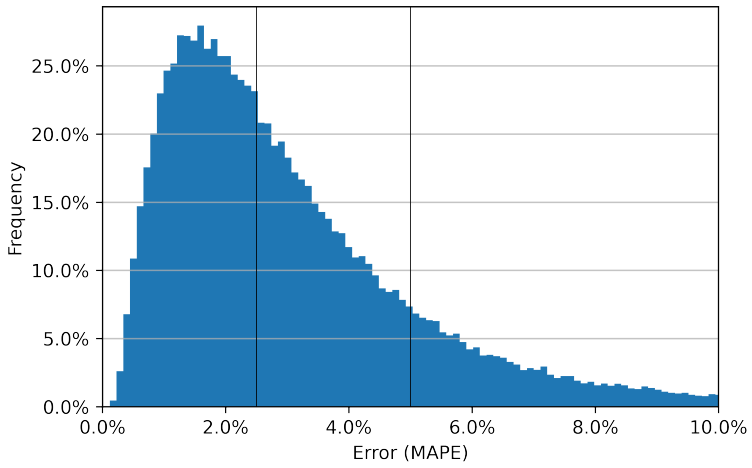


Figure 4.13: Relative Histogram of the MAPE for the Test of AvgPooling.

4.2.4 Replacing Flatten with Global Average Pooling

The final test for new layers that came with an accuracy improvement was the replacement of the flatten layer for a more recent approach, the global average pooling.

Table 4.8: Average Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for AvgPoolCNN.

	Huber Loss	MAPE b. 10%	MAPE b. 5%	MAPE b. 2.5%
DeeperCNN	22.73	96.8%	81.5%	46.9%
AvgPoolCNN	21.47	97.3%	83.8%	50.3%
Difference	-1.3(5.5%)	+0.5%	+2.3%	+3.4%

The flatten operation transforms the output of the convolutional-pooling block into a one dimensional array, that can then be inputted into the following dense layers that cap the process, as been done so far.

The global average pooling layer works more in line with the logic behind convolutional neural-networks, as it reads from the feature maps outputted by the previous convolutional-pooling block. In a process illustrated in Figure 4.14, the average from each map is calculated, and then fed into a one-dimentional array. This results in an analysis of each filter as a whole, and establishes an early relation between them all. This can then be directly outputted as the final result from the neural-network, or can be processed by a final dense block, to establish the final relation between the feature maps and the output.

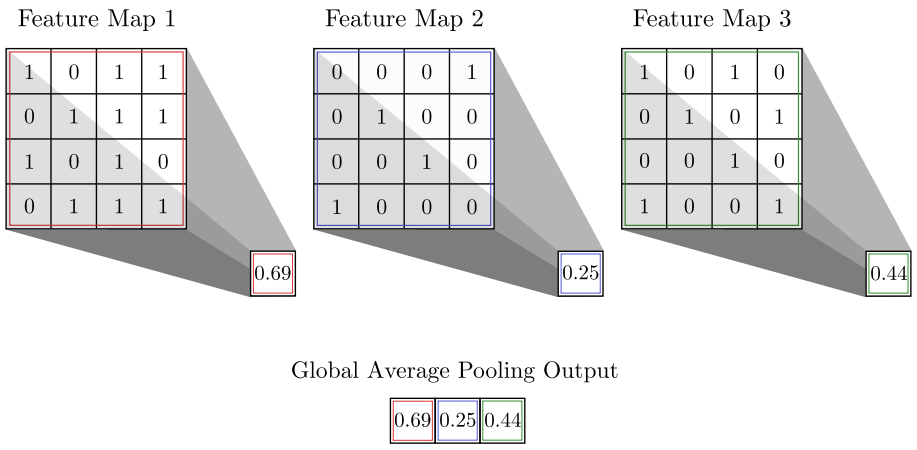


Figure 4.14: Global Average Pooling of three 4×4 feature maps.

To analyze the performance of this layer, a new model was constructed based on AvgPoolCNN (Model 5), called GlobAvgPoolCNN (Model 6), simply replacing the flatten layer used thus far. In hindsight, this may not have been the best approach, as this layer changes completely the workings of the neural network, and called-for a more careful and in-depth analysis. However, this tests still provided good feedback on the change's impact on model's accuracy.

Model 6: GlobAvgPoolCNN - CNN with Global Average Pooling

Input: Mesh (Shape: 16×16 integer matrix)

2×	**2D Convolution Layer:** Number of Kernels: 8 Kernel Size: 3 Activation: ReLu Stride: (1,1) **Batch Normalization Layer**
4×	**2D Convolution Layer:** Number of Kernels: 8 Kernel Size: 3 Activation: ReLu Stride: (1,1) **Batch Normalization Layer** **Average Pooling 2D Layer:** Filter Size: (2,2) Stride: (1,1)
1×	**Global Average Pooling 2D Layer**
2×	**Dense Layer:** Number of Nodes: 64 **Batch Normalization Layer**
1×	**Dense Layer:** Number of Nodes: 11

Output: Curve (Shape: 11 float array)

Model Evaluation

The results from the Stratified K-Fold CV for this specific model (Tables 4.9 and 4.10 and Figure 4.15) proven to be not as inspiring as expected, but still showed an improvement to the previous the values. This performance, However marginal as it might be, during tuning, the next stage of model development, proven to have added benefits, as the number of kernels per convolutional layer was substantially increased.

Even with such a small difference, this new layer type was adopted into the model.

Table 4.9: Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for GlobAvgPoolCNN.

	Huber Loss	MAPE Bellow 10%	MAPE Bellow 5%	MAPE Bellow 2.5%
Fold 1	21.69	96.6%	83.8%	53.5%
Fold 2	21.27	97.3%	84.5%	52.8%
Fold 3	21.38	97.5%	84.4%	49.3%
Fold 4	21.33	97.3%	85.3%	51.3%
Fold 5	21.84	97.2%	84.2%	48.2%
Fold 6	20.78	97.1%	85.6%	56.0%
Fold 7	21.42	97.2%	83.4%	52.1%
Fold 8	21.24	97.5%	84.0%	50.6%
Fold 9	20.42	97.5%	85.4%	54.6%
Fold 10	21.42	97.3%	84.8%	50.3%
Average	21.28	97.2%	84.5%	51.9%

Table 4.10: Average Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for GlobAvgPoolCNN.

	Huber Loss	MAPE b. 10%	MAPE b. 5%	MAPE b. 2.5%
AvgPoolCNN	21.48	97.3%	83.8%	50.3%
GlobAvgPoolCNN	21.28	97.2%	84.5%	51.9%
Difference	-0.2(0.01%)	-0.1%	+0.7%	+1.6%

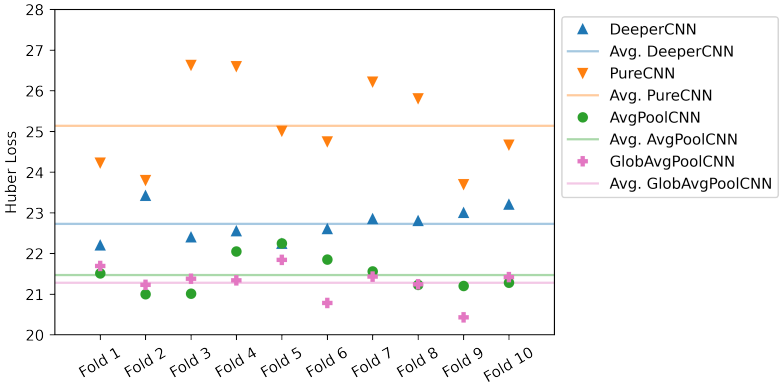


Figure 4.15: Huber losses across Folds for GlobAvgPoolComp.

Chapter 5

Model Refinement and Data Augmentation

After the model's layers and basic structure is defined, it enters the next stage of development, where both it and the dataset used for training are refined and expanded.

Starting with model refining, a tuning package called Optuna was used to ease the process of hyperparameter selection, extracting performance data over hundred of models trained. This allowed to establish a relation between multiple model settings and the testing losses, helping to select parameters such as batch size, learning rates, number of layers, and many more. During these analyses, some evidences of overfitting were found, and it was reduced through the implementation of Dropout layers.

After tuning, when it's considered that the model achieved peak performance, attentions are turned to the dataset. To improve it, it was expanded through the addition of new RVEs and some classical and custom-made data augmentation techniques.

5.1 Model Tuning

Model tuning consists of analyzing the different hyperparameters and their respective impact on model performance, and adjusting them accordingly. Whist manual adjustment is possible, as done thus far for selecting the general architecture, establishing which layers were more capable of predicting the stress strain curve, and was the first approach to fine-tuning this model, it has proven very labor-intensive due to the large amount of variable parameters. It also needed constant overview and user input. For this very reason, it's preferable to implement a solution based on a pre-made package, such as Optuna or Hyperopt, which automate the AI training and evaluation, accelerating and facilitating this process.

In this section, there is a brief overview of Optuna and its workings, to ease the interpretation of the results of the five key analyses that moved the model analysis forward. In the end, the tuned model is pitched against the previous architectures in the same Stratified F-Fold CV.

5.1.1 Implementing Optuna

Optuna is a Python package and one of the most popular hypertuning frameworks there is. It runs concurrently with TensorFlow, allowing automatic training and hyperparameter testing. It uses information regarding training history and performance to predict an optimal interval for each hyperparameter that might lead to better accuracy. [25][26]

An Optuna analysis is called a study, and for each one, a model is programmed similarly to a solo analysis, but replacing whatever parameter intended to analyze by a `suggest` function. A study can be separated into trials. For each, the hyperparameters at hand are selected from a predicted optimal interval, and the model compiled and trained. This is then tested using an unseen section of the dataset, outputting a testing loss, used as the evaluation parameter. At the end of the trial, Optuna uses the history of selected hyperparameters and their resulting losses to update the optimal interval used for the selection at the start of the next trial. [27]

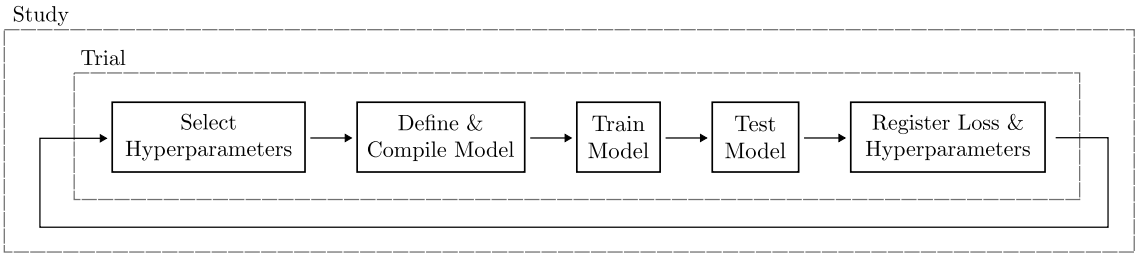


Figure 5.1: Schematic of the workings of Optuna's Hyper-tuning.

The `suggest` function can generate many types of hyperparameters, but the ones used in this work are:

- Integer;
- Float;
- Category.

For both integers and floats, an intended interval is selected, and the parameters will be contained inside these. The categories limit the choice to a list of strings defined in the code. These can either be activation functions, losses, initializers or integers/floats when is only intended for those to be amongst a certain selection, what ever the reason may be. For each analysis, the type and settings for each variable hyperparameter are detailed.

For implementation, firstly is necessary to define the objective function. This is the code to be run every trial, and must include model creation, compiling and training, beside the definition of the testing hyperparameters. In the end, it must output an evaluation value for each trial. The objective function used for the analyses in this chapter is defined in Algorithm 4. Worth noting that for every trial, the model is trained two times. This is done to reduce the effect of the variation

on model performance for each training, caused by the Adam optimizer and the randomness of the initial weights[1]. [13]

Algorithm 4 Objective Function

```
Clear TensorFlow's Backend Section                ▷ Done to preserve RAM
Create empty array Losses
Define hyperparameters                            ▷ Both fixed and variable
for i in [1,2] do
    Define and compile model
    Train model
    Calculates test_loss
    Losses append test_loss
end for
best_loss ← minimum of Losses
Return best_loss
```

Afterwards, the study itself must be defined, selecting the direction intended for the evaluation criteria: either maximize, used for model accuracy, of minimize, for losses, such as in this application. Once defined, it can be run, specifying the maximum number of trials the study can contain.

5.1.2 Analysis 1: evaluate model's tendencies

The first step in the optimization process was to simply give Optuna as much control over the architecture as possible. This was done to try to identify which hyperparameters were limiting the model the most, highlighting the best place to start fine-tuning.

For this run, the model used is described in Model 7, where each variable hyperparameter is identified in 5.1. In this stage, four separate studies were carried out, to verify results. Each of them consisted of 250 trials, over a random dataset of 10 000 RVEs. This was done to speed up the tuning stage, as a decreased dataset allowed faster training. Worth noting that the testing and training sets where kept consistent over all studies in this chapter, to guarantee direct comparison between results. These are presented on Table 5.2.

