Simulating Stresses and Strains in Solid Mechanics Directly from Images Using Convolutional Neural Networks

BEATRIZ SUSANA VIEIRA

(Licenciatura em Engenharia Mecânica)

Dissertação para obtenção do grau de Mestre em Matemática Aplicada para a Indústria, na área de especialização de Tratamento de Dados

Orientadores:

Doutor José A. Rodrigues

Doutor Stéphane P. A. Bordas

Júri:

Presidente: Doutor Luís Silva

Vogais:

Doutor José Leonel Rocha

Doutor José A. Rodrigues

Setembro de 2025

Simulating Stresses and Strains in Solid Mechanics Directly from Images Using Convolutional Neural Networks

BEATRIZ SUSANA VIEIRA

(Licenciatura em Engenharia Mecânica)

Dissertação para obtenção do grau de Mestre em Matemática Aplicada para a Indústria,

na Área de Especialização de Tratamento de Dados

Orientador(es):

Doutor José A. Rodrigues, CIMA, ISEL

Doutor Stéphane P. A. Bordas, Legato, UL

Júri:

Presidente: Doutor Luís Silva, CIMA, ISEL

Vogais:

Doutor José Leonel Rocha, CEAUL, ISEL

Doutor José A. Rodrigues, CIMA, ISEL

Setembro de 2025

Agradecimentos

Gostaria de expressar a minha mais profunda gratidão aos meus orientadores, Professor José A. Rodrigues e Professor Stéphane Bordas, pela sua orientação inestimável, encorajamento e paciência ao longo do desenvolvimento desta dissertação. A sua experiência e o seu feedback esclarecedor foram essenciais para a conclusão bem-sucedida deste trabalho.

Agradeço também, sinceramente, ao Departamento de Matemática do Instituto Superior de Engenharia de Lisboa (ISEL) por proporcionar o ambiente académico e técnico que tornou esta investigação possível.

Um agradecimento especial à minha família e aos meus amigos pelo apoio constante, compreensão e incentivo durante os períodos mais exigentes do meu percurso académico. A confiança que depositaram em mim foi uma fonte de força e motivação.

Por fim, sou grata a todos os que, de uma forma ou de outra, contribuíram para este projeto, seja através de discussões, colaboração ou apoio moral. Este trabalho não teria sido possível sem as vossas contribuições.

Statement of integrity

I declare that this dissertation is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

Lisbon, October 22, 2025

Resumo

A análise de deslocamentos, deformações e tensões com o Método dos Elementos Finitos (FEM) é fiável, mas torna-se lenta quando é preciso testar muitas combinações de geometria e carregamento. Em vários contextos a informação inicial chega sob a forma de imagem e dá origem a malhas que não são retangulares, o que dificulta o uso de modelos que trabalham apenas em grelha. Este trabalho estuda modelos substitutos baseados em aprendizagem profunda que usam soluções FEM como referência e que permitem obter o campo mecânico de forma muito mais rápida.

Apresenta-se uma cadeia de processamento completa que parte da segmentação da imagem, constrói malhas estruturadas e não estruturadas e, sobre elas, treina dois modelos distintos. O primeiro é o modelo U-Net em grelha, pensada para domínios retangulares. O segundo é o modelo MAgNET, que opera diretamente na malha e preserva a discretização original nas fronteiras e nas zonas onde a força é aplicada. Ambos os modelos são treinados e avaliados exatamente com o mesmo conjunto de dados, o mesmo plano de treino e as mesmas métricas, incluindo análises com forças acima do intervalo usado no treino e medições do tempo de treino e de inferência, o que permite uma comparação justa entre as abordagens.

Os resultados mostram que os dois modelos reproduzem bem o deslocamento e que os maiores erros de deformação e tensão aparecem junto ao carregamento e nas zonas de maior gradiente. O modelo U-Net é mais rápida e muito competitiva em malhas regulares; o modelo MAgNET é preferível quando a geometria vem de imagem e a malha não é regular.

Palavras-chave U-Net; segmentação; processamento de imagem; MAgNET; método dos elementos finitos.

Abstract

Finite Element Method (FEM) simulations provide reliable displacement, strain and stress fields, but they become costly when many geometry–load combinations must be tested. In several practical scenarios the input is an image that leads to non-rectangular, unstructured meshes, which is not ideal for strictly grid-based models. This dissertation investigates deep learning surrogates trained on FEM solutions that can deliver the mechanical response much faster.

We propose a complete processing pipeline that starts from image segmentation, builds both structured and unstructured meshes, and trains two distinct models on top of them. The first one is a grid U-Net, designed for rectangular domains. The second one is MAgNET, which operates directly on the mesh and preserves the original discretization at the boundaries and at the loaded regions. Both models are trained and evaluated with exactly the same dataset, training schedule and metrics, including tests with loads above the training range and measurements of training and inference time, which enables a fair comparison between the two approaches.

Results show that both surrogates reproduce displacement accurately, and that the largest strain and stress errors remain confined to the loaded boundary and to high-gradient areas. The grid U-Net is faster and very competitive on regular meshes, while MAgNET is the better option when the geometry comes from images and the mesh is unstructured.

Keywords U-Net; segmentation; image processing; MAgNET; finite element method.

Table of Contents

List of Figures

List of Tables

Symbols and abbreviations

Symbology

Latin

Symbol	Meaning
Solid mechanics / FEM	
Ω	Computational domain (elastic body)
$\partial\Omega$	Boundary of the domain
$\mathbf{n}$	Outward unit normal on $\partial\Omega$
$\mathbf{u}$	Displacement field (trial function)
$\mathbf{v}$	Virtual displacement / test function
$\boldsymbol{\varepsilon}$	Infinitesimal strain tensor
$\boldsymbol{\sigma}$	Cauchy stress tensor
E	Young's modulus
ν	Poisson's ratio
μ	Shear modulus
λ	First Lamé parameter
$\mathrm{tr}(\cdot)$	Matrix trace
$\mathbf{I}$	Identity tensor
$\mathbf{f}$	Body-force density (per unit volume)
$\bar{\mathbf{t}}$	Prescribed traction on Γ_N
$U,\ V$	Trial and test spaces in the weak form
∇	Gradient operator
$\nabla\cdot$	Divergence operator
N_i	P1 (linear) shape function of node i
$\mathbf{B}$	Strain–displacement matrix
$\mathbf{d}_e$	Element nodal degrees of freedom

Symbol	Meaning
$\mathbf{K}^e$	Element stiffness matrix
$\mathbf{f}^e$	Element load vector
$\mathbf{K},\ \mathbf{d},\ \mathbf{f}$	Global stiffness matrix, displacement vector, and load vector
τ_{xy}	Engineering shear stress component
γ_{xy}	Engineering shear strain (convention)
CNN / learning	
$\mathbf{X}$	Input feature vector (image/forces)
Y	Target / label
Z_m	Hidden feature m
T_k	Output pre-activation (unit k)
$f_k(\mathbf{X})$	Network output (unit k)
$\sigma(\cdot),\ g(\cdot)$	Activation functions (hidden / output)
$\mathbf{w}_m,\ b_m$	Weights and bias of hidden unit m
$\beta_k,\ \beta_{0k}$	Weights and bias of output unit k
$N,\ M,\ K,\ p$	Number of samples, hidden units, classes, input dimension
D_{tr}	Training dataset
R	Empirical risk (training objective)
ℓ	Per-sample loss
$t_{\text{train}},\ t_{\text{infer}}$	Training time; inference (prediction) time

Greek

Symbol	Meaning
Solid mechanics / FEM	
$\varepsilon_x,\ \varepsilon_y,\ \gamma_{xy}$	In-plane strain components
$\sigma_x,\ \sigma_y,\ \tau_{xy}$	In-plane stress components
$\mu,\ \lambda$	Lamé parameters
ν	Poisson's ratio
$\Gamma_D,\ \Gamma_N$	Dirichlet and Neumann boundary parts
Optimization / learning	
α	Learning rate

Symbol	Meaning
θ	Trainable parameters of the network
θ^*	Optimal (trained) parameters
β	Momentum coefficient
$\beta_1,\ \beta_2$	Adam decay rates
$m_t,\ v_t$	First and second moment estimates (Adam)
$\hat{m}_t,\ \hat{v}_t$	Bias-corrected moment estimates (Adam)
ρ	RMSProp decay parameter
ϵ	Numerical stability constant
$\sigma(\cdot)$	Sigmoid activation (context: ML)
Γ	Boundary symbol used in $\Gamma_D,\ \Gamma_N$

Abbreviations FEM: Finite Element Method; DoF: Degree of freedom; CNN: Convolutional Neural Network; MSE: Mean squared error; MAgNET: Mesh-Graph Network.

Chapter 1

Introduction

1.1 Introduction

The study of how materials respond to external forces is one of the fundamental pillars of engineering and applied science. The ability to accurately determine where and how stresses and strains develop within a structure is critical to ensuring its safety, performance, and durability. Structural failures, whether in bridges, aircraft, medical implants, or industrial machinery, can have catastrophic consequences, resulting in economic loss, environmental damage, or even human casualties. For this reason, stress and strain analysis lies at the core of modern design, maintenance, and certification processes. Stress measures how a material resists external forces, while strain shows how much it deforms under those forces [1]. These concepts are key to ensuring the strength and reliability of engineered systems.

Historically, engineers approached this challenge using analytical methods derived from the principles of continuum mechanics. These methods, grounded in well-established mathematical formulations, provided elegant and exact solutions for a limited set of geometries and loading conditions [10]. Classic problems, such as the bending of a simply supported beam or the torsion of a circular shaft, could be solved with closed-form expressions, offering both physical insight and predictive capability [10]. However, as designs became more complex and materials more advanced, these analytical solutions reached their practical limits. Real-world engineering problems often involve irregular geometries, non-linear material behavior, and complex boundary conditions, making analytical approaches infeasible.

The need for a more general and flexible framework led to the development of numerical methods in the mid-20th century. Among these, the Finite Element Method (FEM) quickly became the standard tool for structural and solid mechanics analysis. Originating in the aerospace industry in the 1950s and 1960s [2], FEM provided a systematic way to approximate solutions to

differential and integral equations governing material behavior [2]. By discretizing a domain into smaller, manageable elements connected through nodes in a mesh, FEM allowed engineers to model virtually any shape or physical phenomenon with remarkable accuracy [3].

Over the decades, FEM has matured into a highly versatile technique, capable of simulating linear and nonlinear elasticity, plasticity, fracture, heat transfer, fluid–structure interaction, and more. It is now a mainstay in commercial engineering software and is used across industries ranging from civil and mechanical engineering to biomedical device design. However, FEM is not without its challenges.