Table 5.1: Variable hyperparameters in Analysis 1.

Hyperparameter	Symbol	Type
Number of Kernels	α	categorical - 16 or 32
Repeats of Conv. Block	β	categorical - 4, 6 or 8
Repeats of Conv. & Pool. Block	γ	categorical - 4, 6 or 8
Size of Dense Layers	δ	categorical - 32, 64 or 128
Batch Size	—	categorical - 8 or 16
Learning Rate	—	float - $[1 \times 10^{-4},\ 1 \times 10^{-1}]$

[1]The weights were initiated using the random normal distribution implemented in Keras[28]

Model 7: Optuna1 - Model for Analysis 1

Input: Mesh (Shape: 16×16 integer matrix)

$\beta\times$ **2D Convolution Layer:**
- Number of Kernels: α
- Kernel Size: 3
- Activation: ReLu
- Stride: (1,1)

Batch Normalization Layer

$\gamma\times$ **2D Convolution Layer:**
- Number of Kernels: α
- Kernel Size: 3
- Activation: ReLu
- Stride: (1,1)

Batch Normalization Layer

Average Pooling 2D Layer:
- Filter Size: (2,2)
- Stride: (1,1)

$1\times$ **Global Average Pooling 2D Layer**

Dense Layer:
- Number of Nodes: δ

Batch Normalization Layer

$1\times$ **Dense Layer:**
- Number of Nodes: $\delta/2$

Dense Layer:
- Number of Nodes: 11

Output: Curve (Shape: 11 float array)

Table 5.2: Best results for each Study in Analysis 1.

	Studies			
Hyperparameter	1	2	3	4
Number of Kernels	32	32	32	32
Repeats of Conv. Block	4	4	8	8
Repeats of Conv. & Pool. Block	4	4	4	4
Size of the Dense Layers	64	32	64	128
Batch Size	16	16	16	16
Learning Rate	1.06×10^{-2}	1.23×10^{-2}	4.03×10^{-3}	5.51×10^{-3}
Huber Loss	21.96	22.44	24.15	21.00

This analysis allowed to establish some hyperparameters straightaway, such as number of kernels and batch size, that were kept consistent amongst all studies. The size of dense layers was quite ambiguous, and none seemed to outperform the others. Learning rate was kept as a variable because during manual tuning the model proven to be very picky with it, so allowing Optuna to adjust accordingly made the most sense, and is not expected to be fixed this early on. However, the most relevant result was the number of repeats of the convolutional & pooling block, where, all studies indicated better performance the lower the value. This might indicate some overfitting, as it can occur with excessively deep neural networks. This theory was backed by the tendency for lower repeats of the convolutional block as well, even if it wasn't as accentuated as with the previous one.

5.1.3 Analysis 2: implementing Dropout

Overfitting tends to happen with larger neural networks, with more layers and more nodes per layer, so, an obvious way to oppose this is to decrease the number of trainable model parameters. On the past decade, however, a new methodology has proven capable of tackling this phenomenon, particularly useful for larger models, called the Dropout. In this method, whenever a dropout layer is present, it randomly drops part of the connections between nodes of its adjacent layers. This makes the training process noisier, introducing some randomness in the architecture. [29]

One theory on why this might help performance, is that it breaks some co-adaption by some layers developed during training, where one corrects mistakes from the other. This decreases co-dependency of some layers for feature recognition, resulting in a more robust model. [30]

For this analysis, Dropout was implemented into the model after the convolutional layers in both blocks, with variable drop rates between them. The model was fixed at 8 repeats for both convolutional blocks, trying to preserve model depth and correct the overfitting using solely the Dropout. The Model 8 was tested with both ε and ζ being the Dropout rates being floats between 0.05 and 0.30 (Table 5.3) and results are presented on Table 5.4.

Model 8: Optuna2 - Model for Analysis 2

Input: Mesh (Shape: 16×16 integer matrix)

8× **2D Convolution Layer:**
- Number of Kernels: 32
- Kernel Size: 3
- Activation: ReLu
- Stride: (1,1)

Dropout Layer:
- Drop Rate: ε

Batch Normalization Layer

8× **2D Convolution Layer:**
- Number of Kernels: 32
- Kernel Size: 3
- Activation: ReLu
- Stride: (1,1)

Dropout Layer:
- Drop Rate: ζ

Batch Normalization Layer

Average Pooling 2D Layer:
- Filter Size: (2,2)
- Stride: (1,1)

1× **Global Average Pooling 2D Layer**

Dense Layer:
- Number of Nodes: 64

Batch Normalization Layer

1× **Dense Layer:**
- Number of Nodes: 32

Dense Layer:
- Number of Nodes: 11

Output: Curve (Shape: 11 float array)

Table 5.3: Variable hyperparameters in Analysis 2.

Hyperparameter	Symbol	Type
Dropout 1	ε	float - $[0.05,\ 0.30]$
Dropout 2	ζ	float - $[0.05,\ 0.30]$
Learning Rate	—	float - $[1 \times 10^{-4},\ 1 \times 10^{-1}]$

Table 5.4: Best results for each Study in Analysis 2.

	Studies		
Hyperparameter	1	2	3
Dropout 1	0.06	0.06	0.04
Dropout 2	0.25	0.22	0.23
Learning Rate	8.33×10^{-3}	8.11×10^{-3}	6.25×10^{-3}
Huber Loss	22.92	23.04	23.8

The result from the studies conducted on Model 8, show that, to compensate for the overfitting, a massive dropout at the second block was needed. As such, the best solution might well be to reduce model size, as trying to compensate for overfitting by simply including dropout in the architecture proven inefficient.

5.1.4 Analysis 3: early drop rates

Accepting that the number of repeats for the second block must be reduced, for further testing it felt necessary to establish temporary dropout rates, even if these must be retuned in the future. This would allow following studies to better represent how the final architecture would work.

For this analysis, the number of repeats for the second block were fixed at 4 to combat overfitting, but keeping the first at 8 cycles, in an attempt to maintain depth. Dropouts were implemented all across the model, but to have as little testing variables as possible, the rate of the dense block was selected to be half of the convolutional & pooling block. The size of the dense layers was also kept variable, as previous analysis were inconclusive about its influence. This is detailed in Table 5.5 and Model 9, and the results are presented on Table 5.6.

Model 9: Optuna3 - Model for Analysis 3

Input: Mesh (Shape: 16×16 integer matrix)

8×	**2D Convolution Layer:** Number of Kernels: 32 Kernel Size: 3 Activation: ReLu Stride: (1,1) **Dropout Layer:** Drop Rate: ε **Batch Normalization Layer**
4×	**2D Convolution Layer:** Number of Kernels: 32 Kernel Size: 3 Activation: ReLu Stride: (1,1) **Dropout Layer:** Drop Rate: ζ **Batch Normalization Layer** **Average Pooling 2D Layer:** Filter Size: (2,2) Stride: (1,1)
1×	**Global Average Pooling 2D Layer**
1×	**Dense Layer:** Number of Nodes: δ **Batch Normalization Layer** **Dense Layer:** Number of Nodes: $\delta/2$ **Dense Layer:** Number of Nodes: 11

Output: Curve (Shape: 11 float array)

Table 5.5: Variable hyperparameters in Analysis 3.

Hyperparameter	Symbol	Type
Dropout 1	ε	float - $[0.05,\ 0.30]$
Dropout 2	ζ	float - $[0.05,\ 0.30]$
Size of Dense Layers	δ	categorical - 4, 6 or 8
Batch Size	—	categorical - 8 or 16
Learning Rate	—	float - $[1 \times 10^{-4},\ 1 \times 10^{-1}]$

Table 5.6: Best results for each Study in Analysis 3.

	Studies			
Hyperparameter	1	2	3	4
Dropout 1	0.22	0.15	0.08	0.14
Dropout 2	0.17	0.13	0.30	0.08
Size of the Dense Layers	64	32	32	64
Batch Size	16	16	16	16
Learning Rate	1.76×10^{-2}	5.86×10^{-2}	1.05×10^{-2}	1.01×10^{-2}
Huber Loss	24.50	24.21	26.10	21.87

From these results, the dropout for the second block was fixed at 0.8, as this one had, by far, the best results. However, there is still some uncertainty regarding the behavior of the first block, as its size may still be affecting performance. This analysis confirmed the Batch Size from the previous studies.

5.1.5 Analysis 4: testing blocks' repeats vs Dropout

The main goal of this analysis is to understand the relation between the size of the first block and the testing loss when under the influence of Dropout. Letting only those two vary should show which does the architecture favor and help fix the repeats of the convolutional block.

For this, the only variables were the number of cycles for the convolutional block and the drop rate inside it, as per Table 5.7 and Model 10. The number of nodes of the Dense Layers were kept at 64 and 32 each, as the model previously shown to be more inclined to lower values instead of 128 and 64, but did not show massive difference between 64/32 and 32/16. The number of studies was also doubled, to better reinforce conclusions, as some of previous studies where differing slightly too much between them.

Model 10: Optuna4 - Model for Analysis 4

Input: Mesh (Shape: 16×16 integer matrix)

$\beta\times$
- **2D Convolution Layer:**
  - Number of Kernels: 32
  - Kernel Size: 3
  - Activation: ReLu
  - Stride: (1,1)
- **Dropout Layer:**
  - Drop Rate: ε
- **Batch Normalization Layer**

$4\times$
- **2D Convolution Layer:**
  - Number of Kernels: 32
  - Kernel Size: 3
  - Activation: ReLu
  - Stride: (1,1)
- **Dropout Layer:**
  - Drop Rate: 0.08
- **Batch Normalization Layer**
- **Average Pooling 2D Layer:**
  - Filter Size: (2,2)
  - Stride: (1,1)

$1\times$
- **Global Average Pooling 2D Layer**

$1\times$
- **Dense Layer:**
  - Number of Nodes: 64
- **Dropout Layer:**
  - Drop Rate: $\zeta/2$
- **Batch Normalization Layer**
- **Dense Layer:**
  - Number of Nodes: 32
- **Dense Layer:**
  - Number of Nodes: 11

Output: Curve (Shape: 11 float array)

Table 5.7: Variable hyperparameters in Analysis 4.

Hyper-parameter	Symbol	Type
Repeats of Conv. Block	β	categorical - 4, 6 or 8
Dropout 1	ε	float - [0.05, 0.30]

Table 5.8: Best results for each Study in Analysis 4.

	Studies							
Hyper-parameter	1	2	3	4	5	6	7	8
Repeats of Conv. Block	6	4	4	4	8	4	4	4
Dropout 1	0.09	0.16	0.17	0.15	0.11	0.26	0.13	0.12
Huber Loss	27.55	23.22	21.44	22.06	22.67	27.56	21.52	22.71

The results from these studies (Table 5.8) are very consistent, proving that the model was still too deep, and this was causing loss in accuracy. The Dropout rates for the first block also seem to be optimal in between 0.10 and 0.15. This will be fine-tuned next, alongside the other rates.

Worth noting, lastly, that the increase in the number of studies also helped distinguish the best results, and should have been this many from the beginning. This comes with an extended processing time, however, and would not be practical to increase it any further with current equipment, as this analysis took several days of processing as is.

5.1.6 Analysis 5: final Dropout rates

In a neural network, it's necessary that all layers work together, and, as such, it's ideal to tune them all at once, especially when dealing with such chaotic layers as Dropout. The previous studies provided some early estimates for the best rates for each block, but, once established the number of cycles, it was crucial to re-do these tests, to verify if these values still maintain in these intervals.

The sole objective of this analysis is to obtain a set of three distinct drop rates (Table 5.9) for the multiple layers in Model 11, that would result in the best accuracy.

Model 11: Optuna5 - Model for Analysis 5

Input: Mesh (Shape: 16×16 integer matrix)

4×
- **2D Convolution Layer:**
  - Number of Kernels: 32
  - Kernel Size: 3
  - Activation: ReLu
  - Stride: (1,1)
- **Dropout Layer:**
  - Drop Rate: ε
- **Batch Normalization Layer**

4×
- **2D Convolution Layer:**
  - Number of Kernels: 32
  - Kernel Size: 3
  - Activation: ReLu
  - Stride: (1,1)
- **Dropout Layer:**
  - Drop Rate: ζ
- **Batch Normalization Layer**
- **Average Pooling 2D Layer:**
  - Filter Size: (2,2)
  - Stride: (1,1)

1×
- **Global Average Pooling 2D Layer**

1×
- **Dense Layer:**
  - Number of Nodes: 64
- **Dropout Layer:**
  - Drop Rate: η
- **Batch Normalization Layer**
- **Dense Layer:**
  - Number of Nodes: 32
- **Dense Layer:**
  - Number of Nodes: 11

Output: Curve (Shape: 11 float array)

Table 5.9: Variable hyperparameters in Analysis 5.

Hyper-parameter	Symbol	Type
Dropout 1	ε	float - [0.05, 0.30]
Dropout 2	ζ	float - [0.05, 0.30]
Dropout 3	η	float - [0.05, 0.30]

Table 5.10: Best results for each Study in Analysis 5.

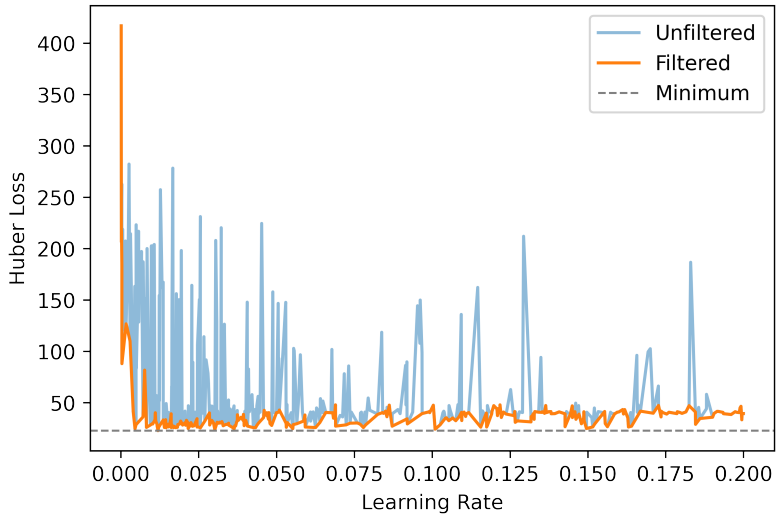
	Studies							
Hyper-parameter	1	2	3	4	5	6	7	8
Dropout 1	0.14	0.18	0.15	0.24	0.17	0.15	0.11	0.12
Dropout 2	0.21	0.28	0.19	0.08	0.05	0.26	0.30	0.10
Dropout 3	0.18	0.09	0.13	0.15	0.19	0.15	0.07	0.05
Huber Loss	21.68	24.82	24.45	24.24	26.85	24.74	21.68	21.37

The results from these studies are quite varied, but the Studies 1, 7 and 8 proven to be far more capable of curve prediction than the other ones. From these, Study 8 proven to be the best, and, as the rate of Dropout's was decreasing along the model, it established the end values for these.

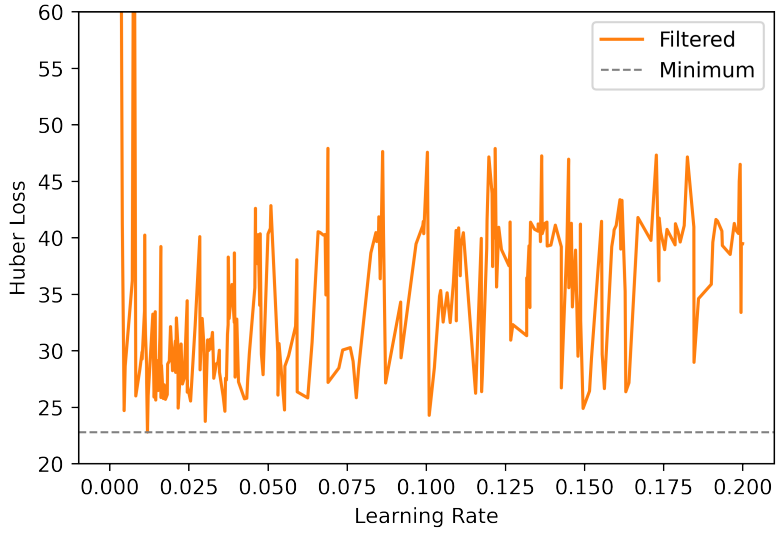
5.1.7 Analysis 6: Learning Rate

Learning Rate is one of the most important hyperparameters for model performance, and has proven to be, thus far, one of the hardest to set. During manual tuning, slight variations in learning rate could cause a working model to lose consistency during training, resulting in losses five to seven times the usual for that architecture. These losses would not occur every training, but could happen as much as half the time, even if learning rate was sightly off the useful interval.

To set this hyperparameter, the Model 11 with the losses established in the previous analysis was tested over multiple studies for multiple learning rates, in order to establish the best performance wise. The results of these studies are presented on Table 5.12, with the relation between learning rate and Huber Loss during them on Figure 5.2.



(a)



(b)

Figure 5.2: Huber Loss by Learning Rate during Analysis 6.

Based on these results, the learning rate selected was of 0.01, but even this one had some serious consistency problems, as easly seen on Figure 5.2a, which shows the instability of model training, and the actual filtered results that could be used for tuning. On Figure 5.2b is also possible to see the slight increase in Losses with the increase of Learning Rate over 0.2.

The architecture consistency increased with the full dataset, and these occurrences of unpredictable training became few and far in-between. The cause of this extreme learning instability was never confirmed, as even changing the activation functions did not solve these issues. One thing that might explain it is the large number of trainable parameters when compared to the size of the reduced dataset, of around 17 000 to 10 000. This might result in some serious overfitting and, in turn, cause massive training losses. Another possible culprit would be the optimizer, as the

only one tested was Adam, but it wouldn't justify the training becoming consistent with dataset increase.

Table 5.11: Variable hyperparameters in Analysis 6.

Hyper-parameter	Symbol	Type
Learning Rate	—	float - $[1 \times 10^{-4},\ 1 \times 10^{-1}]$

Table 5.12: Best results for each Study in Analysis 6.

	Studies				
Hyper-parameter	1	2	3	4	5
Learning Rate	3.04×10^{-2}	3.64×10^{-2}	1.19×10^{-2}	1.01×10^{-1}	1.88×10^{-2}
Huber Loss	22.81	24.65	22.80	24.29	29.09

5.1.8 Tuned Model - Performance and Comparison

The tuned model is far more complex and capable of capturing all the necessary details to predict the stress strain curve from the mesh.

This step proved crucial to improve the architecture's performance, as it hinted to the existence of some considerable overfitting, and allowed the establishment of the necessary parameters for reducing it. This improvement can be seen both in Tables 5.13 and 5.14 and Figure 5.3, comparing the TunedCNN model to the previously tested ones, when tested over the same Stratified K-Fold CV.

Table 5.13: Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for TunedCNN.

	Huber Loss	MAPE Bellow 10%	MAPE Bellow 5%	MAPE Bellow 2.5%
Fold 1	18.89	97.9%	88.0%	59.4%
Fold 2	17.53	98.3%	90.2%	63.0%
Fold 3	18.97	98.0%	87.3%	58.8%
Fold 4	17.75	98.3%	90.2%	63.8%
Fold 5	17.57	98.4%	90.0%	62.6%
Fold 6	18.08	98.3%	89.2%	60.9%
Fold 7	18.50	97.9%	88.4%	61.5%
Fold 8	18.02	98.5%	88.8%	59.3%
Fold 9	17.83	98.2%	89.2%	61.5%
Fold 10	18.60	98.0%	88.3%	59.7%
Average	18.17	98.2%	88.9%	61.0%

Table 5.14: Average Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for TunedCNN.

	Huber Loss	MAPE b. 10%	MAPE b. 5%	MAPE b. 2.5%
GlobAvgPoolCNN	21.28	97.2%	84.5%	51.9%
TunedCNN	18.17	98.2%	88.9%	61.0%
Difference	-3.1(14.6%)	+1.0%	+4.4%	+9.1%

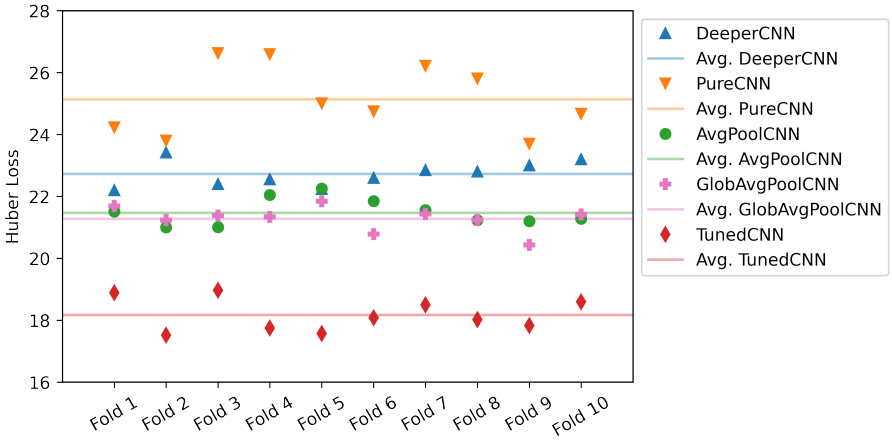


Figure 5.3: Huber losses across Folds for TunedCNN.

Worth noting that this improvement is due to the considerable increase in model parameters, as detailed in Table 5.15. It is interesting to see the evolution of number of parameters by model with the losses at every stage of this architecture's development (Figure 5.4). Beside the already known rise in training loss for PureCNN, followed by constant progress over the next iterations, the parameter evolution over iteration is quite curious. The considerable dip on PureCNN, quite intuitively, results in a massive decrease in accuracy, but the second results in a slight improvement on Huber Loss. This is due to the different workings of the Global Average Pooling and Flatten. The former outputs an array with same dimension as the number of kernels in the previous convolutional layer, in this case 8. The second rearranges the last kernel into a single array, dimension 256. The most important takeaway is the fact that with only 11.1% of parameters, it still outperformed the flatten layer. This demonstrates the power of this method for CNN and why this should have been taken into account when previously designing the network.

Table 5.15: Parameters evolution from GlobAvgPoolCNN to TunedCNN.

	Trainable	Non-trainable	Total
GlobAvgPoolCNN	8 803	352	9 155
TunedCNN	70 315	704	71 019
Increase	+698.8%	+200.0%	+675.7%

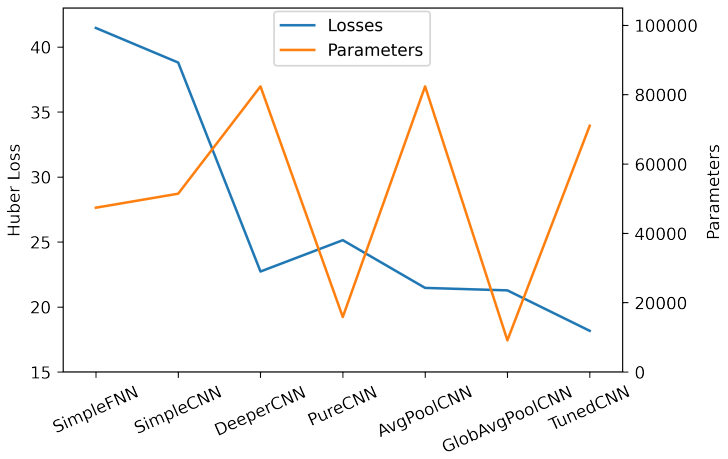
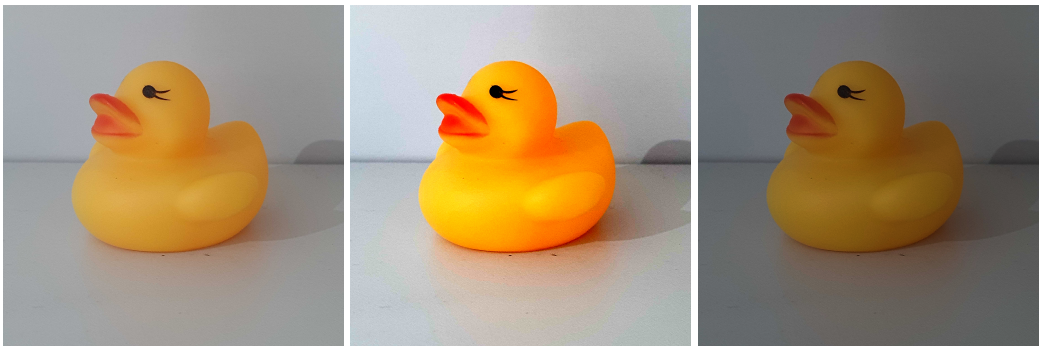


Figure 5.4: Evolution of Parameters and Losses over Model Development.

5.2 New Samples and Data Augmentation

As established before, one way to help a model's accuracy is to train it in more data, as a larger sample would allow a better and broader adjustment of the parameters. This can be done by either adding new samples to the dataset or by pretreating the data to introduce slight variations in the already existing entries that make them different from the original, without invalidating it.

One example of this technic used in image labeling, is to change the brightness or the contrast of the images in the dataset randomly, as explained in Figure 5.5. These changes do not invalidate the labels associated with the input, as each image is still a rubber duck, but for the algorithm, these variations of the original data still count as new, never before seen, entries. In image analysis or feature recognition, it allows the model to be exposed to multiple light and contrast conditions, making it more capable than a model without this training, for a very low cost in both data gathering and computing time. Although the dataset for this work is vastly different, and these exact approaches do not apply, similar technics may prove beneficial.



(a) Original (b) High Contrast (c) Low Brightness

Figure 5.5: Example of data augmentation on an image.