The accuracy of FEM simulations depends heavily on the quality of the input data, including precise geometry, reliable material properties, and well-defined loading and boundary conditions [3]. Assumptions or errors in these inputs can propagate through the analysis, leading to misleading results. Furthermore, as the complexity or resolution of a model increases, so too does the computational cost [3]. Large-scale FEM analyses, especially those involving fine meshes or nonlinear material models, can require hours or days of computation on high-performance computing systems. While modern FEM software offers user-friendly interfaces, effective use still demands a deep understanding of both the physics and the numerical methods involved [3].

In parallel with these developments, the past two decades have seen machine learning move to the center of scientific computing. In computational mechanics, deep learning tends to appear in three strands that differ in how they use data and physics. One strand is purely data driven and learns supervised mappings from experimental measurements or high-fidelity simulations without explicitly enforcing the governing equations. Another strand is physics-informed neural networks (PINNs), where the training objective includes residuals of the PDE and boundary conditions, which can lower the need for labeled data and improve generalization, although performance depends on loss weighting and the placement of collocation points [4]. A third strand learns the solution operator itself so that predictions transfer across resolutions and meshes, and physics-aware variants blend operator learning with PDE guidance [5, 6]. In this thesis we remain within the supervised data-driven setting and use FEM solutions as ground truth to keep comparisons to the FEM reference clean and controlled. Within this strand we consider grid-based CNNs and mesh-native encoder–decoder networks that act directly on unstructured meshes in the spirit of MAgNET [7].

Within this landscape, convolutional neural networks have become especially influential in computer vision and pattern recognition and they translate naturally to mechanics problems on regular grids. Stacked convolutions learn hierarchical spatial features and exploit local correlations, which suits images and regularly sampled fields [9]. To handle unstructured discretizations we

step beyond grids and adopt mesh-native encoder–decoder networks that pool and unpool directly on the graph of the mesh. The MAgNET architecture is an example that brings the spirit of U-Net to finite-element-style meshes and enables fast single-pass inference on the native mesh, with no grid-mapping step [7].

One particularly promising direction is the direct estimation of deformation from images. Using experimental data from imaging techniques such as digital image correlation (DIC), microscopy, or high-resolution photography, CNNs can be trained to map visual information about a material or structure directly to its mechanical response. The potential impact of such an approach is far-reaching: in engineering design, rapid feedback could enable iterative testing of concepts without the computational burden of repeated FEM runs; in structural health monitoring, real-time image analysis could detect emerging deformations or localized stress concentrations before failure occurs; in materials science, microstructural images could be directly linked to predicted performance, aiding in the development of new materials.

In this work, we investigate the use of CNNs as a data-driven alternative to FEM for predicting stresses and strains in solid mechanics problems. We focus particularly on the problem of predicting deformation directly from images, comparing the accuracy, efficiency, and generalization capability of CNN-based approaches against conventional FEM simulations. By exploring the strengths and limitations of both methods, our goal is to assess whether CNNs can complement, or in some cases replace, FEM in practical engineering applications, thereby contributing to faster, more adaptive, and more efficient mechanical analysis workflows.

This thesis is guided by a small set of questions that frame both the modelling choices and the evaluation protocol. The first asks whether a grid based model can rival a mesh native model when the geometry departs from rectangular domains and sharp gradients appear near boundaries and features. The second asks how performance shifts with mesh density and with load amplitude, both inside and outside the training range. The third asks what inference speedups are attainable over the finite element reference while keeping relative L^2 error and energy error within practical engineering limits.

The contributions follow directly from these questions. The work delivers an image to solution pipeline that covers segmentation, mesh generation on both structured and unstructured domains, supervised training against FEM solutions and consistent post processing of fields into strains and stresses. It establishes a controlled comparison between U-Net and MAgNET under the same data, training schedule and metrics across rectangular and irregular shapes. It quantifies regimes of preference by analysing sensitivity to mesh density, segmentation quality and load scaling, including tests outside the training range. It reports training and inference times and

memory footprints with identified hardware and summarises accuracy through relative L^2 error, energy discrepancy and error maps that expose where mistakes concentrate in the field.

1.2 Structure and goals of the report

This report investigates how modern neural networks can complement or accelerate the Finite Element Method (FEM) in solid mechanics. The central task is to learn the forward map from external loads to deformations and to recover strains and stresses a posteriori. In addition to accuracy with respect to FEM ground truth, the study emphasizes inference efficiency and robustness within an image-to-solution workflow.

A practical aim is deformation from an image: starting with an image of the domain, we extract geometry, generate a mesh, and predict displacement, strain, and stress fields under specified boundary conditions. Two complementary surrogate families are developed to support this pipeline. The first is grid-based, using a convolutional U-Net that consumes grid-mapped tractions. The second is mesh-native, using a graph U-Net (MAgNET) that operates directly on unstructured meshes and produces node-wise predictions with no grid mapping. Training and evaluation are aligned against a common FEM baseline so that observed differences reflect architectural choices rather than data handling.

The document proceeds from principles to practice. It first establishes the FEM reference, covering governing equations, weak formulation, spatial discretization, assembly, and the recovery of strain and stress in plane stress. It then reviews the deep-learning components used throughout—optimization, losses, regularization, and the encoder–decoder pattern underlying U-Net. Building on this foundation, two surrogate pipelines are presented: a grid-based U-Net for force-to-displacement on structured grids, and a mesh-based MAgNET that replaces image convolutions with graph aggregation and static pooling/unpooling so that predictions remain on the original mesh. The experimental section unifies setups, training schedules, and metrics to compare both surrogates against FEM on matched cases, followed by a discussion of limitations, practical guidance on when to favor each approach, and implementation notes to support reproducibility.

The results in this file focus specifically on magnitude extrapolation of the applied load while holding geometry, mesh, and boundary conditions fixed. Models are trained on forces within the nominal range defined for each dataset and are then evaluated on an out-of-range case at 30N, applied in the same direction and on the same boundary as in training. No alternative meshes, mesh resolutions, or boundary-condition patterns are introduced here; the goal is to isolate the effect of load magnitude on prediction quality. Results follow the same visualization protocol

and metrics used elsewhere in the report—field maps and absolute error maps for displacement, strain, and stress, together with global relative L^2 errors—to enable a like-for-like comparison focused on force-magnitude variation.

Chapter 2

Finite Element Method

The Finite Element Method is a powerful numerical technique for approximating the solutions of partial differential equations (PDEs) that govern many problems in engineering and physics. It has become a standard tool in structural analysis, heat transfer, fluid dynamics, and electromagnetics, particularly in cases where complex geometries, heterogeneous materials, or non-trivial boundary conditions make analytical solutions impractical [12, 13, 14].

The central idea of FEM is to subdivide the problem domain into smaller, simpler subdomains called finite elements, which are connected at specific points known as nodes. Together, these elements form a mesh that approximates the entire structure [12, 13], as illustrated in Figure 2.1. Within each element, the unknown field variable (such as displacement, temperature, or potential) is approximated using interpolation functions. This local approximation transforms the original continuous problem into a discrete system of algebraic equations that can be solved numerically.

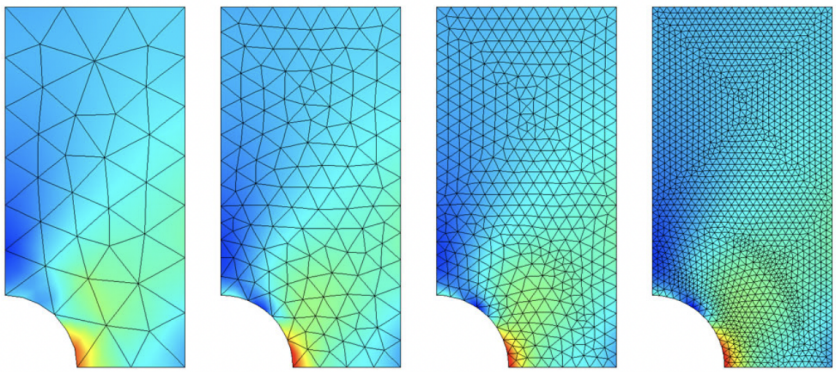


Figure 2.1: Example of stresses in a plate with a hole, solved with different element sizes (source: [11]).

A complete finite element analysis typically consists of three main stages [12, 13]:

- Pre-processing: definition of the geometry, assignment of material properties, mesh generation, and specification of boundary conditions and loads;

- Solution: formulation and assembly of the element stiffness matrices into a global system of equations, application of boundary conditions, and computation of the nodal values of the unknowns;
- Post-processing: evaluation of derived quantities such as stresses and strains, visualization of results, and comparison with theoretical predictions or experimental data.

The following sections provide the theoretical foundations of FEM, including the governing equations of solid mechanics, the weak formulation of the problem, and the discretization process that leads to the system of algebraic equations. The chapter concludes with a discussion of accuracy, computational cost, and the role of FEM in modern engineering analysis.

2.1 Governing Equations in Solid Mechanics

Solid mechanics is a branch of applied mechanics concerned with the behavior of solid bodies under the action of external loads, temperature variations, or other physical effects [13, 14]. The main objectives are to determine the deformation of the structure, the internal stresses that develop, and the strains experienced by the material. Understanding these quantities is essential for predicting structural performance, ensuring safety against failure, and optimizing design. In what follows, the quantities are introduced in the usual order: displacement, then strain (obtained from displacement), and finally stress (which represents the internal reaction).

Deformation refers to the change in shape or size of a body when subjected to forces or other influences. It can include both rigid-body motion, where the body changes its position without altering its shape, and true deformation, where the geometry of the body changes [13]. To describe this mathematically, we introduce the displacement vector field. For a point $\mathbf{x}$ in the undeformed configuration, the displacement vector is

$$\mathbf{u}(\mathbf{x}) = \begin{bmatrix} u_x(\mathbf{x}) \\ u_y(\mathbf{x}) \\ u_z(\mathbf{x}) \end{bmatrix}, \tag{2.1}$$

where u_x, u_y, and u_z represent the displacement components in the x, y, and z directions, respectively [13]. Each component has units of length (m). This field specifies the new position of every material point but, by itself, it does not yet distinguish between a pure rigid motion and an actual deformation; that distinction appears when spatial variations of $\mathbf{u}$ are examined.

Once the displacement field is known, the deformation of the body can be quantified through strain, which measures the relative displacement of neighbouring material points. For small

deformations in one dimension, the normal strain is

$$\varepsilon = \frac{\Delta L}{L}, \tag{2.2}$$

where L is the original length and ΔL is the change in length [13]. This expression shows that strain is dimensionless (ratio of two lengths) and that it measures change, not absolute size.

In three dimensions, strain is represented by the infinitesimal strain tensor

$$\boldsymbol{\varepsilon}(\mathbf{u}) = \frac{1}{2}\left[\nabla\mathbf{u} + (\nabla\mathbf{u})^{\mathrm{T}}\right], \tag{2.3}$$

which corresponds to the symmetric part of the displacement gradient [13, 14]. Here $\nabla\mathbf{u}$ denotes the matrix of first spatial derivatives of the displacement components. Taking only the symmetric part removes the purely rotational contribution and retains the part of the motion that actually stretches or shears the material. This small-strain formula is valid when displacement gradients and rotations are sufficiently small, which is the regime considered in the FEM simulations used later in this work.