These procedures can be implemented in one of two ways: offline data augmentation, when the dataset itself gets bigger with the addition of pretreated samples, before training; or online or on-the-fly data augmentation, when the training data gets treated every epoch, with every entry having a chance of being modified. The preferred option, usually, tends to be online data argumentation, as it does not increase the memory allocation for the dataset, and can be faster when training, when compared to offline data argumentation. However, this might not always be viable, depending on the data augmentation technic intended. In this section, three different methods to artificially increase the data were used, one of them being offline.

Also in this section, new samples will be added to the original dataset and an overview of the impact this changes brought on the model's loss is done as well.

5.2.1 Increasing the Dataset

The first step in improving the dataset was to include new samples, by simply generating new RVEs and running their respective FE^2 analysis. Worth noting that this is not considered data argumentation, as all added RVEs and curves are brand new.

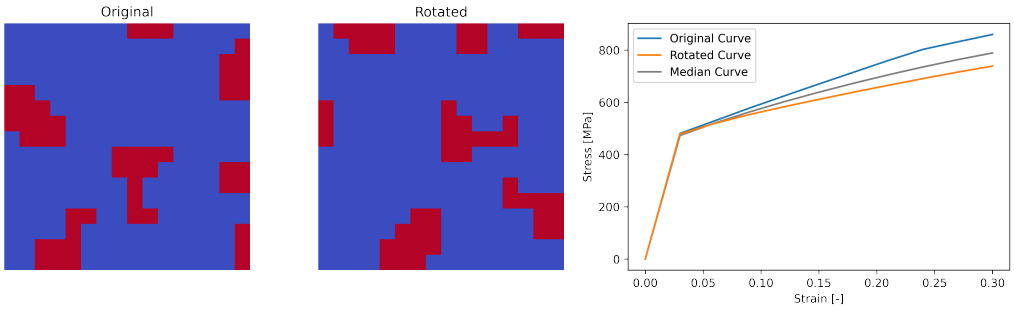
So, 50 000 new RVEs were generated, using the same methodology explained in Chapter 3. These were then mixed together with the original 100 000, totaling an increased dataset of 150 000 RVEs.

5.2.2 Rotating Meshes

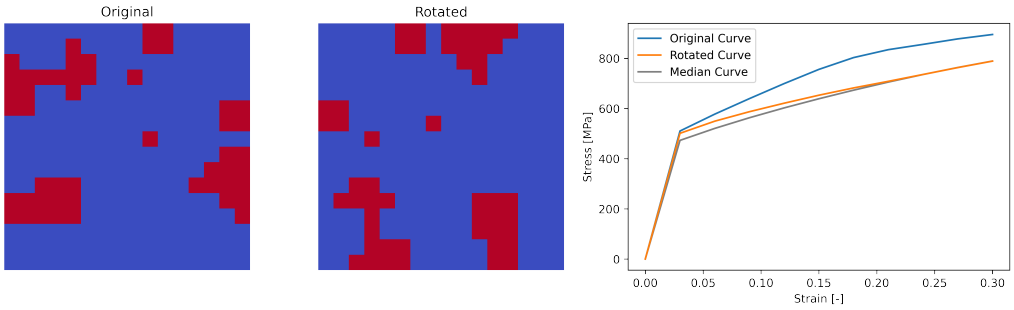
As previously explained, the most time-consuming step of generating new data was building the microstructure itself, as the FE^2 analysis was parallelized and could be done fairly quick. Although there were some attempts of circumventing the usage of Neper, in the end, it was still needed to generate new samples. This process was devised as a way to take advantage of the vasts amounts of already generated microstructures to obtain new curves.

The first thing to establish is that the microstructures generated should not be isotropic, that is, they should not have the same homogenized properties for all loading directions. This happens as the distribution of the hard phase elements forms clusters in the soft phase matrix, their size, shape and number would result in slight response variations depending on loading direction. That is not to say that none of them have very similar, or even the same response for different load configurations, as that is perfectly reasonable for some very specific arrangements. But the key aspect is that for the vast majority of RVEs, they have slightly anisotropic behavior. One could take advantage of this by including those different curves in the dataset, taken along the vertical loading of the microstructures, without the need for creating new meshes.

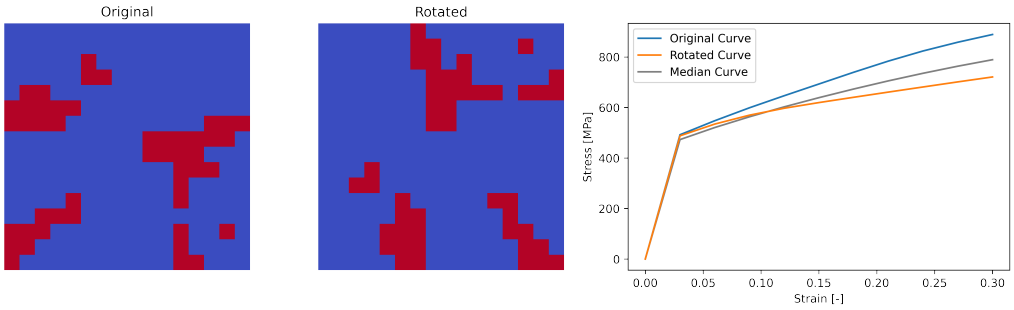
However, to do so, was a somewhat complicated process. The model is not capable of differentiating loading direction, because it always assumes horizontal loading, as that's what was used to generate all data thus far. So, to preserve the loading direction, the meshes themselves had to be rotated. By doing so, the model would have different RVEs than the originals, each with its own curve (Figure 5.6). This would effectively double the dataset, but skipping the longest step.



(a) Example 1



(b) Example 2



(c) Example 3

Figure 5.6: Original and Rotated RVEs with respective stress-strain curves.

Preparing the analysis itself was rather complex, as it involved the creation of a new `.rve` file for each of the 150 000 samples, then running Link2, separating the useful files and compiling a new `.pkl`. Luckily, as part of the Neper substitutes tests, the process for generating the `.rve` for a Python created mesh was already coded, and needed only minor adjustments. This was implemented using Python for RVE rotation and creating the respective `.rve` (Algorithm 5) and a Bash script similar to the one described in Algorithm 1, but skipping the Neper cycle.

Algorithm 5 RVE Rotation

```
df ← dataframe                                  ▷ Reads 150 000 RVEs dataframe
for micro in df'microstructures do              ▷ micro is a 256 element array
    micro ← reshaped micro as 16 × 16 matrix
    micro ← rotated micro by 90º
    Creates .rve from micro
end for
```

Once all 150 000 RVEs were rotated, analyzed and compiled, they were joined to the original 150 000, totaling a dataset of 300 000 samples. This is an example of offline data augmentation, as this was way more practical than doing it on-the-fly. It took about 9 days of processing to complete all FE^2 analyzes. An 150 000 sample dataset with Neper and Links2 would take about 20 days to generate from scratch on the same machine, which proves this method very good value per computing time.

5.2.3 RVE mirroring

As described previously when addressing the difference in layout for RVEs in Links2 and Python, it was established that mirroring an RVE alongside an axis parallel to its loading direction does not affect its response. However, this is not the only axis were this applies, as it is the same for the axis perpendicular to the first.

For this very reason, one possible implementation of data augmentation is to use this mirror planes to create new RVEs, that can be paired to the original's curve. As there are two different planes, this allows four variations of each entry.

Contrary to the previous method, this can be implemented on-the-fly, in parallel to training. To do so, a preprocessing Keras layer called `RandomFlip` [31] was added to the model. This will randomly flip the training inputs along both axis every epoch.

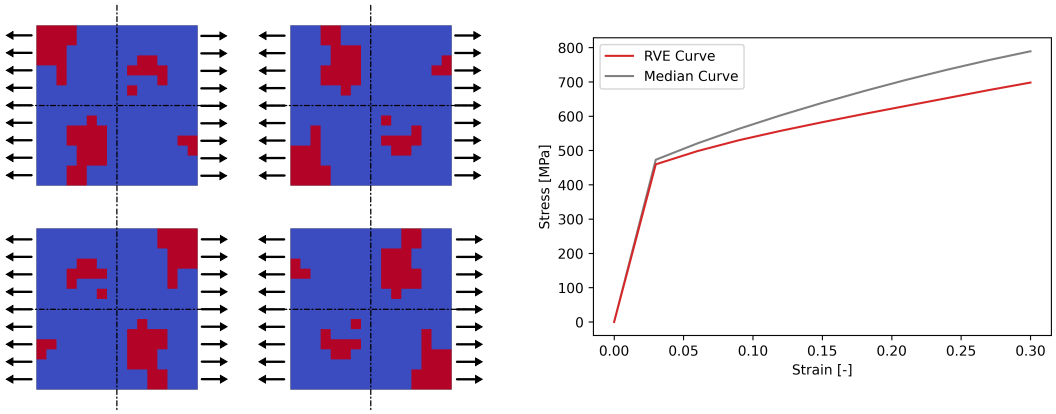


Figure 5.7: Mirrored RVEs and respective curve.

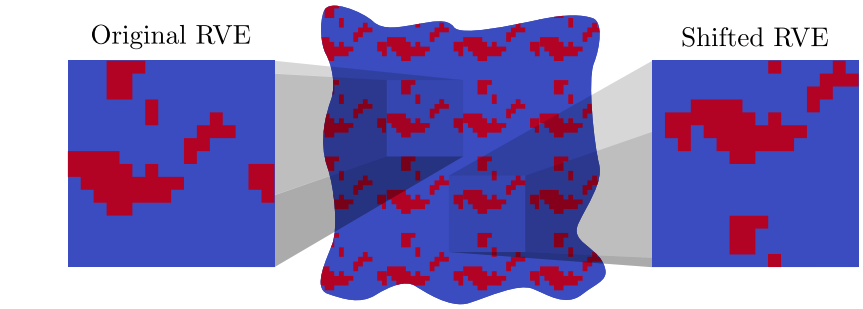
5.2.4 Shifting RVEs

Recalling when the periodic boundary condition was introduced, it was mentioned that it emulates a mosaic layout of the RVE, similarly to what is seen on Figure 5.8a. The micro scale boundary condition was chosen as it was lax, and it provided good convergence. Coincidently, this very decision opens an opportunity for massive online data augmentation.

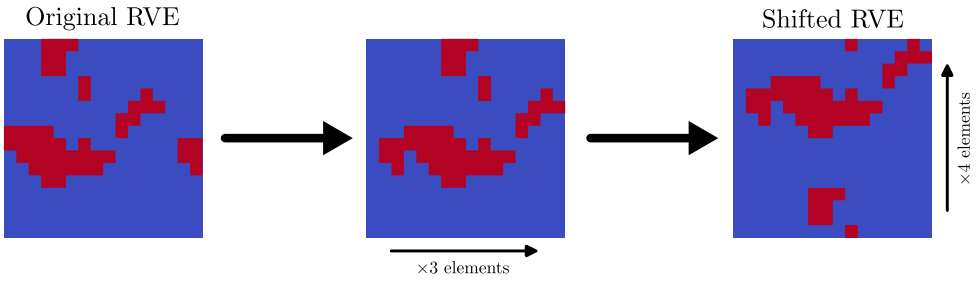
On Figure 5.8a, is represented the periodic arrangement formed by the RVE alongside it. As seen in this illustration, any 16×16 matrix that can be isolated from the larger microstructure,

is a periodic unit of it. This is apparent as the other isolated RVE, called Shifted RVE, when periodically arranged, will result in the same pattern as the original one.

As the periodic boundary condition applies in the dataset generated, it should come that any two RVEs that form the same periodic pattern should share their mechanical response for the same loading conditions. As such, the proposed data augmentation technic works on this, to shift the RVE around by some random amount in each epoch, resulting in as equivalent variant, as illustrated in Figure 5.8b.



(a) Periodic Arrangement



(b) Shifting Process

Figure 5.8: Representation of equivalent RVEs on their periodic arrangement (a) and the process to obtain the shifted one (b).

To verify this possibility, an isolated study was conducted, from which 200 entries from the dataset were separated and, for each, three variants from random shifts where selected. Then, all new RVEs underwent FE^2 analysis, and their stress strain curves were isolated. For all the variants, the study concluded that the difference between the variants where in the order 0.02%. This verified that this would be an applicable data augmentation technic.

This is, by far, the largest data augmentation technic available for this dataset, as each sample can be shifted horizontally 16 times, and another 16 times on top of that along the vertical axis, with no changes to the output. That was implemented using the `"RandomTranslation"` preprocessing layer [32], that works similarly to the previously explained `RandomFlip`.

5.2.5 Model Performance with Data Augmentation

Once all these data augmentation strategies were applied to the already expanded 150 000 RVEs, the number of different samples and permutations went from the original 100 000 to 307 200 000

RVEs. This massive increase is calculated on Table 5.16.

Table 5.16: Calculation of final dataset size.

Data Augmentation Technic	Multiplication Factor
RVE Rotation	2×
Mirror Horizontaly	2×
Mirror Vertically	2×
Shift Horizontaly	16×
Shift Vertically	16×
Size of Dataset	150 000 RVEs
Size of Augmented Dataset	307 200 000 RVEs

Obviously, not all of them were actually used during training, as it would take an unfathomable amount of time for training. Thanks to early stopping, a function that stops training once the model doesn't see improvements in a set number of epochs, the training is only as prolonged as it needs to be. As the two most intensive dataset augmentation technics were implemented on-the-fly, the algorithm only generates the necessary amount to train the model until it reaches a minimum.