Stress complements strain by quantifying the internal forces that develop as the body resists deformation. At its most basic level, stress is defined as force per unit area,

$$[\sigma] = \mathrm{N/m^2}, \tag{2.4}$$

with SI unit of Pascal (Pa) [13]. In engineering applications, stresses are commonly expressed in megapascals (MPa), where $1\,\mathrm{MPa} = 10^6\,\mathrm{Pa}$. More formally, if ΔA is a differential area inside the body, the intensity of the internal force acting on this area is

$$\sigma = \lim_{\Delta A \to 0} \frac{\Delta F}{\Delta A}, \tag{2.5}$$

a definition due to Cauchy [13]. This limit indicates that stress is a local measure: it is defined for a point and for a given plane passing through that point.

If the force acts perpendicular to ΔA, the stress is called normal stress. A tensile stress occurs when the force pulls on the area, while a compressive stress occurs when it pushes. If the force acts tangentially to ΔA, the stress is called shear stress, with components such as

$$\tau_{zx} = \lim_{\Delta A \to 0} \frac{\Delta F_x}{\Delta A}, \qquad \tau_{zy} = \lim_{\Delta A \to 0} \frac{\Delta F_y}{\Delta A}, \tag{2.6}$$

which represent, respectively, force in the x- or y-direction acting on a face whose normal is the

z-direction.

By considering planes aligned with the coordinate axes, the general three-dimensional state of stress at a point can be defined. This is represented mathematically by the Cauchy stress tensor:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix}, \tag{2.7}$$

where σ_{ii} are the normal stresses and σ_{ij}, $i \neq j$, are the shear stresses. For bodies in static equilibrium without body couples, the stress tensor is symmetric, i.e. $\sigma_{ij} = \sigma_{ji}$ [13, 14]. This symmetry follows from balance of angular momentum and is later used to reduce the number of unknowns in numerical formulations.

The way a material resists deformation under applied loads is commonly described in terms of its stiffness. At the material level, stiffness refers to the proportionality between stress and strain [13]. For instance, in uniaxial tension the ratio of normal stress to normal strain is defined as Young's modulus E (with units Pa), which measures resistance to stretching or compression. Similarly, the shear modulus G (also in Pa) characterizes resistance to shear deformation. At the structural level, stiffness is often expressed as the ratio of applied force to the corresponding displacement (with units N/m), which in the finite element method is represented by the stiffness matrix [12, 13].

To establish a relationship between stress and strain, assumptions about material behavior must be introduced. Real engineering materials can exhibit complex responses such as plasticity, anisotropy, or viscoelasticity, but for many structural applications a simplified model is sufficient. The classical formulation of Hooke's law relies on three assumptions: homogeneity, isotropy, and linear elasticity [13]. A material is said to be homogeneous if its mechanical properties are identical at every point in the body; isotropic if its properties are the same in all directions at a given point; and linearly elastic if the relationship between stress and strain is linear and the material returns to its original configuration upon unloading, provided stresses remain within the elastic limit [12, 13]. These are the assumptions adopted in this work so that the FEM solutions are well defined and can be used as reference data for training and validating the neural surrogates.

Under these assumptions, the stress–strain relation is described by Hooke's law of linear

elasticity. In three dimensions, the normal strains are related to the normal stresses by

$$\varepsilon_x = \frac{1}{E}\left(\sigma_x - \nu(\sigma_y + \sigma_z)\right), \tag{2.8a}$$

$$\varepsilon_y = \frac{1}{E}\left(\sigma_y - \nu(\sigma_x + \sigma_z)\right), \tag{2.8b}$$

$$\varepsilon_z = \frac{1}{E}\left(\sigma_z - \nu(\sigma_x + \sigma_y)\right), \tag{2.8c}$$

and the shear strains are related to the shear stresses by

$$\gamma_{xy} = \frac{\tau_{xy}}{G}, \quad \gamma_{yz} = \frac{\tau_{yz}}{G}, \quad \gamma_{zx} = \frac{\tau_{zx}}{G}, \tag{2.8d}$$

where the shear modulus is given by

$$G = \frac{E}{2(1+\nu)}. \tag{2.9}$$

For many engineering applications, two-dimensional approximations are used. Hooke's law then admits two common specializations depending on the boundary conditions [12, 13]. When the out-of-plane stresses vanish ($\sigma_z = \tau_{xz} = \tau_{yz} = 0$), the in-plane stresses and strains are related by the plane stress constitutive matrix:

$$\begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{bmatrix}. \tag{2.10}$$

Similarly, when the out-of-plane strains are constrained to zero ($\varepsilon_z = \gamma_{xz} = \gamma_{yz} = 0$), the relation is given by the plane strain constitutive matrix:

$$\begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{bmatrix}. \tag{2.11}$$

Thus, Hooke's law provides the fundamental link between stress and strain in both three-dimensional and two-dimensional cases. The plane stress and plane strain formulations are particularly important in finite element modeling of thin structures and constrained bodies, respectively [12, 13].

Together, the concepts of displacement, strain, stress, stiffness, and constitutive relations form

the basis of solid mechanics. They provide the essential framework from which the governing equations of equilibrium and numerical formulations, such as the Finite Element Method, are developed.

2.2 Weak Formulation

The equilibrium equations of linear elasticity in strong form require that the divergence of the stress tensor balances the body forces at every point in the domain. While exact, this formulation is not directly suitable for finite element discretization, as it requires higher regularity of the displacement field than can be achieved by piecewise-polynomial approximations. To overcome this difficulty, one adopts the weak (variational) formulation, which enforces equilibrium in an integral sense and relaxes differentiability requirements [13, 14, 15, 16].

The geometric setting considered throughout this section, including the domain Ω, the Dirichlet region Γ_D, and the Neumann region Γ_N, is illustrated in Fig. 2.2.

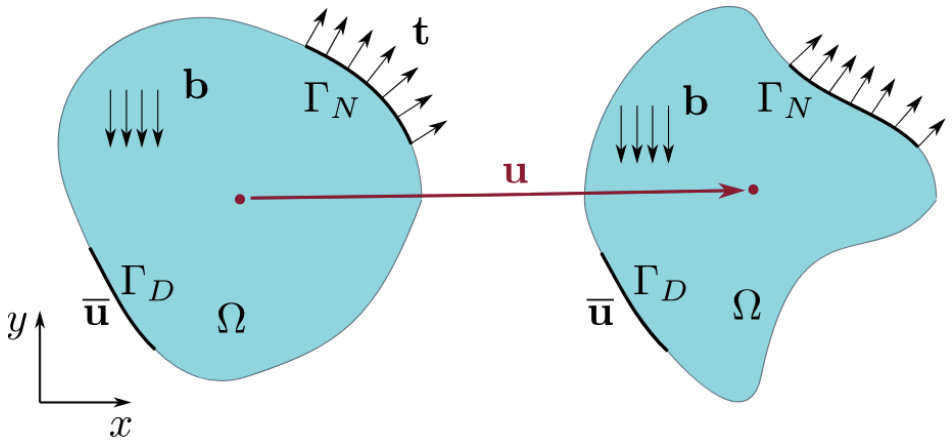


Figure 2.2: Elastic body Ω with Dirichlet boundary Γ_D, Neumann boundary Γ_N, body force $\mathbf{b}$, surface tractions $\mathbf{t}$, and displacement field $\mathbf{u}$ (source: [44].

Let $\Omega \subset \mathbb{R}^2$ denote the elastic body, with boundary $\partial\Omega = \Gamma_D \cup \Gamma_N$. The strong form reads

$$\nabla \cdot \boldsymbol{\sigma} + \mathbf{f} = \mathbf{0} \quad \text{in } \Omega, \tag{2.12}$$

where $\mathbf{u}$ is the displacement field, $\boldsymbol{\sigma}$ the Cauchy stress tensor, and $\mathbf{f}$ the body force per unit volume [13, 14]. For a homogeneous, isotropic, linearly elastic material, Hooke's law provides the constitutive relation

$$\boldsymbol{\sigma}(\mathbf{u}) = \lambda \, \mathrm{tr}(\boldsymbol{\varepsilon}(\mathbf{u}))\mathbf{I} + 2\mu \, \boldsymbol{\varepsilon}(\mathbf{u}), \tag{2.13}$$

with Lamé parameters

$$\mu = \frac{E}{2(1+\nu)}, \tag{2.14a}$$

$$\lambda = \begin{cases} \dfrac{E\nu}{(1+\nu)(1-2\nu)}, & \text{plane strain,} \\ \dfrac{E\nu}{1-\nu^2}, & \text{plane stress,} \end{cases} \tag{2.14b}$$

and infinitesimal strain tensor

$$\boldsymbol{\varepsilon}(\mathbf{u}) = \tfrac{1}{2}(\nabla\mathbf{u} + (\nabla\mathbf{u})^\top). \tag{2.15}$$

For later reference, in plane stress the parameters are explicitly

$$\mu = \frac{E}{2(1+\nu)}, \qquad \lambda = \frac{E\nu}{1-\nu^2}. \tag{2.16}$$

The weak form is obtained from the principle of virtual work, which states that the virtual work of the internal stresses equals the virtual work of the external forces for any admissible virtual displacement $\mathbf{v}$ [12, 13]. The variational problem is therefore: find

$$\mathbf{u} \in \mathcal{U} = \{\mathbf{u} \in [H^1(\Omega)]^2 : \mathbf{u} = \bar{\mathbf{u}} \text{ on } \Gamma_D\} \tag{2.17}$$

such that

$$\int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v})\, dx = \int_\Omega \mathbf{f}\cdot\mathbf{v}\, dx + \int_{\Gamma_N} \bar{\mathbf{t}}\cdot\mathbf{v}\, ds, \qquad \forall \mathbf{v} \in \mathcal{V}, \tag{2.18}$$

where

$$\mathcal{V} = \{\mathbf{v} \in [H^1(\Omega)]^2 : \mathbf{v} = \mathbf{0} \text{ on } \Gamma_D\} \tag{2.19}$$

is the space of admissible test functions. The choice of the Sobolev space $H^1(\Omega)$ reflects the fact that only square-integrable functions with square-integrable first derivatives are required. This makes finite element approximations feasible, since displacements remain continuous across elements while derivatives may be discontinuous [14, 15, 16].

2.3 Boundary Conditions

Boundary conditions supplement the weak formulation by prescribing either displacements or forces on the boundary of the domain. These fall into two complementary categories [13, 14].

Dirichlet, or essential, boundary conditions prescribe the displacement on Γ_D, written as $\mathbf{u} = \bar{\mathbf{u}}$ on Γ_D. In the finite element setting, they are enforced directly on the nodal degrees of freedom, either by eliminating the corresponding unknowns or by modifying the global system of

equations to impose the prescribed values. A sufficient number of displacement constraints is required to eliminate rigid-body motions; otherwise, the global stiffness matrix remains singular and the system admits no unique solution [12, 13].

Neumann, or natural, boundary conditions prescribe tractions on Γ_N, expressed as $\boldsymbol{\sigma}\mathbf{n} = \bar{\mathbf{t}}$. These enter the weak form naturally through boundary integrals and, after discretization, contribute to the global load vector. In practice, distributed tractions are converted into equivalent nodal forces using the element shape functions, while concentrated forces may be applied directly to the relevant nodal entries of the load vector [12, 13].

Together, Dirichlet and Neumann boundary conditions ensure that the finite element model is both mathematically well-posed and physically consistent. With the variational formulation and boundary conditions in place, one can proceed to discretize the problem using finite element shape functions [14].

2.4 Finite Element Discretization

To obtain a computable approximation, the infinite-dimensional weak problem is restricted to a finite-dimensional subspace spanned by interpolation functions, also called shape functions [13, 14]. The domain Ω is partitioned into finite elements, each with n nodes, and the displacement field inside each element is approximated as

$$\mathbf{u}_h(x, y) = \sum_{i=1}^{n} \mathbf{u}_i \, N_i(x, y), \tag{2.20}$$

where $N_i(x, y)$ are the shape functions associated with node i, and $\mathbf{u}_i = (u_{x,i}, u_{y,i})^\top$ are the nodal displacement values [13, 12]. For first-order triangular elements ($P1$), the shape functions are linear, so that the displacement varies linearly across the element and is continuous at the nodes. This guarantees compatibility of displacements across elements while allowing strains and stresses to be piecewise discontinuous, which is acceptable in the weak formulation [13, 14].

From this approximation, the strain field inside an element is expressed as

$$\boldsymbol{\varepsilon}_h = \mathbf{B}\,\mathbf{d}_e, \tag{2.21}$$

where $\mathbf{d}_e = (u_{x,1}, u_{y,1}, \ldots, u_{x,n}, u_{y,n})^\top$ is the vector of nodal unknowns for the element, and $\mathbf{B}$ is the strain–displacement matrix containing derivatives of the shape functions. The stresses then follow from

$$\boldsymbol{\sigma}_h = \mathbf{D}\,\boldsymbol{\varepsilon}_h, \tag{2.22}$$

where $\mathbf{D}$ is the constitutive stiffness matrix for either plane stress or plane strain [12, 13].

Substituting these approximations into the weak form gives the element equations. Here $\Omega^e \subset \Omega$ denotes the domain of element e, and $\Gamma_N^e = \Gamma_N \cap \partial\Omega^e$ is the portion of its boundary subject to Neumann conditions. The stiffness matrix and equivalent nodal force vector of an element are

$$\mathbf{K}^e = \int_{\Omega^e} \mathbf{B}^\top \mathbf{D}\, \mathbf{B}\, dx, \qquad \mathbf{f}^e = \int_{\Omega^e} \mathbf{N}^\top \mathbf{f}\, dx + \int_{\Gamma_N^e} \mathbf{N}^\top \bar{\mathbf{t}}\, ds, \tag{2.23}$$

where $\mathbf{N}$ is the interpolation matrix built from the shape functions [12, 13]. For isoparametric elements, the integrals are evaluated on a reference element $\hat{\Omega}$ by means of the Jacobian $J_e = \partial \mathbf{x}/\partial \hat{\boldsymbol{\xi}}$ of the mapping, with gradients transformed as

$$\nabla N_i = J_e^{-T} \hat{\nabla} \hat{N}_i. \tag{2.24}$$

Numerical quadrature is used to evaluate these integrals; for linear triangles, the strain–displacement matrix $\mathbf{B}$ is constant, so a one-point rule suffices for exact integration [12, 13, 14].

Numerical quadrature on the reference element is used to evaluate these integrals [13, 12]. For $P1$ triangles the matrix $\mathbf{B}$ is constant, so a one–point rule is exact [13, 12]. For $Q1$ quadrilaterals a 2×2 Gauss–Legendre rule at $\xi, \eta = \pm 1/\sqrt{3}$ is adopted to integrate the bilinear mapping and material response robustly; reduced 1×1 rules are avoided to prevent spurious zero–energy (hourglass) modes [12, 13]. This formulation covers both triangular and rectangular meshes and leaves the global assembly and boundary conditions unchanged [14, 13].

2.5 Assembly and Solution

The element stiffness matrices $\mathbf{K}^e$ and load vectors $\mathbf{f}^e$ are then combined into a global system that represents the behaviour of the whole structure. This process, called assembly, consists of adding each element contribution into the global stiffness matrix $\mathbf{K}$ and load vector $\mathbf{f}$ at the positions corresponding to the global degrees of freedom of its nodes [12, 13]. If several elements meet at a node, their contributions are summed, ensuring both displacement compatibility and force equilibrium at that node [12, 13]. The result is the global algebraic system

$$\mathbf{K}\mathbf{u} = \mathbf{f}, \tag{2.25}$$

where $\mathbf{u}$ is the vector of all nodal displacements in the mesh [13].

At this stage, the system is not yet solvable: without boundary conditions, $\mathbf{K}$ is singular, reflecting the possibility of rigid-body motions [12, 13]. Essential (Dirichlet) conditions are

applied by prescribing displacement values at selected nodes, which removes the indeterminacy and ensures a unique solution. These conditions are implemented in practice by modifying the system to enforce the prescribed displacements directly [12, 13]. Natural (Neumann) conditions, such as prescribed tractions, appear in the load vector through the boundary integrals evaluated at the element level. Concentrated loads may also be applied by adding point forces directly to the relevant entries of $\mathbf{f}$ [12, 13].

With the boundary conditions enforced, the system matrix is symmetric and positive-definite for linear elasticity problems [13, 14]. The linear system is then solved for the nodal displacements $\mathbf{u}$ using either direct solvers, such as LU or Cholesky factorization, or iterative methods for large-scale systems [12, 13]. Once the displacement vector is known, strains and stresses are recovered at the element level via

$$\boldsymbol{\varepsilon} = \mathbf{B}\mathbf{d}_e, \qquad \boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\varepsilon}, \tag{2.26}$$

as derived from the discretization and constitutive relations [12, 13].

In this way, the weak formulation, discretization, assembly, and solution stages of the finite element method connect the continuum equations of elasticity to computable numerical approximations, producing the displacement, strain, and stress fields required for analysis [12, 14, 13].

Chapter 3

Neural Networks

In recent decades, neural networks have become a central paradigm in machine learning, offering a flexible framework for modeling complex, high-dimensional data. Their ability to approximate nonlinear functions and automatically extract meaningful representations makes them particularly suitable for tasks that are difficult to address with traditional methods [17]. In this chapter, we present the fundamental concepts of neural networks, providing the theoretical and methodological basis for their later application to problems in computational mechanics.

We begin with an overview of the basic building blocks of neural networks, including their layered architecture, activation functions, and the role of weights and biases. We then discuss the process of fitting these models, covering the forward pass, the definition of loss functions, and the principle of backpropagation for parameter updates. The training of neural networks is examined in more detail, introducing optimization algorithms such as gradient descent, stochastic gradient descent, momentum-based methods, and Adam. Key hyper-parameters, like learning rate, batch size, and epochs, are discussed, together with regularization strategies such as dropout, weight decay, and early stopping. We also address standard metrics used to evaluate model performance.

Beyond these foundations, we introduce convolutional neural networks (CNNs), which extend the basic architecture by incorporating convolutional and pooling layers for feature extraction, and which have become essential for image-based tasks. Finally, we present the U-Net architecture, a specialized encoder–decoder network with skip connections, widely used for image segmentation. The inclusion of CNNs and U-Net establishes the connection between machine learning techniques and the finite element method tasks addressed later in this work, particularly those involving image-based modeling and simulation.

3.1 Fundamentals of Neural Networks

A neural network can be regarded as a regression or classification model that takes an input vector of p variables,

$$X = (X_1, X_2, \ldots, X_p). \tag{3.1}$$

and learns a function $f(X)$ to predict the response Y [9]. In regression problems ($K = 1$), there is only a single output Y. For a K-class classification problem, the output layer contains K units, where the k-th unit models the probability of class k. The class labels are represented by a one-hot encoded vector

$$Y = (Y_1, Y_2, \ldots, Y_K). \tag{3.2}$$

which consists of K dummy variables, taking the value 1 at the true class position and 0 elsewhere [9].

Figure 3.1 illustrates a feed-forward neural network with a single hidden layer. The input layer consists of the features $X_1, X_2, \ldots, X_p$. Each input is connected to all the units in the hidden layer $Z_1, Z_2, \ldots, Z_M$, as indicated by the arrows. The activations of the hidden units are then passed to the output layer $f_1(X), f_2(X), \ldots, f_K(X)$, where each output unit corresponds to one class label or response value.

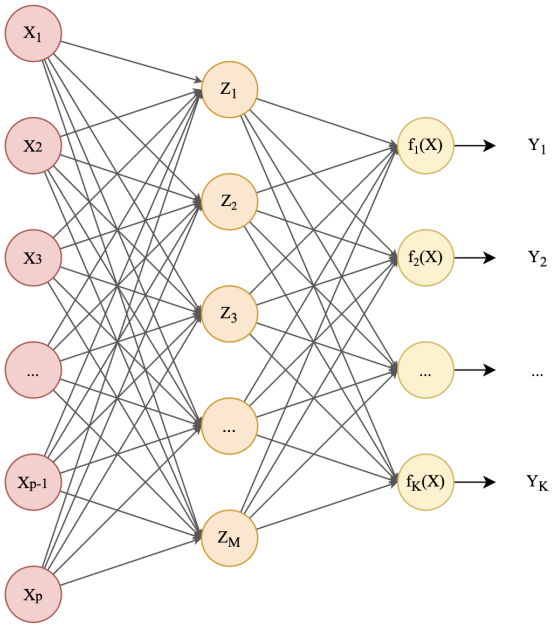


Figure 3.1: Schematic of a feed-forward neural network, with one hidden layer (adapted from [9]).