To verify the effectiveness of the data augmentation, a Stratified K-Fold CV was prepared with the 300 000 RVEs dataset. This was then run with and without the other data augmentation methods, to verify their effectiveness. The results of these runs are presented on Tables 5.17 and 5.26 respectively.

Table 5.17: Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for TunedCNN for 300 000 RVEs.

	Huber Loss	MAPE b. 10%	MAPE b. 5%	MAPE b. 2.5%	MAPE b. 1%
Fold 1	16.05	98.6%	91.8%	68.4%	23.4%
Fold 2	16.75	98.6%	90.8%	65.4%	21.8%
Fold 3	16.61	98.6%	90.7%	65.4%	22.1%
Fold 4	16.09	98.7%	91.8%	67.9%	23.5%
Fold 5	17.14	98.3%	90.5%	65.5%	21.8%
Fold 6	16.55	98.7%	91.5%	66.2%	21.0%
Fold 7	16.60	98.6%	91.9%	68.2%	24.4%
Fold 8	16.74	98.8%	92.8%	65.9%	18.4%
Fold 9	16.81	98.4%	91.2%	66.7%	21.8%
Fold 10	16.94	98.5%	91.0%	66.6%	20.7%
Average	16.63	98.6%	91.4%	66.6%	21.9%

Table 5.18: Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for TunedCNN for Augmented Dataset.

	Huber Loss	MAPE b. 10%	MAPE b. 5%	MAPE b. 2.5%	MAPE b. 1%
Fold 1	22.20	97.1%	83.4%	52.2%	14.8%
Fold 2	21.50	96.9%	83.2%	52.1%	14.7%
Fold 3	22.37	96.6%	81.5%	49.9%	14.0%
Fold 4	22.16	97.0%	82.5%	48.9%	12.2%
Fold 5	22.40	97.1%	81.6%	46.5%	11.5%
Fold 6	21.06	97.0%	83.8%	52.6%	14.8%
Fold 7	22.24	96.6%	81.9%	52.3%	15.4%
Fold 8	23.78	95.8%	79.9%	48.4%	12.3%
Fold 9	21.70	97.1%	82.3%	50.7%	13.9%
Fold 10	21.67	96.8%	82.6%	52.1%	14.7%
Average	22.11	96.8%	82.3%	50.6%	13.8%

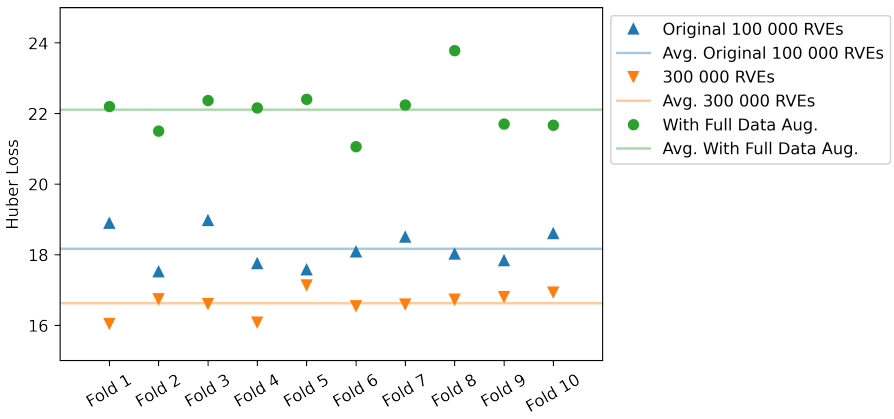


Figure 5.9: Changes in loss with data augmentation.

The results shown above were completely unexpected. At first, it was through that the massive increase in loss would be caused by the learning rate, as it was, thus far, problematic to choose. But after much manual tuning of this hyperparameter, the results would only get worst. However, after careful consideration, one possible culprit of this lack in performance might be a low number of trainable parameters. Previously, these were limited by the overfitting verified for the smaller dataset, but the increase in data is known to cause a reduction in overfitting on CNN models. In this case, this might even be causing some underfitting, not allowing the model to capture the necessary relations. To test this theory, and in search for a solution, the architecture entered a second stage of tuning.

5.3 Model Expansion

The possible underfitting that was happening in the previous section can be solved, possibly, by increasing the size of the neural network. On the first tuning stage, the model was increased as much as possible, until it started experiencing overfitting, and dropout layers had to be implemented to solve this issue. With the increased dataset, these limitations might no longer apply.

To test this theory, multiple studies were conducted yet again, but this time, to fully understand the limit of the network with this dataset, they were trained with the entire augmented dataset. This came with an increased computational cost, and, due to time restrictions, taking the same approach as with the previous tuning stage of doing multiple studies to confirm results was no longer viable. So, for each iteration in this tuning step, only one study was computed, but with more thoughtful analysis of each hyperparameter evolution during it and losses associated.

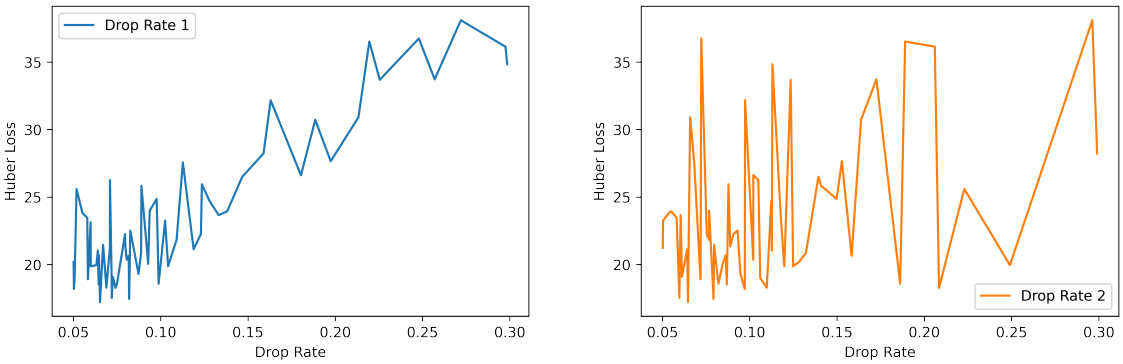
5.3.1 Study 1 - Tweaking Drop Rates

The first approach was to retune the drop rates at both dropout layers. They were set as variables according to Table 5.19. This study counted with 60 trials.

Table 5.19: Variable hyperparameters in Study 1.

Hyperparameter	Symbol	Type
Dropout 1	ε	float - [0.05, 0.30]
Dropout 2	ζ	float - [0.05, 0.30]

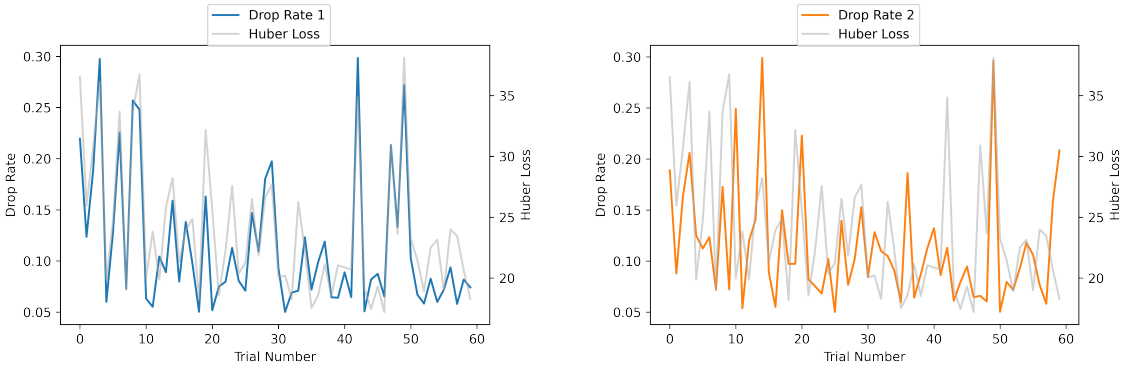
The study results were used to plot a relation between both drop rates and testing Huber losses (Figure 5.10), as well as them and losses over each trial (Figure 5.11). Bearing in mind that Optuna selects the hyperparameters according to where their optimal interval might be, based on the study thus far; overtime they should start to converge to that theoretically optimal value. All conclusions had to be taken from this limited amount of information due to the prolonged computing time required per trial.



(a) Drop Rate 1

(b) Drop Rate 2

Figure 5.10: Variation in Huber Loss for Drop Rates in Study 1.



(a) Drop Rate 1

(b) Drop Rate 2

Figure 5.11: Variation of drop rates and Huber loss over trials on Study 1.

From these results, one can see the clear tendency that the drop rate 1 has to drop to lower values, easily seen on both Figures 5.10a and 5.11a. The drop rate 2 does not have such a clear tendency, but seems from Figure 5.11b to be slowly tending for lower values. Along that, the losses seen on Figure 5.11 don't seem to be improving over the values obtained with the much smaller dataset.

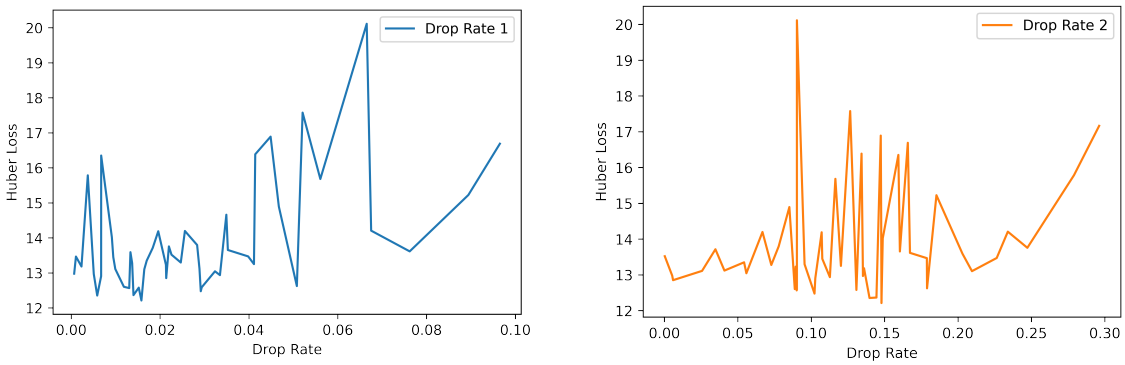
The main conclusion from this study was that the model may very well be having serious underfitting issues, but that may allow it to grow beyond what it could before.

5.3.2 Study 2 - Increasing the number of kernels

Concluding that the model may be underfitting, the next test was to see how it reacted to having double the number of kernels as before, jumping from 32 to 64. The variable hyperparameters were kept the same, but with slight changes to their allowed interval (Table 5.20), as both were allowed to drop to 0%, that is the same as having no layer at all. Drop rate 1 was also more limited in its upper value, as it had a great tendency for lower values in the previous study. The same results layout as before are presented on Figures 5.12 and 5.13.

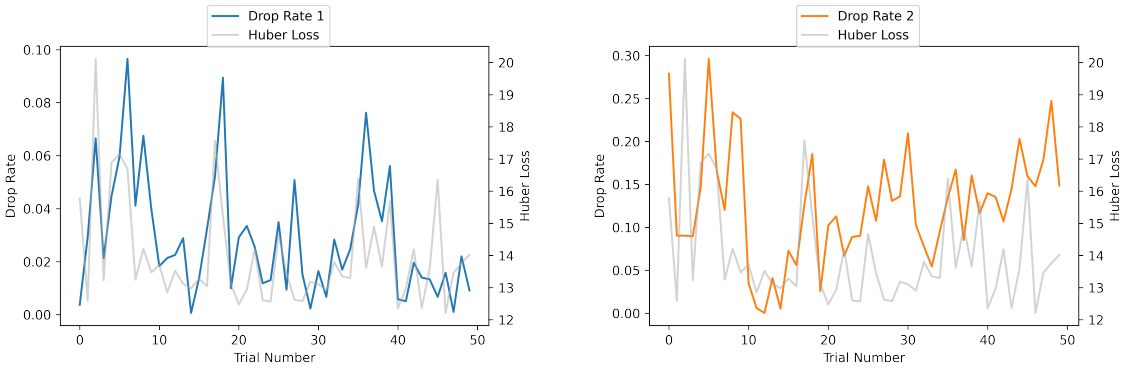
Table 5.20: Variable hyperparameters in Study 2.

Hyperparameter	Symbol	Type
Dropout 1	ε	float - [0.00, 0.10]
Dropout 2	ζ	float - [0.00, 0.30]



(a) Drop Rate 1 (b) Drop Rate 2

Figure 5.12: Variation in Huber Loss for Drop Rates in Study 2.



(a) Drop Rate 1 (b) Drop Rate 2

Figure 5.13: Variation of drop rates and Huber loss over trials on Study 2.