Derived features Z_m are created from linear combinations of the inputs, and the outputs $f_k(X)$ are estimated as functions of linear combinations of the Z_m:

$$
\begin{aligned}
Z_m &= \sigma(b_m + w_m^T X), \quad m = 1, \ldots, M, \\
T_k &= \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K, \\
f_k(X) &= g(T_k), \quad k = 1, \ldots, K,
\end{aligned}
\tag{3.3}
$$

where $Z = (Z_1, Z_2, \ldots, Z_M)$ and $T = (T_1, T_2, \ldots, T_K)$. Here, b_m are the biases of the hidden layer and β_{0k} are the biases of the output layer, while w_m^T and β_k are the corresponding weights. The functions $\sigma(\cdot)$ and $g(\cdot)$ denote activation functions specified in advance [18]. Figure 3.2 illustrates two units: the one on the right belongs to the hidden layer, while the one on the left is from the output layer. It shows the construction of the derived features Z_1 and $f_1(X)$ as the result of a linear combination of the weights together with the bias terms.

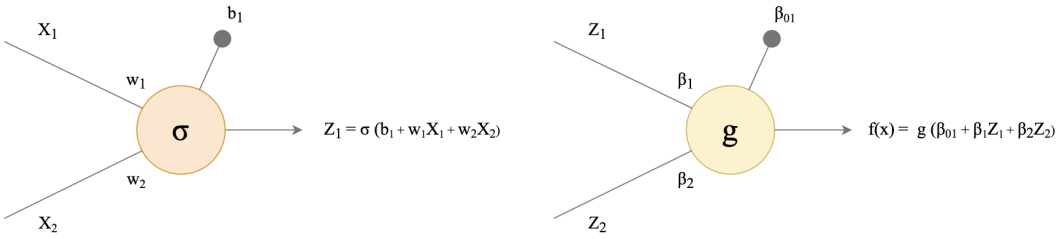


Figure 3.2: Schematic of two units, illustrating the construction of the derived features Z_1 and $f_1(X)$ as a result of the linear combination of the weights and bias terms.

Each unit applies a nonlinear transformation, known as an activation function, to the weighted sum of its inputs. Activation functions are essential, as without them the network would reduce to a simple linear model regardless of its depth [9]. The most commonly used activation functions are illustrated in Figure 3.3.

Different tasks require different activation functions at the output layer. For regression problems, a linear activation $g_k(T) = T_k$ is typically used. If the same linear activation were applied across all layers, however, the entire network $f_k(X)$ would collapse into a simple linear model in the inputs $X_1, X_2, \ldots, X_p$ [18].

For classification problems, the goal is to estimate class probabilities. In this case, the output layer usually employs the softmax activation function, which ensures that the outputs sum to one and can be interpreted as probabilities:

$$
f_k(X) = Pr(Y = k \mid X) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}, \quad k = 1, \ldots, K. \tag{3.4}
$$

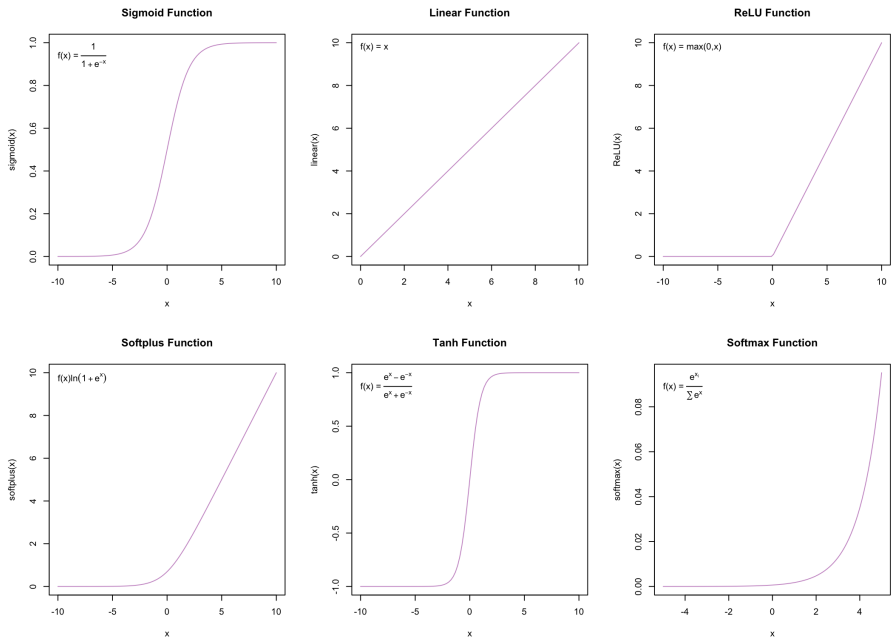


Figure 3.3: Activation functions: *sigmoid*, *linear*, *ReLU*, *softplus*, *tanh*, and *softmax*.

This guarantees that the K outputs form a valid probability distribution. The classifier then assigns the predicted label to the class with the highest probability [9].

In the binary case ($K = 2$), the output layer typically employs the sigmoid activation function:

$$f(X) = Pr(Y = 1 \mid X) = \frac{1}{1 + e^{-T}}. \tag{3.5}$$

At inference time, another common operation is the argmax function, which converts the highest estimated probability into the predicted class by setting that value to 1 and all others to 0. However, the argmax function is piecewise constant: its output remains unchanged for small variations in the input, and its gradient is zero almost everywhere and undefined at points where two values are equal. Since backpropagation relies on meaningful gradients to update the weights and biases (see Chapter 3.2), the argmax function cannot be used during training. Instead, it is applied only after training to assign the final class label [19].

Finally, a neural network may include more than one hidden layer, where each subsequent layer is a transformation of the features Z_m from the previous layer. Since these features are themselves functions of the input vector X, deeper networks can construct increasingly complex representations of the input data. This layered composition enables neural networks to approximate highly nonlinear relationships, providing the expressive power that underlies their success in modern machine learning applications [18, 9].

3.2 Training a Neural Network

Training a neural network is the end-to-end process of turning data and a chosen architecture into a model that generalizes to unseen inputs. It includes specifying the architecture and outputs, choosing an objective (loss) and evaluation metrics, preparing the data and defining train/validation/test splits, selecting an optimizer and hyperparameters, applying regularization and stabilization (e.g., weight decay, dropout, early stopping, normalization), and monitoring on a validation set with checkpointing and model selection [9, 22, 21]. Parameter estimation, learning the weights and biases via backpropagation, is the core inner loop of this broader process.

Let $(x_i, y_i),\ i = 1, \ldots, N$ denote the training data. The parameters are $\{b_m, w_m;\ m = 1, \ldots, M\}$ for the hidden layer, with $M(p+1)$ coefficients, and $\{\beta_{0k}, \beta_k;\ k = 1, \ldots, K\}$ for the output layer, with $K(M+1)$ coefficients. In the inner optimization loop, these parameters are estimated by minimizing a loss function that quantifies the discrepancy between predictions and observed outputs.

For regression problems, a common choice is the sum-of-squared errors:

$$R(\theta) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} \big(y_i - f(x_i; \theta)\big)^2. \tag{3.6}$$

For classification, the most widely used loss is the cross-entropy:

$$R(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \Big[y_i \log f(x_i; \theta) + (1 - y_i) \log \big(1 - f(x_i; \theta)\big) \Big], \tag{3.7}$$

which can be derived from the log-likelihood function,

$$\ell(\theta) = \log \prod_{i=1}^{N} f(x_i; \theta)^{y_i} \big(1 - f(x_i; \theta)\big)^{1 - y_i}. \tag{3.8}$$

More generally, given a training dataset $D_{\text{tr}} = \{(x_i, y_i)\}_{i=1}^{N}$, the loss function can be written as

$$R(D_{\text{tr}}, \theta) = \frac{1}{N} \sum_{i=1}^{N} \ell\big(f(x_i; \theta), y_i\big), \tag{3.9}$$

where $\ell(\cdot, \cdot)$ is an appropriate error measure, such as squared error for regression or cross-entropy for classification. Training consists of finding the optimal parameters

$$\theta^* = \arg\min_{\theta} R(D_{\text{tr}}, \theta). \tag{3.10}$$

The function $R(D_{\text{tr}}, \theta)$ quantifies the discrepancy between predictions and targets, and its

minimization guides the optimization process. However, because neural networks are highly non-linear in θ, the loss surface is generally non-convex and can contain multiple local minima, saddle points, and flat regions. In local neighborhoods where the gradients are well behaved, the loss may exhibit approximate convexity, but in general the optimization landscape remains complex [9]. These properties have important implications for convergence and stability, motivating the use of advanced optimization strategies and regularization, as discussed in Section 3.2.

Minimizing the loss is typically achieved through gradient descent, which iteratively updates the parameters in the direction of the steepest decrease of $R(\theta)$. In the neural network setting, this procedure is known as backpropagation.

For illustration, consider a binary classification problem with a hidden unit

$$Z_1 = \sigma(b_1 + w_1 X_1 + w_2 X_2). \tag{3.11}$$

Assume a sigmoid output $f(x;\theta) = \sigma(\beta_0 + \beta Z_1)$. The derivatives of the loss with respect to the parameters are

$$\frac{\partial R(\theta)}{\partial w_1} = \frac{1}{N}\sum_{i=1}^{N} \left(f(x_i;\theta) - y_i\right) \beta\, \sigma'(b_1 + w_1 x_{i1} + w_2 x_{i2})\, x_{i1}, \tag{3.12}$$

$$\frac{\partial R(\theta)}{\partial w_2} = \frac{1}{N}\sum_{i=1}^{N} \left(f(x_i;\theta) - y_i\right) \beta\, \sigma'(b_1 + w_1 x_{i1} + w_2 x_{i2})\, x_{i2}, \tag{3.13}$$

$$\frac{\partial R(\theta)}{\partial b_1} = \frac{1}{N}\sum_{i=1}^{N} \left(f(x_i;\theta) - y_i\right) \beta\, \sigma'(b_1 + w_1 x_{i1} + w_2 x_{i2}), \tag{3.14}$$

where $\sigma'(u) = \sigma(u)(1 - \sigma(u))$.

Using these gradients, the parameter update at iteration $t+1$ takes the form

$$\theta^{(t+1)} \longleftarrow \theta^{(t)} - \alpha \nabla R(\theta^{(t)}), \tag{3.15}$$

where α is the learning rate.

This process is illustrated in Figure 3.4. Note that in this thesis the trainable parameters are denoted by θ, whereas the figure uses the conventional symbol W for weights; the two notations are equivalent. Panel (A) corresponds to the calculation of partial derivatives using the chain rule, as expressed in (3.12)–(3.14). Panel (B) represents the parameter update rule in (3.15), where the error terms and local derivatives are combined to adjust the weights. Finally, Panel (C) summarizes the full training cycle: a forward pass computes the activations, a backward pass propagates the error gradients, and an update step modifies the weights.