In this study, the drop rate 1 still seems to favor low values, with its best result at circa 1.5%, and the rate 2 starts to stabilize at around 15%. We see also a significant decrease in loss, that might confirm the underfitting theory. This also proves the need for a complete new model tuning, so these studies continued.

5.3.3 Study 3 - Eliminating Dropout 1

Considering the results thus far, everything points to a complete removal of the Dropout layer in the first block, so, keeping the rest of the model the same and only letting the drop rate 2 vary (Table 5.21), the study was repeated. The results are plotted on Figures 5.14 and 5.15. As there is only one variable, this study was kept much shorter, at 40 trials.

Table 5.21: Variable hyperparameters in Study 3.

Hyperparameter	Symbol	Type
Dropout 2	ζ	float - [0.00, 0.30]

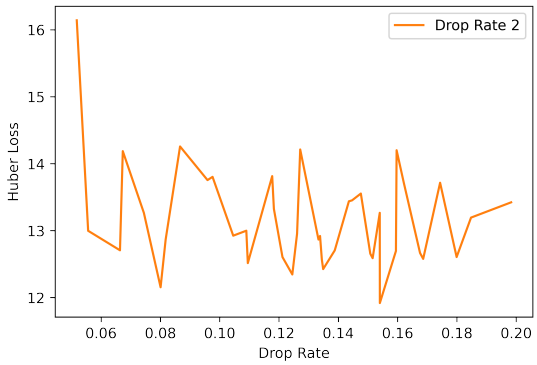


Figure 5.14: Variation in Huber Loss for Drop Rate 2 in Study 3.

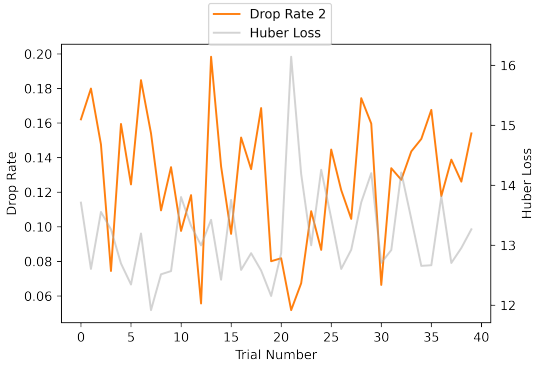


Figure 5.15: Variation of drop rate 2 and Huber loss over trials on Study 3.

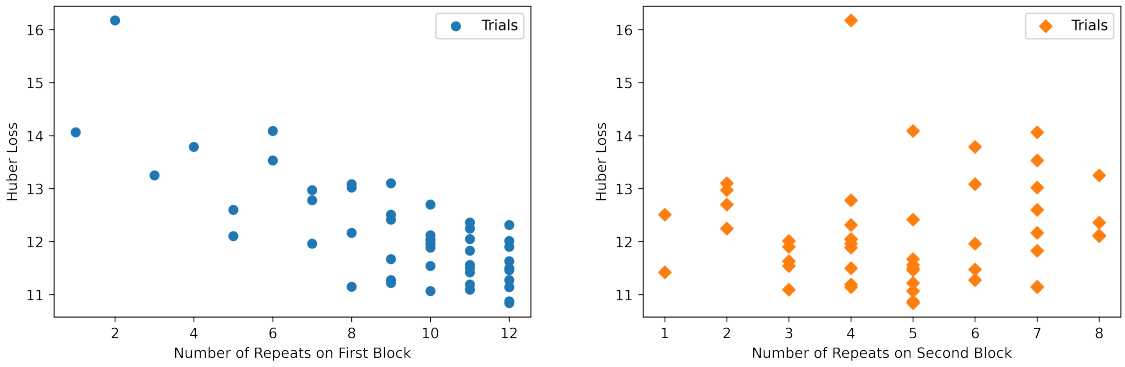
As seen on Figure 5.14, the losses without Dropout 1 were similar to the previous study, indicating that this layer might very well be no longer necessary. The second rate still seems to prefer values in the $[12\%, 16\%]$ interval, but is not very conclusive.

5.3.4 Studies 4 and 5 - Increasing blocks repeats

Thus far, in this tuning stage, the architecture still did not seem to have reached the number of critical parameter when they start being disadvantageous. To estimate just how large can the model grow, the number of repeats for both blocks were analyzed (Table 5.22). For this, both dropout layer were suppressed, so that only the effect of the block repeats could impact the results (Figures 5.16 and 5.17).

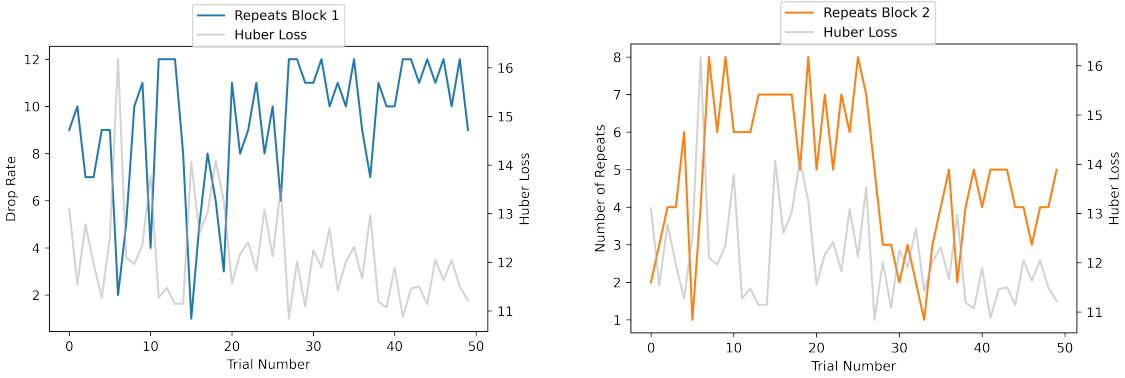
Table 5.22: Variable hyperparameters in Study 4.

Hyperparameter	Symbol	Type
Number of Repeats of First Block	ε	integer - $[1,\ 12]$
Number of Repeats of Second Block	ζ	integer - $[1,\ 8]$



(a) Block 1 (b) Block 2

Figure 5.16: Variation in Huber Loss for Block Repeats in Study 4.



(a) Block 1 (b) Block 2

Figure 5.17: Variation of block repeats and Huber loss over trials on Study 4.

The results from this study were promising, yet inconclusive for the first block. The model shows a clear tendency for higher values in Figure 5.16a, with a considerable cluster of runs in the lower right corner. This is also very apparent in Figure 5.17a, as the plots of number of repeats and Huber loss almost mirror each other. However, as these values are so close to the edge of the admissible interval, more testing is required to guarantee that the model still won't want even more repeats.

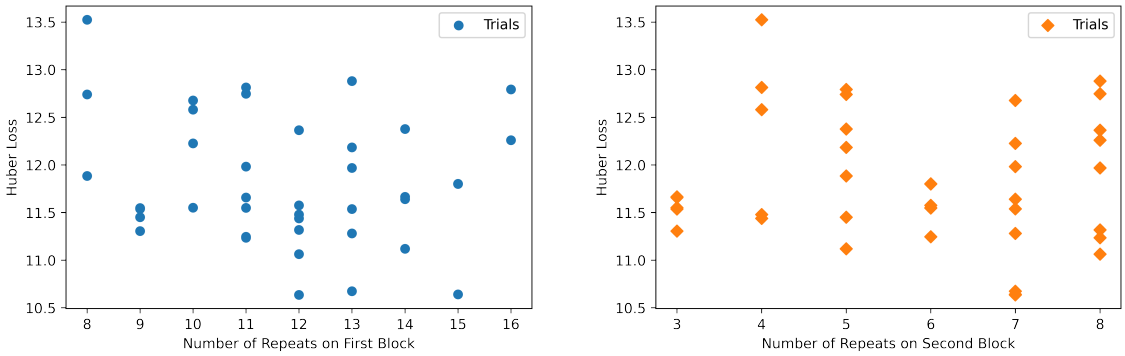
The second block seems more clear, as it favors 4 to 6 repeats, with some good results at 3 and 7 as well (Figure 5.16b). This is also confirmed by Figure 5.17b, as it seems to converge more to these values closer to the end of the study.

In the end, this study was deemed too ambiguous, and some changes in the admissible parameters for both blocks was deemed necessary. In a new study, the first block had its interval moved up, from 8 to 16, in order to examine where it will converge. The second simply had it's lower extremity trimmed off, as it was clear those were not giving good results. Finally, also in this analysis, the batch size was given room to vary slightly, to confirm if 32 was its best value.

However, as the batch size should always be in powers of two for efficiency, it was capped at either 32 or 64. All these variables are detailed in Table 5.23.

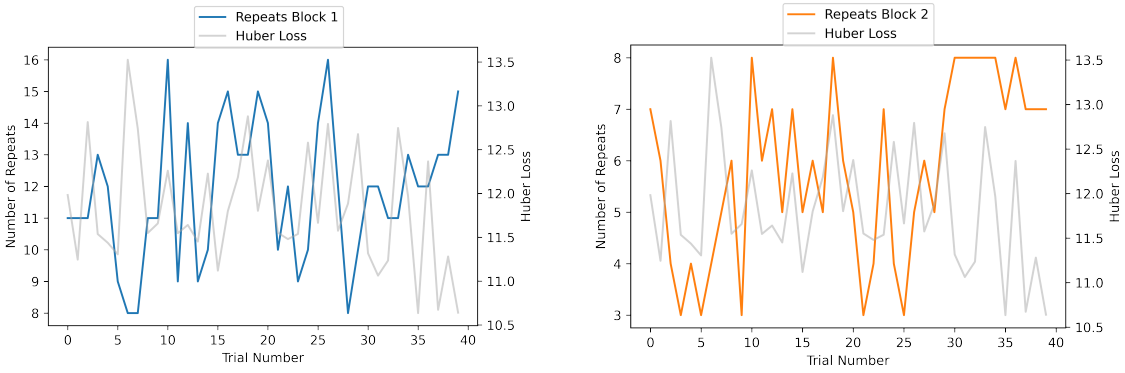
Table 5.23: Variable hyperparameters in Study 5.

Hyperparameter	Symbol	Type
Number of Repeats of First Block	ε	integer - $[1,\ 12]$
Number of Repeats of Second Block	ζ	integer - $[1,\ 8]$
Batch Size	—	categorical - 32 or 64



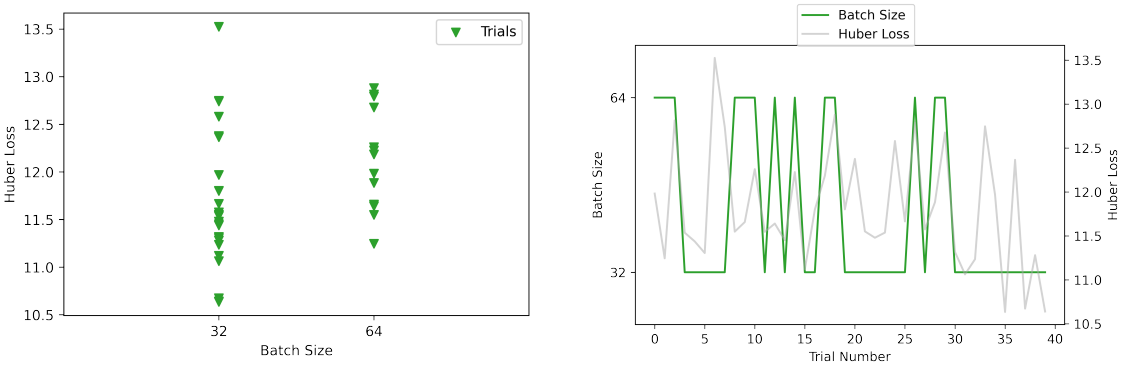
(a) Block 1 (b) Block 2

Figure 5.18: Variation in Huber Loss for Block Repeats in Study 5.



(a) Block 1 (b) Block 2

Figure 5.19: Variation of block repeats and Huber loss over trials on Study 5.



(a) Huber Loss per batch size

(b) Batch Size over trials

Figure 5.20: Batch Size variance in Study 5.

The first and most obvious conclusion to take is the preference the architecture has for the 32 batch size, easily seen on Figure 5.20. In Figure 5.20a the 32 has most of its runs with losses in the 11.0 to 11.7 range, while the 64 where more spread out, but closer to 12. This preference is also considerably apparent in Figure 5.20b, as the spikes in batch size are usually accompanied by increases in losses. This with this, the batch size was fixed at 32.

The next value to establish was the number of repeats for the first block. On Figure 5.16a it already had shown a tendency for 12 repeats, but as this was the higher value it could take in that study, it was inconclusive. However, in this it still showed preference for this value, as losses over 13 started to rise yet again, with some scarce exceptions. With this, the number of repeats for the first block was set at 12.