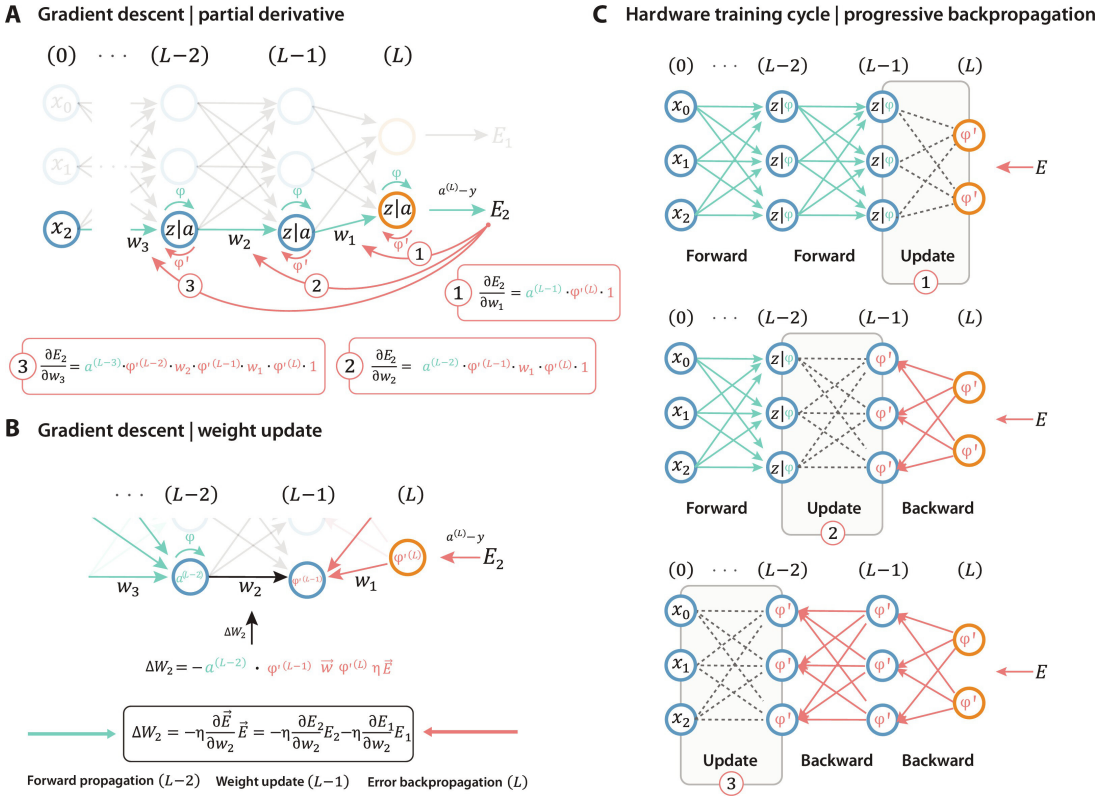


Figure 3.4: Illustration of the backpropagation algorithm. (A) Computation of partial derivatives of the error via the chain rule. (B) Weight updates according to gradient descent. (C) Training cycle showing the sequence: forward propagation, backward propagation, and parameter update (source [20]).

In practice, the procedure starts with an initial guess $\theta^{(0)}$, computes predictions $f(x_i;\theta)$ in the forward pass, evaluates the loss and its gradient, and updates the parameters according to (3.15). This cycle is repeated until convergence, e.g., when the loss on the training (and preferably validation) set ceases to improve or the gradients become small. For a sufficiently small learning rate α and full-batch updates, a step along the negative gradient decreases the objective $R(\theta)$, i.e., $R(\theta^{(t+1)}) \leq R(\theta^{(t)})$; with mini-batch updates this monotonic decrease need not hold at every step but typically holds in expectation or across an epoch. If the gradient vector is zero, the algorithm has reached a stationary point (which may be a minimum, maximum, or saddle point) [9].

Beyond these basics, practical training requires more than gradient descent. Effective learning also depends on optimization algorithms, hyperparameter tuning, and regularization to ensure both convergence and generalization.

Because neural networks are highly nonlinear, the minimization of the loss function has no closed-form solution, and iterative methods must be employed [9]. The foundation of modern optimization is gradient descent, which updates parameters in the opposite direction of the gradient (see Eq. 3.15). Depending on how much data is used per update, we obtain three basic

variants: full-batch gradient descent, which is stable but computationally expensive; stochastic gradient descent (SGD), which updates with a single example and is fast but noisy; and mini-batch gradient descent, which balances efficiency and stability and is the default in deep learning [22]. Figure 3.5 summarizes the three gradient-descent algorithms. A full sweep through the dataset is often referred to as an *epoch*.

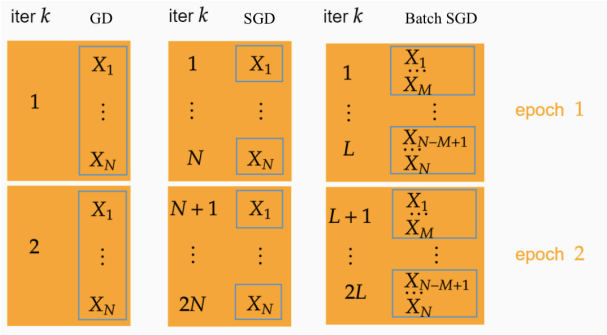
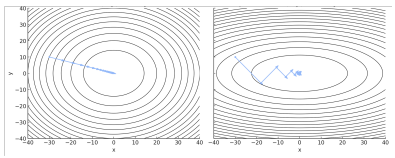
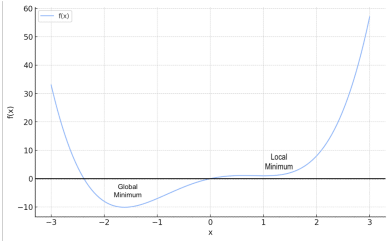


Figure 3.5: Three types of gradient-descent learning algorithms: full-batch, stochastic, and mini-batch gradient descent.

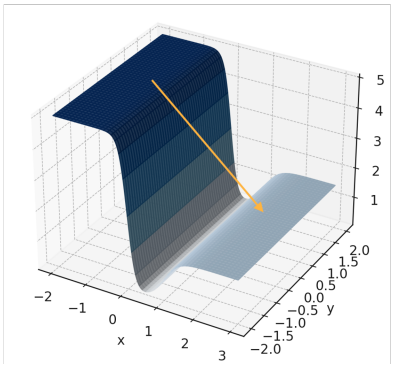
Although conceptually simple, gradient descent struggles with the complex loss landscapes of neural networks. These landscapes are highly non-convex and exhibit several characteristic difficulties [21]. The optimizer may become trapped in local minima rather than reaching a better solution; in narrow valleys with different curvatures, gradients often flip direction in high-curvature dimensions, leading to oscillations and slow convergence; and in flat plateaus or near cliffs, gradients nearly vanish, causing training to stall. These problems are illustrated in Figure 3.6.



(a) The effect of the shape of the loss function on steepest-gradient descent.



(b) Non-convex loss with local and global minima.



(c) Gentle gradient plateau before a steep cliff, where small step sizes undershoot and large step sizes overshoot.

Figure 3.6: Illustrations of common challenges in neural network optimization landscapes: (a) the effect of curvature leading to oscillations in narrow valleys, (b) the presence of multiple local and global minima, and (c) gentle gradient plateaus followed by steep cliffs, where small step sizes undershoot and large step sizes overshoot.

These difficulties motivated refinements of gradient descent. SGD already improves over

full-batch methods by injecting noise that helps the optimizer escape shallow minima and explore the loss landscape more effectively. In practice, its mini-batch variant is most widely used [22], balancing computational efficiency with stochastic exploration. However, SGD still suffers from slow convergence in narrow, curved valleys of the loss surface, where the gradient direction fluctuates dramatically and the optimizer zigzags, making progress inefficient.

To address this, the momentum method was introduced. Momentum can be interpreted as a ball rolling down a slope, instead of responding only to the current gradient, it accumulates a fraction of past gradients, effectively smoothing out oscillations and accelerating movement along directions of consistent descent. Formally, momentum updates are given by

$$v_t = \beta v_{t-1} + \nabla_\theta L(\theta^{(t)}),\ \theta^{(t+1)} = \theta^{(t)} - \alpha v_t, \tag{3.16}$$

where $\beta \in [0, 1)$ is the momentum parameter, controlling how much past gradients influence the current update, and α is the learning rate.

This accumulation allows momentum to build speed in shallow directions while damping oscillations in steep or narrow directions, leading to faster and more stable convergence compared to vanilla SGD.

Although momentum improves stability, it does not adapt the learning rate across dimensions. In loss landscapes with steep curvature in some directions and gentle slopes in others, a fixed learning rate can still hinder convergence. RMSProp tackles this by scaling updates according to a running average of squared gradients, which normalizes step sizes across parameters,

$$s_t = \rho s_{t-1} + (1 - \rho) \left(\nabla_\theta L(\theta^{(t)}) \right)^2, \tag{3.17}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\alpha}{\sqrt{s_t} + \epsilon} \nabla_\theta L(\theta^{(t)}), \tag{3.18}$$

where $\rho \in [0, 1)$ is the decay parameter and ϵ is a small constant (e.g., 10^{-8}) added for numerical stability.

Building on both ideas, the Adam optimizer (Adaptive Moment Estimation) combines the benefits of momentum and RMSProp. It maintains exponentially decaying averages of both the gradient (first moment) and its square (second moment), while applying bias correction to

account for initialization effects,

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta L(\theta^{(t)}), \tag{3.19}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla_\theta L(\theta^{(t)})\right)^2, \tag{3.20}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \tag{3.21}$$

leading to the parameter update

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}. \tag{3.22}$$

By simultaneously smoothing oscillations, adapting step sizes per parameter, and correcting initialization bias, Adam integrates the strengths of both momentum and RMSProp, which explains its widespread success in deep learning [21].

Even with effective optimizers, training remains highly dependent on hyperparameters, which govern the dynamics of learning but are not learned directly from the data. The most critical is the learning rate: values that are too small imply slow convergence, while values that are too large cause the training to diverge or oscillate. Batch size also plays a central role: small values introduce useful noise for exploration but reduce stability, while large values smooth gradients but increase computation. Optimizers with momentum or adaptive learning rates require tuning of decay parameters, such as β for momentum or (β_1, β_2) for Adam, which determine how much past gradients contribute to current updates and thus balance immediate and accumulated gradient information. Choosing appropriate hyperparameters is therefore essential for stable and efficient training, often requiring systematic search or empirical tuning [9, 18, 21].

3.2.1 Challenges in Training Neural Networks

Training neural networks is difficult because the optimization problem is highly non-convex, the models are generally over-parameterized, and the learning process is unstable unless certain guidelines are followed [9]. Even with well-tuned hyperparameters, networks are prone to overfitting due to their high capacity [9]. In fact, the global minimum of the loss function is often not desirable, as it typically corresponds to an overfit solution [9, 18]. Instead, some form of regularization is needed: either penalties are imposed on the parameters, or training is stopped earlier when overfitting is detected [18]. An explicit approach is to augment the objective function with a penalty term

$$R(\theta, \lambda) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log(f_k(x_i)) + J(\theta), \tag{3.23}$$

where $J(\theta) = \sum_{k,m} \beta_{km}^2 + \sum_{m,l} w_{ml}^2$ represents an L_2 penalty on the parameters [9]. The term L_2 refers to the squared magnitude of the coefficients, which penalizes large parameter values and encourages smaller, more stable weights [9, 18]. In the context of linear models, adding such a penalty to the least squares objective is known as ridge regression, where the inclusion of the quadratic term reduces variance and improves generalization by shrinking coefficients toward zero [9, 18]. Figure 3.7 shows the result of training a neural network with ten hidden units, comparing results obtained without (upper panel) and with regularization (lower panel). The broken purple line shows the Bayes error rate, while the black line is the test error. It is noticeable that the inclusion of regularization improves the prediction, moving it closer to the Bayes error rate (see [18] for details about the data).

In addition to explicit penalties, heuristic approaches play a major role in regularization. Early stopping monitors the validation error and interrupts training once it ceases to improve, effectively acting like L_2 regularization by preventing unchecked growth of the weights [22]. Dropout is conceptually related to L_2 regularization, but operates in a stochastic way [22]. The idea is to randomly remove a fraction of the hidden units in a layer during training, so that each iteration processes the data with a slightly different subnetwork [9]. The surviving units stand in for those removed, and their weights are scaled up by a factor of $1/(1-\phi)$, where ϕ is the dropout rate, to ensure the overall magnitude of activations remains consistent [9]. This prevents hidden units from becoming overly specialized and reduces co-adaptation, thereby improving generalization [9]. Figure 3.8 illustrates this process.

Another important difficulty lies in initialization. When the weights are set very close to zero, the network initially behaves almost linearly, since the operative region of the sigmoid function is approximately linear [18]. If the weights are instead too large, the activations can saturate and the dynamics become unstable [18]. A practical compromise is to initialize the weights randomly near zero, so that the network begins in a nearly linear regime and becomes progressively more nonlinear as training advances [18]. Closely related to initialization is the scaling of inputs. Standardizing inputs to have mean zero and unit variance ensures that all features are treated equally, simplifies the choice of initial weights, and improves the effectiveness of regularization [18]. Without scaling, convergence can be slow and optimization may get stuck in poor solutions [18]. The choice of architecture adds another layer of complexity. Too few hidden units limit the network's ability to capture nonlinearities in the data, while too many increase the risk of overfitting [18]. A common practice is to use more hidden units than strictly necessary while

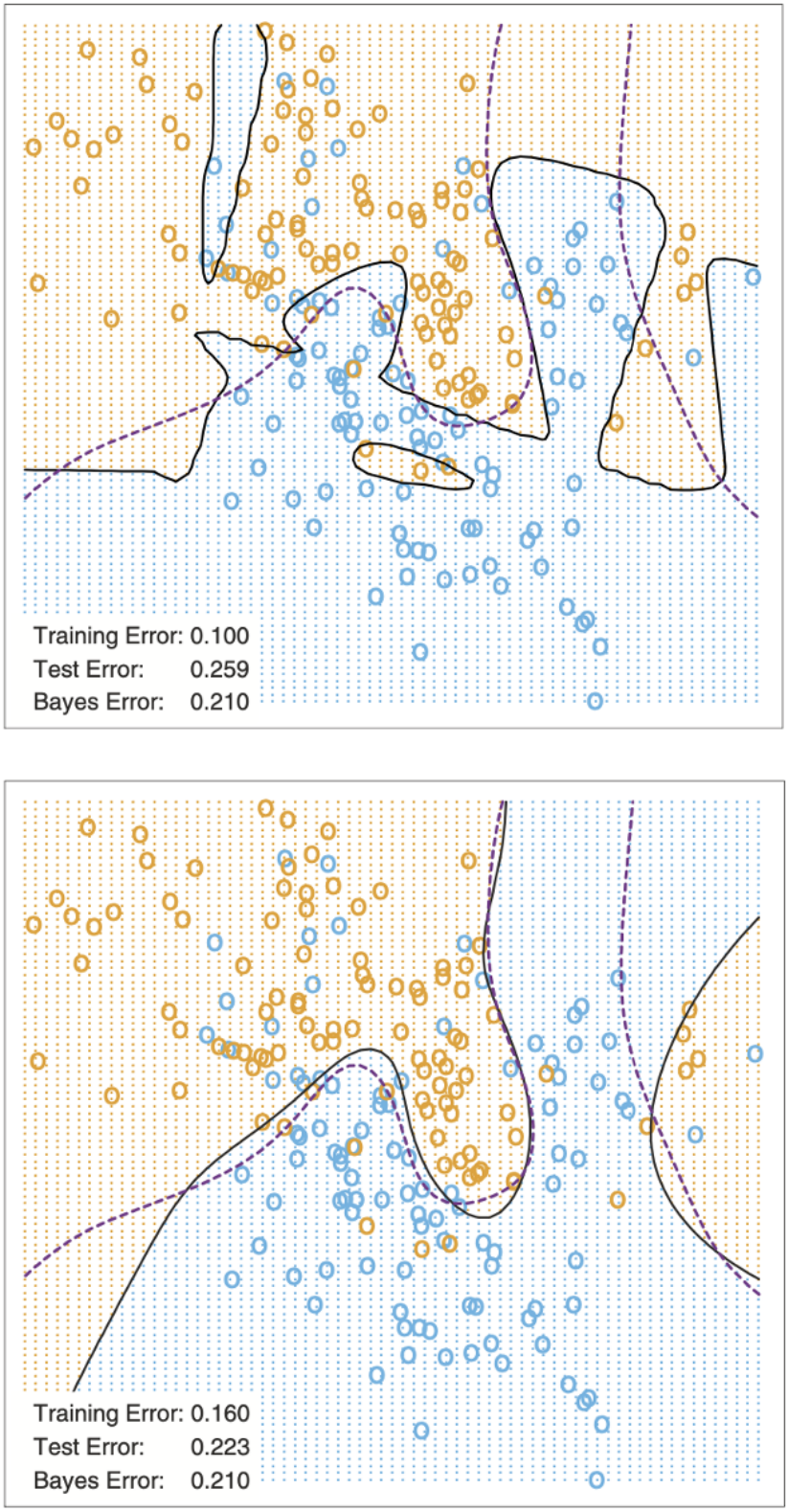


Figure 3.7: A neural network trained on a simulated dataset with 10 hidden units. The upper panel shows training without regularization, while the lower panel applies the penalty in (3.23). Both use the *softmax* activation and cross-entropy error (source: [18]).

relying on regularization to shrink redundant parameters [18]. Multiple hidden layers allow the extraction of hierarchical features at increasing levels of abstraction, but at the cost of greater

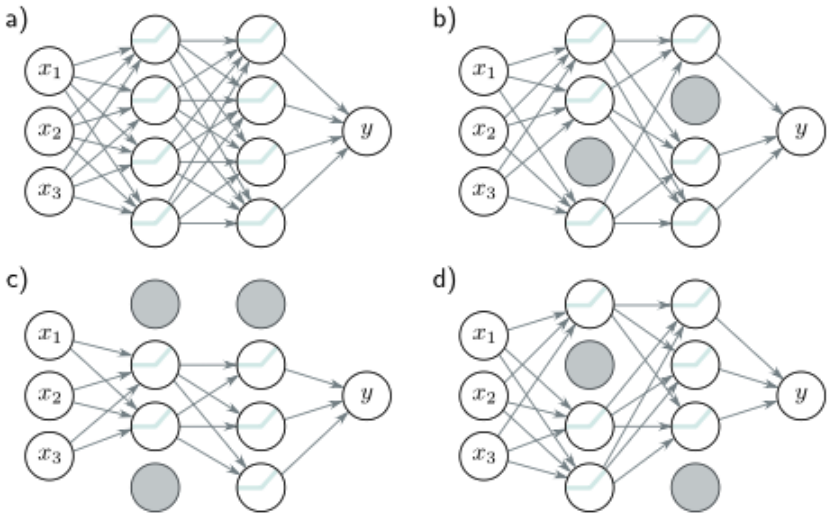


Figure 3.8: Illustration of dropout regularization. At each training iteration, a random subset of hidden units is deactivated (gray nodes), forcing the network to rely on multiple pathways and reducing co-adaptation between neurons. At test time, all units are active, with rescaled weights to account for the dropout rate (source: [22]).

training complexity and sensitivity [18]. Finally, the non-convex error surface is filled with local minima and saddle points, making convergence heavily dependent on initialization and the stochastic trajectory of training [21]. Strategies such as using multiple random starts, averaging predictions from ensembles, or bagging with perturbed training data can mitigate this sensitivity [21].

In summary, training neural networks involves managing a combination of issues, from overfitting and regularization to initialization, scaling, and architectural design. Their flexibility makes them powerful predictive models, but also fragile, requiring careful design choices and stabilization techniques to achieve robust and generalizable performance.

3.3 Convolution Neural Networks

A Convolutional Neural Network (CNN or ConvNet) is a type of neural network distinguished by its superior performance for processing data with a grid-like topology, such as images [24]. The main difference between a standard neural network and a CNN is that while the former processes data without considering local context—often leading to overfitting to noise—a CNN uses convolution operations to capture the local neighborhood of input data and construct features from it [23].

CNNs classify images by identifying specific features or patterns anywhere in the image that distinguish each object class. The network initially detects low-level features, such as small edges

or patches of color, which are then combined to form higher-level representations. Ultimately, the presence or absence of these higher-level features contributes to the probability of any given output class [9].

This hierarchical representation is built by combining two specialized types of hidden layers: convolution layers and pooling layers. Convolution layers search for instances of small patterns in the image, while pooling layers downsample these patterns to retain the most prominent ones [9].

3.3.1 Convolution Layers

A convolutional layer is made up of a large number of convolution filters, also known as kernel filters, each serving as a template to identify specific local features within an image. These filters rely on a mathematical operation known as convolution, involving the repeated multiplication of matrix elements followed by summation of the results [9].

Figure 3.9 shows an image of the number 1, which has been changed into a binary representation with a resolution of 5 x 5 pixels. In this representation, each black pixel representing the number 1 is assigned the value 1, while the remaining pixels are assigned the value 0. Generally, the model is trained using a database containing multiple images. Each image consists of several pixels, collectively referred to as resolution, with pixel values ranging from 0 to 255. The numbers for each image are organized in a three-dimensional array called a feature map. The first two axes are spatial and the third is the channel axis, representing the three colors (RGB). For grayscale images, the representation is simplified to a two-dimensional array [9].