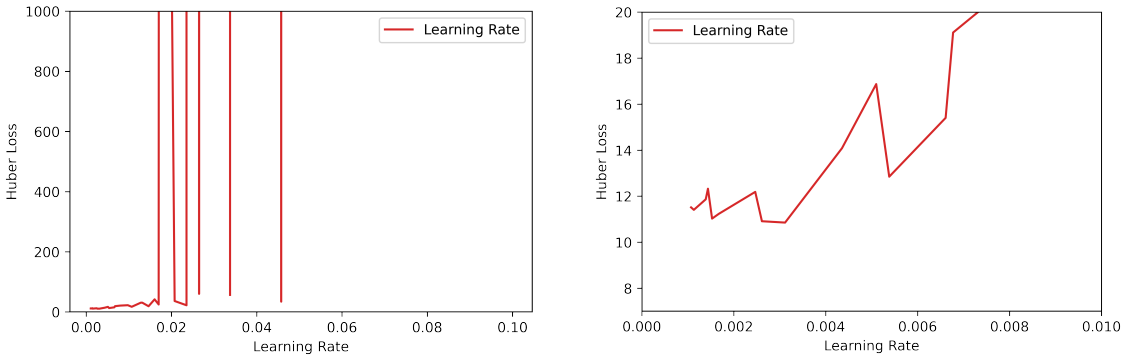
The last hyperparameter to analyze in this study would be the number of repeats for the second block, as it was also the less apparent. This was fixed at 7, mostly because the model showed some tendencies for the interval from 5 to 8 in Figure 5.18b, and the last runs in Study 5 (Figure 5.19b) where the best it had, all of them with 7 repeats. Compared to the previous hyperparameters, the variance in this one seemed to be least impactful, and was somewhat masked by the other more volatile variables. Was it not by the limited time available at this stage, a study with solely this variable could have proven helpful in setting it.

5.3.5 Studies 6 and 7 - Fixing Learning Rate

Like the in the previous tuning step, the last hyperparameter to set was the learning rate (Table 5.24). This was a simple study, with 50 trials.

Table 5.24: Variable hyperparameters in Study 6.

Hyperparameter	Symbol	Type
Learning Rate	—	float - $[1 \times 10^{-3},\ 1 \times 10^{-1}]$



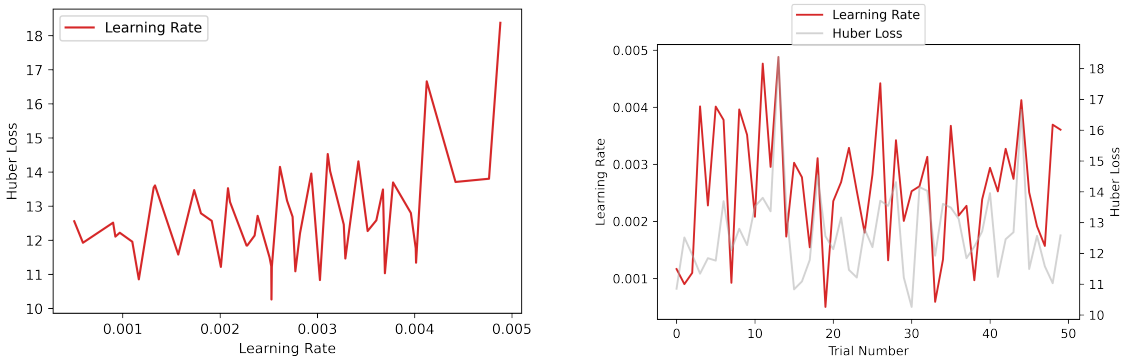
(a) Full Plot (b) Interest zone

Figure 5.21: Huber Loss per Learning Rate.

As it is fairly clear from Figure 5.21a, the analysis did not run for most of the interval. Many trials failed, as the model did not adapt to the dataset. This shows an increase in sensitivity from the architecture, especially when compared to Figure 5.2, showing the same analysis for the previous tuning step. However, it was possible to isolate an area with great results, seen in Figure 5.21b. Yet, this area had way too few trials to base any conclusion. For this reason, the study had to be repeated with this smaller area of interest in mind (Table 5.25).

Table 5.25: Variable hyperparameters in Study 7.

Hyperparameter	Symbol	Type
Learning Rate	—	float - $[5 \times 10^{-5},\ 5 \times 10^{-3}]$



(a) Huber loss per learning rate (b) Learning rate over trials

Figure 5.22: Learning Rate variance in Study 7.

In both plots in Figure 5.22 the learning rate shows unstable performance across the entire domain, with little to know relation between it and losses. This could be due to early stopping being to aggressively implemented with only 3 epochs before ceasing training, not giving enough opportunities for the model to converge. However, the learning rate ended up being fixed at $2.5 \times$

10^{-3}, as this had the lowest loss amongst all trials. However, patience will be increased in the final model.

5.4 Final Model Performance

Post re-tuning the model, and redoing the same Stratified K-Fold CV, the results obtained were as shown in Table 5.26 and Figure 5.23. Compared to the previous models, this come with a great increase in performance, but with an added computing cost, as the number of parameters increase considerably (Figure 5.24).

Table 5.26: Losses and Percentages of Test RVEs with MAPE bellow 10%, 5% and 2.5% for FinalCNN.

	Huber Loss	MAPE b. 10%	MAPE b. 5%	MAPE b. 2.5%	MAPE b. 1%
Fold 1	9.04	99.8%	98.3%	89.6%	53.3%
Fold 2	9.76	99.7%	97.9%	88.7%	49.3%
Fold 3	10.31	99.7%	97.7%	86.8%	43.7%
Fold 4	10.01	99.6%	97.7%	87.9%	47.6%
Fold 5	9.73	99.7%	98.0%	88.5%	49.6%
Fold 6	10.29	99.7%	97.9%	87.7%	41.9%
Fold 7	9.76	99.7%	98.0%	89.0%	47.9%
Fold 8	10.75	99.5%	97.6%	86.7%	43.5%
Fold 9	9.84	99.7%	97.8%	88.6%	50.0%
Fold 10	10.30	99.7%	97.8%	87.2%	43.9%
Average	9.98	99.7%	97.9%	88.1%	47.1%

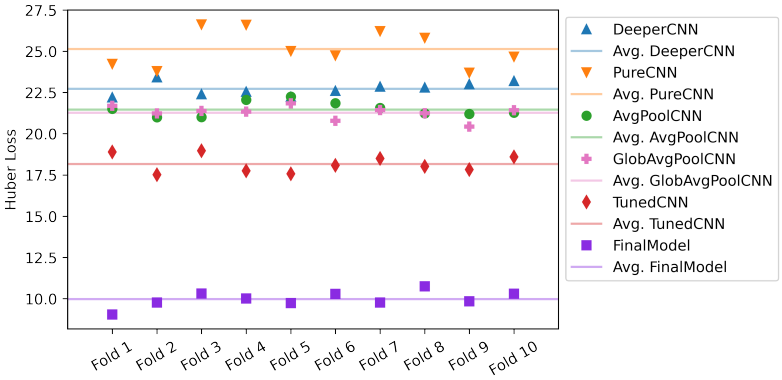


Figure 5.23: Huber losses across Folds for FinalModel.

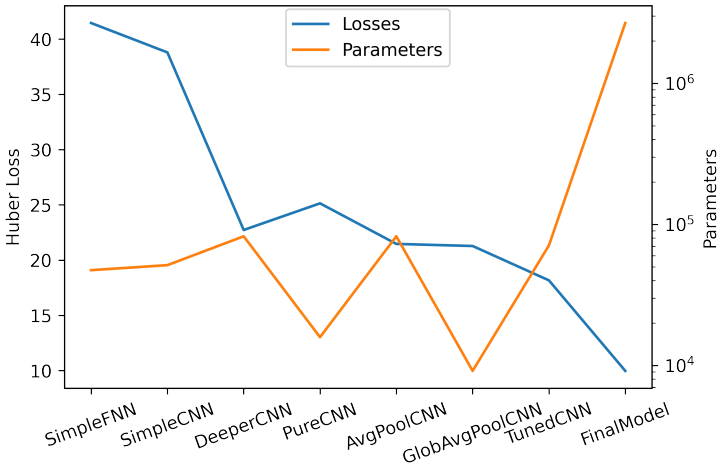
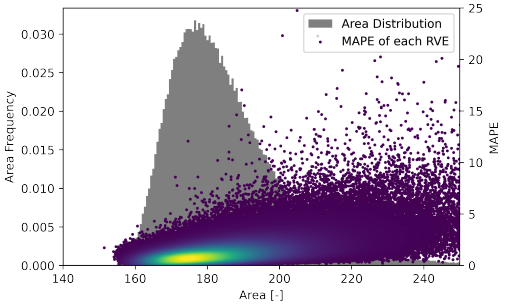
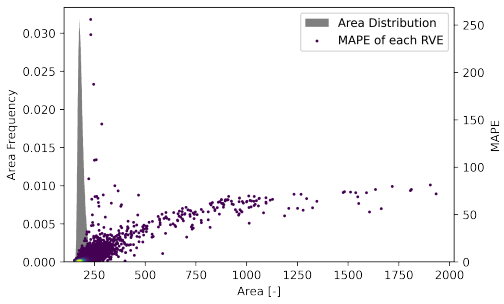


Figure 5.24: Evolution of Parameters and Losses up to Final Model.

On Figure 5.25 is plotted the relation between the MAPE for the model's prediction for each RVE with the area under the curve, with the color of the scattered RVEs representing the density of samples. In Figure 5.25a is obvious the great performance the model has for the vast majority of the curves, being those in the vicinity of the yellow zone. However, in Figure 5.25b one can identify two obvious outliers groups. Those with high areas under the curves (over 250) that represent an increase in MAPE to upwards of 70%. The other group is situated in between MAPE 90% to 250%. Worth reminding that each dot represents one of 300 000 samples, and has very little meaning to overall performance.



(a) Zone of interest



(b) Entire 300 000 RVEs dataset

Figure 5.25: Relation between model's prediction MAPE and area under the curve, with the color of each dot representing the density in said region.

The final takeaway from the final model is the processing time when compared to Links2. While Links2 needs and average of 5.2 seconds per RVE running on an Intel Core i7-10750H CPU, the machine learning model can process 1997 RVEs per second on average in the same machine. This makes the machine learning model 10384 times faster than FE^2, with very slight accuracy loss.

Chapter 6

Conclusions

Turning back to the question asked at the start of this work, is ML capable of capturing all necessary details to predict, with acceptable degrees of certainty, the mechanical behavior of complex microstructural arrangements? With the results obtained, and the models developed, one can conclude that yes, it can. There are still some problematic responses the models has trouble picking up, but as discussed in Section 3.5.2, some of those RVEs have displacements so out of the expected that one may even question the validity of the FE^2 analysis the AI is trying to predict.

The final model designed had vastly outperformed expectations, with 97.9% of test samples bellow 5% MAPE, with close to 90% under 2.5%. All of that at a fraction of a fraction of the time required to do the complete FE^2 run on that same RVE. This leads the way to other questions, such as "Can this methodology be applied to other loading conditions and materials?". The fact that they had such success for this application, hits they likely can. The models will need some adjustments, and the dataset will have to be started from scratch, but so far, it might very well be possible.

The most time-consuming aspect in the development of different models, for other loads and materials, would be the creation of a dataset capable of training them. However, the methodologies to optimize the generation that were tested and implemented, even the discarded attempt of bypassing Neper, could prove as excellent ways to accelerate this step. The FE^2 analysis was the heart of the dataset, and with the parallelization, acceptable speeds can be achieved even using under powered machines. With high-end processors, with more and faster cores, and using GPU acceleration, it would speed up this stage massively; and research in ways to bypass tessellation in mesh creation could prove crucial to obtain samples in under a second each. Still, the main problem with these datasets is their repetitiveness, as most curves are concentrated on a small area. Using methodologies to strategically remove entries that will not contribute to model accuracy and would only slow the training process down is something to be considered.

An evolution of the framework developed in this work, capable of model training and data generation, may very well be capable of aiding it the numerical analysis of components that would otherwise require multiple massive FE^2 calculations. If the ML model was tweaked to output not only the stress along one axis, but the whole stress tensor, in a fraction of the processing time

for each integration point across the entire macro mesh; it could possibly bypass the micro-scale model in most stages of component development. It would never fully replace the need for a complete FE^2 analysis, and one would always be needed to confirm results, the same way FEM calculations should be verified with live tests, but it has the potential to greatly accelerate the process.

Truth be told, there is still a long way to go, and a lot of research needs to be done in this field before an actual application of this technology is possible, but results obtained are promising. Seeing how the model would react to different datasets should now be the focus. Varying phase percentages and changing material properties would give an idea of were to take the framework, and what methods would prove useful to push these neural networks further.

6.1 Future Work

Even with this final results, the model still has room to grow. With both improvements to the dataset, and a finer tuning of the architecture, it should be possible to achieve even better results.

Developing technics to filter off both the outliers in the dataset, as well as redundant RVEs that provide nothing to training, could provide crucial to increase the dataset and diversify the RVEs that constitute it. Using clustering technics, it should be possible to identify and remove some samples from the dataset, and this could be worth exploring before going in any other direction.

Once this is in place, it would be beneficial to vary the percentages of each phase in the microstructure. Generating new RVEs and curves with more of one phase or the other should be advantageous as it would allow the model to be more flexible with its inputs. As is expected for the knowledge obtained from the 80% hard phase used thus far to bleed into the other percentages, those probably would not be needed in such large RVE quantities. However, only with some testing and careful stratification can this be confirmed or not.

Another different path this framework could be taken is by designing a non-sequential neural network, capable of accepting two inputs and combining it into a single output, as shown in Figure 6.1. This should allow the creation of a model capable of bypassing the microscale analysis within some load range. This would come with its own set of limitations and problems, and a new approach to data generation would also have to be taken, as the one used thus far only had one loading scenario. But if a CNN model is capable of capturing the features in a RVE and predicting a curve with the precision the final model in this work had, it is likely it will be capable of outputting the homogenized stress tensor for a given strain tensor as well. If the processing times are nearly as good as the ones obtained in this work, this might prove a valuable tool in the future of FE^2.

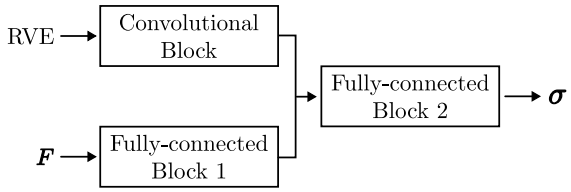


Figure 6.1: Schematic for a possible non-sequential neural network.

Appendix A

Bash for RVE Generation

```
#!/bin/bash
# Main control program
# Responsible for:
#   -Generating SEEDS
#   -Running neper_by_phase_modded
#   -Creating 2D_QUAD4_P.dat files for every seed
#   -Running LINKS2 for every seed in parallel
#   -Compressing files for ease of storage and movement
#
# From SEED to SEED:
FROM=103556
TO=153556
#
# ZIP name:
ZIPname="ResultsTower"
# Generating SEEDS array
SEEDS=($(seq $FROM 1 $TO))
#
# Writing Seeds to FILE .sd
fileSeeds="seeds.sd"
printf "" > $fileSeeds
for SEED in "${SEEDS[@]}"; do
    printf "$SEED\n">> $fileSeeds
done
#
# Running Neper (creates .rve files)
./neper_by_phase_modded.sh
```

```
#
# Creating 2D_QUAD4_P.dat files
fileorig="2D_QUAD4_P.dat"
linksloc="./LINKS-2.1.2/bin/LINKS2"
for SEED in "${SEEDS[@]}"; do
   fileloc="./neperRVE/2-phase_50-grain_qua8_${SEED}-seed_256-elem/2
      D_QUAD4_L.dat"
   cp $fileorig $fileloc
done
#
# Running LINKS2 in parallel
find ./neperRVE/ -name "2D_QUAD4_P.dat" | sort -V | parallel --
   progress --no-notice -j 4 --joblog ./Log.out mpiexec -n 2
   $linksloc | grep "local:"
echo "LINKS analysis completed"
#
# Zips output files for ease of moving and storing
echo "Compressing..."
zip -r ${ZIPname}.zip ./neperRVE/
```

Appendix B

Bash for Separating Useful Files

```
#!/bin/bash
cd raw
for d in */; do
    cd $d
    echo $d
    for e in */; do
        IFS="_"
        read -a delim <<< "$e"
        delim=${delim[3]}
        IFS="-"
        read -a delim <<< "$delim"
        seed=${delim[0]}
        IFS=" "
        IntermFolder="./interm/RVE_${seed}"
        echo "Isolating Seed: ${seed}"
        cd ../..
        mkdir -p $IntermFolder
        cp "./raw/${d}/${e}/2-phase_50-grain_qua8_256-elem_P.rve" "${IntermFolder}/2-phase_50-grain_qua8_256-elem_P.rve"
        cp "./raw/${d}/${e}/2D_QUAD4_L/2D_QUAD4_L_ELEM_1_GP_1.out" "${IntermFolder}/2D_QUAD4_L_ELEM_1_GP_1.out"
        cd raw/$d
    done
    cd ..
done
cd ..
```

Appendix C

Python for Compiling Dataframe

```
import os, os.path
import numpy as np
import pandas as pd
#
#Establishes Empty Dataframe
df = pd.DataFrame()
#
#Loops through every folder in ./interm and adds to DataFrame
root ='./interm'
for subdir, dirs, files in os.walk(root):
    for dir in dirs:
        NameRVE = f"./interm/{dir}/2-phase_50-grain_qua8_256-elem_P.
            rve"
        NameOUT = f"./interm/{dir}/2D_QUAD4_L_ELEM_1_GP_1.out"
        Seed = dir.split("_")[1]
        print(f"Adding Seed {Seed} to DataFrame")
        FileRVE = open(NameRVE, "r")
        domain = False
        Micro = np.empty(0)
        for index, line in enumerate(FileRVE):
            linha = line.strip()
            if domain == True:
                se=linha.split(" ")
                Micro = np.append(Micro, se[1])
            elif "ELEMENTS" in linha:
                domain = True
        Micro = Micro.astype(int)
```

```
        FileRVE.close()
        FileOUT = open(NameOUT, "r")
        Stress = np.empty(0)
        Strain = np.empty(0)
        AdmLoads = ["0.00000000E+00", "0.10000000E+00", "0.20000000E
           +00", "0.30000000E+00", "0.40000000E+00", "0.50000000E+00
           ", "0.60000000E+00", "0.70000000E+00", "0.80000000E+00",
           "0.90000000E+00", "0.10000000E+01"]
        for index, line in enumerate(FileOUT):
            linha = line.strip().split(" ")
            se = [x for x in linha if x != ""]
            LoadFac=se[1]
            if LoadFac in AdmLoads:
                Stress = np.append(Stress, float(se[20]))
                Strain = np.append(Strain, float(se[2])-1)
        FileOUT.close()
        if Stress.shape[0] == 11:
            data = {"Seed": Seed, "Microstructure": Micro, "Stress":
                Stress, "Strain": Strain}
            #df = df.append(data, ignore_index = True)
            df = pd.concat([df, pd.DataFrame([data])], ignore_index
               = True)
#
#Sort by seed
print("Sorting DataFrame")
df = df.sort_values(by=["Seed"],ignore_index=True)
#
#Saves pkl file
ID = len([name for name in os.listdir('./processed')]) + 1
#
#Counts how many pkl already exists
df.to_pickle(f"./processed/RVEdf_{ID}.pkl")
print(f"DataFrame saved to RVEdf_{ID}.pkl")
```

Appendix D

Maps for Elements ID in Neper and Python

240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

(a) Element ID in Neper and Link2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

(b) Element ID in Python Code.

Figure D.1: Mapping of Element ID in Neper/Link2 (a) and Python Code (b).

References

[1] Oñate Eugenio. *Structural Analysis with the Finite Element Method*. Springer, 2009.

[2] Vaz Junior Miguel, de Souza Neto Eduardo A., and Munoz-Rojas Pablo A. "Advanced Computational Materials Modeling: from Classical to Multiscale Techniques". In: Wiley-VCH, 2010. Chap. Variational foundations of large strain multiscale solid constitutive models: Kinematical formulation.

[3] Igor André Rodrigues Lopes. "Multi-Scale Modelling and Analysis of Multi-Phase Solids Using Second-Order Computational Homogenisation at Finite Strains with Parallel Computing". PhD thesis. Faculdade de Engenharia da Universidade do Porto, 2019.

[4] Jarosław Rzeszótko and Sinh Hoa Nguyen. "Machine Learning for Traffic Prediction". In: *Fundamenta Informaticae* 119 (2012), pp. 407–420. DOI: 10.3233/FI-2012-745.

[5] Luis A. Estrada Jiménez, Marco E. Benalcázar, and Nelson Sotomayor. "Gesture Recognition and Machine Learning Applied to Sign Language Translation". In: *VII Latin American Congress on Biomedical Engineering CLAIB 2016, Bucaramanga, Santander, Colombia, October 26th -28th, 2016*. Ed. by Isnardo Torres, John Bustamante, and Daniel A. Sierra. Singapore: Springer Singapore, 2017, pp. 233–236. ISBN: 978-981-10-4086-3.

[6] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142. DOI: 10.1109/MSP.2012.2211477.

[7] Jin-A Choi and Kiho Lim. "Identifying machine learning techniques for classification of target advertising". In: *ICT Express* 6.3 (2020), pp. 175–180. ISSN: 2405-9595. DOI: 10.1016/j.icte.2020.04.012. URL: https://www.sciencedirect.com/science/article/pii/S2405959520301090.

[8] Shubham Sharma, Archit Aggarwal, and Tanupriya Choudhury. "Breast Cancer Detection Using Machine Learning Algorithms". In: *2018 International Conference on Computational Techniques, Electronics and Mechanical Systems (CTEMS)*. 2018, pp. 114–118. DOI: 10.1109/CTEMS.2018.8769187.

[9] Google Brain. *TensorFlow*. URL: https://www.tensorflow.org/ (visited on 10/25/2022).

[10] Keras. *Keras: the Python deep learning API*. URL: https://keras.io/ (visited on 10/25/2022).

[11] *scikit-learn: machine learning in Python*. URL: https://scikit-learn.org/stable/ (visited on 10/25/2022).

[12] H. Kinsley and D. Kukieła. *Neural Networks from Scratch in Python: Building Neural Networks in Raw Python*. Harrison Kinsley, 2020.

[13] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[14] Charles Yang et al. "Prediction of composite microstructure stress-strain curves using convolutional neural networks". In: *Materials & Design* 189 (2020), p. 108509. ISSN: 0264-1275. DOI: 10.1016/j.matdes.2020.108509. URL: https://www.sciencedirect.com/science/article/pii/S0264127520300423.

[15] Yinli Wang et al. "Application of deep neural network learning in composites design". In: *European Journal of Materials* 2.1 (2022), pp. 118–171. DOI: 10.1080/26889277.2022.2053302.

[16] Romain Quey. *Neper: Polycrystal Generation and Meshing*. URL: https://neper.info/ (visited on 07/14/2022).

[17] R. Quey, P.R. Dawson, and F. Barbe. "Large-scale 3D random polycrystals for the finite element method: Generation, meshing and remeshing". In: *Computer Methods in Applied Mechanics and Engineering* 200.17 (2011), pp. 1729–1745. ISSN: 0045-7825. DOI: 10.1016/j.cma.2011.01.002.

[18] Romain Quey and Loïc Renversade. "Optimal polyhedral description of 3D polycrystals: method and application to statistical and synchrotron X-ray diffraction data". In: *Computer Methods in Applied Mechanics and Engineering* (Oct. 2017). URL: https://hal.archives-ouvertes.fr/hal-01626440.

[19] Chun-Teh Chen and Grace X. Gu. "Machine learning for composite materials". In: *MRS Communications* 9.2 (2019), pp. 556–566. DOI: 10.1557/mrc.2019.32.

[20] C. Soutis. "Carbon fiber reinforced plastics in aircraft construction". In: *Materials Science and Engineering: A* 412.1 (2005). International Conference on Recent Advances in Composite Materials, pp. 171–176. ISSN: 0921-5093. DOI: 10.1016/j.msea.2005.08.064.

[21] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. DOI: 10.48550/ARXIV.1502.03167. URL: https://arxiv.org/abs/1502.03167.

[22] Sergey Ioffe. *Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models*. 2017. DOI: 10.48550/ARXIV.1702.03275. URL: https://arxiv.org/abs/1702.03275.

[23] Charles Yang et al. “Using convolutional neural networks to predict composite properties beyond the elastic limit”. In: *MRS Communications* 9.2 (2019), pp. 609–617. DOI: 10.1557/mrc.2019.49.

[24] Benjamin Graham. *Fractional Max-Pooling*. 2014. DOI: 10.48550/ARXIV.1412.6071. URL: https://arxiv.org/abs/1412.6071.

[25] Takuya Akiba et al. “Optuna: A Next-generation Hyperparameter Optimization Framework”. In: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[26] Shashank Shekhar, Adesh Bansode, and Asif Salim. “A Comparative study of Hyper-Parameter Optimization Tools”. In: *CoRR* abs/2201.06433 (2022). arXiv: 2201.06433. URL: https://arxiv.org/abs/2201.06433.

[27] Inc. Preferred Networks. *Optuna: A Hyperparameter Optimization Framework*. URL: https://optuna.org/ (visited on 09/15/2022).

[28] Keras. *Layer weight initializers*. URL: https://keras.io/api/layers/initializers/ (visited on 09/15/2022).

[29] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[30] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*. 2012. DOI: 10.48550/ARXIV.1207.0580. URL: https://arxiv.org/abs/1207.0580.

[31] Keras. *RandomFlip layer*. URL: https://keras.io/api/layers/preprocessing_layers/image_preprocessing/random_flip/ (visited on 09/15/2022).

[32] Keras. *RandomTranslation layer*. URL: https://keras.io/api/layers/preprocessing_layers/image_preprocessing/random_translation/ (visited on 09/15/2022).