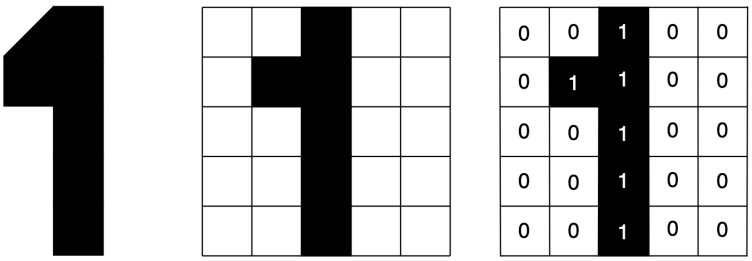


Figure 3.9: Binary representation of the number 1, where each black pixel representing the number 1 is assigned the value 1, while the remaining pixels are assigned the value 0.

To understand how convolution filters work, consider the following filter:

$$Convolution\ Filter = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The objective is to perform the convolution operation by sliding the convolution filter through the image. At each location, an element-wise matrix multiplication is performed, as illustrated in Figure 3.10 [9].

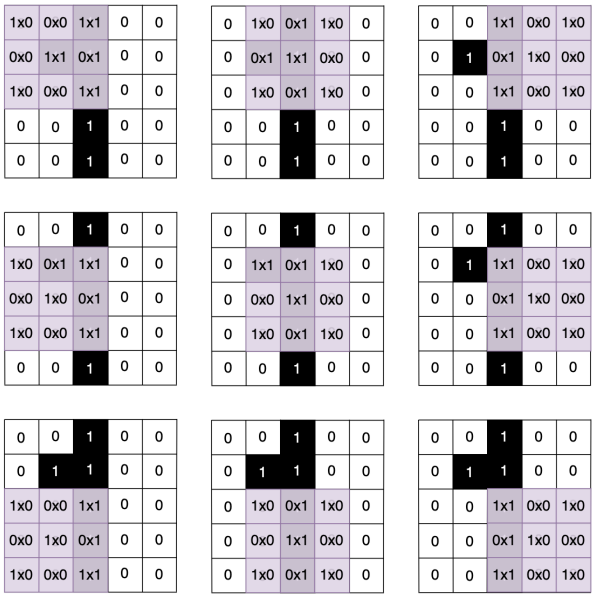


Figure 3.10: Steps of the element-wise multiplication.

For each element-wise multiplication, the results are added to form a new feature map, also known as a convolved image. Figure 3.11 shows the construction of the new convolved image based on the matrix multiplication obtained in Figure 3.10. Due to the size of the filter, the receptive field is also 3x3 [9].

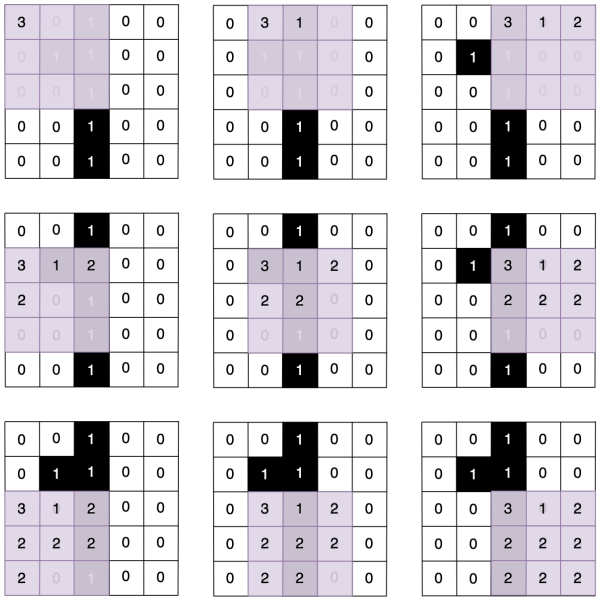


Figure 3.11: Each element-wise multiplication result from the convolution process is summed to generate the values of the new feature map.

Generally, given an image $A \in \mathbb{R}^{n \times m}$ and a convolution filter $E \in \mathbb{R}^{l_1 \times l_2}$, the convolution operation between the image and the filter results in

$$(A \times E)_{r,s} = \sum_{i=0}^{l_1} \sum_{j=0}^{l_2} a_{r-l_1, s-l_2} \cdot e_{l_1, l_2}, \tag{3.24}$$

where r denotes the row index and s denotes the column index of the resulting matrix.

For a convolved image with the same dimensions as the original, padding can be applied. Padding denotes the process of adding zeros to each side of the boundaries of the input. This value can either be manually specified or automatically set [24].

If a submatrix of the original image closely resembles the convolution filter, then it will result in a large value in the convolved image; otherwise, it will have a small value. As a result, the convolved image highlights regions of the original image that resemble the convolution filter. In a convolutional layer, a set of filters are applied to detect a variety of different-oriented edges and shapes in the image. In CNNs, these filters are learned specifically for the classification task, and their weights serve as parameters going from an input layer to a hidden layer, with one hidden unit for each pixel in the convolved image. This learning process is achieved using backpropagation [9].

3.3.2 Pooling Layers

The pooling layer is a downsampling operation, usually applied after a convolution layer. It provides a way to condense similar features into a single pixel. The typical types of pooling are max pooling and average pooling [24]. Given a matrix of the convolved image

$$Convolved\ Image = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix},$$

the result of max pooling is

$$Max\ pool = \begin{bmatrix} max(a,b,e,f) & max(c,d,g,h) \\ max(i,j,m,n) & max(k,l,o,p) \end{bmatrix},$$

and the result of average pooling is

$$Average\ pool = \begin{bmatrix} \frac{1}{4}(a+b+e+f) & \frac{1}{4}(c+d+g+h) \\ \frac{1}{4}(i+j+m+n) & \frac{1}{4}(k+l,o,p) \end{bmatrix}.$$

This operation provides location variance, meaning that as long as there is a large value in one of the four pixels in the block, the entire block is considered to have a large value in the reduced image [9].

3.3.3 Architecture of a Convolutional Neural Network

A Convolutional Neural Network, much like a standard neural network, receives an input, typically an image, and through various operations predicts the response Y [9]. Figure 3.12 shows a typical architecture for a CNN for a cat image classification task.

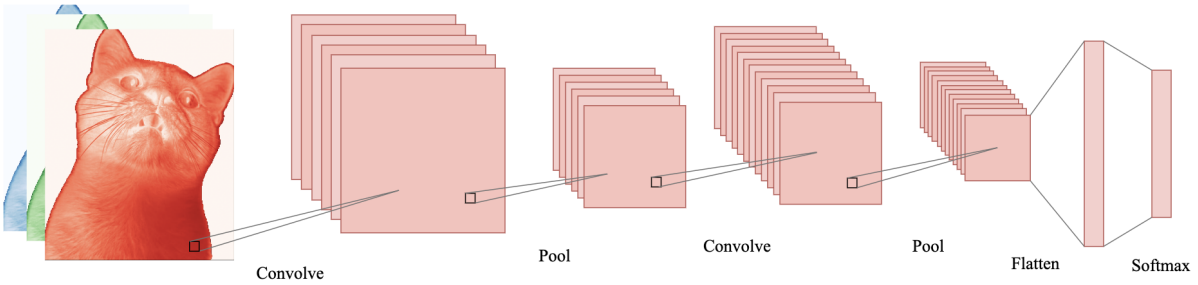


Figure 3.12: Architecture of a CNN for a cat image classification task (adapted from [9]).

The CNN architecture involves a sequence of convolution and pooling operations applied to the input image, followed by fully connected layers. The number of convolution filters in a convolutional layer is similar to the number of units in a hidden layer in a fully connected network, and this number also defines the number of channels in the resulting three-dimensional feature map. Each convolution filter normally has three channels, one per color, with potentially different filter weights. It is common to increase the number of filters in the next convolutional layer after a pooling layer, since the channel feature maps from the previous layer are reduced in size. Sometimes several convolutional layers are stacked before a pooling layer. When each channel feature map has been reduced down to just a few pixels in each dimension, the three-dimensional feature maps are flattened and the pixels are treated as separate units. They are then fed into one or more fully connected layers before reaching the output layer, which typically uses a *softmax activation* for multi-class classification [9].

3.3.4 Data Augmentation

To overcome overfitting caused by a lack of training examples, data augmentation can be applied. This technique generates new examples by randomly transforming existing ones, replicating distortions in the image in a way that does not affect human recognition. Common transformations include rotation, shifting, resizing, exposure adjustment, and contrast change. This is applied only to the training set and can substantially increase the diversity of examples available for

training [9].

3.4 U-Net

The U-Net is a convolutional neural network with a specific architecture aimed at image segmentation and addresses the image localization problem. This network has an architecture that can work with very few training images and still yields precise segmentation. The main concept behind U-Net involves two pathways: a contracting path and an expansive path. The contracting path consists of encoder layers, which capture contextual information and reduce the spatial resolution of the input image. In contrast, the expansive path consists of decoder layers, which decode the encoded data and use information from the contracting path through skip connections to generate a segmentation map [25].

Figure 3.13 shows the architecture of a U-Net, where the network converts a grayscale input image into a binary segmented image. During the contracting path, the U-Net applies multiple convolution layers followed by pooling layers. This process increases the number of channels, allowing the network to capture high-level features as it progresses down the path. The expansive path then converts the segmentation map back into an image of the same size as the original input. This is done using upsampling layers, which propagate context information to higher-resolution layers. As a consequence, the paths are symmetric, and the network has a U-shape. Additionally, high-resolution features from the contracting path are combined with the upsampled output to enhance segmentation accuracy. The segmentation map only contains pixels for which the full context is available in the input image, while pixels in the border region are predicted by extrapolating the input image through mirroring [25]. Each pixel in the output image represents a label that corresponds to a particular object or class in the input image.

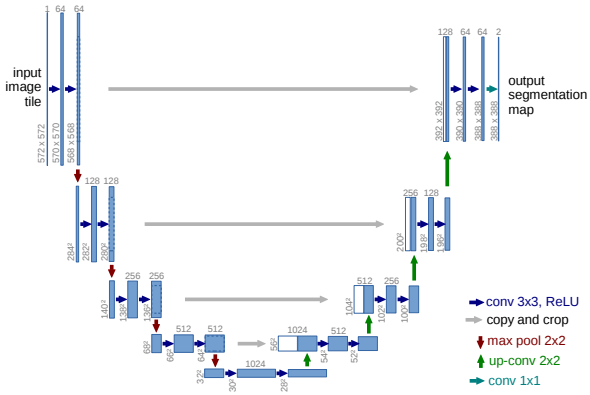


Figure 3.13: U-Net architecture. Each blue box corresponds to a multi-channel feature map and each white box represents copied feature maps. The number of channels is denoted on top of the box and the x-y-size is provided at the lower left edge of the box (source: [25]).

Chapter 4

Neural Networks for Accelerating Simulations in Mechanics

Finite element analysis is the reference method for predicting mechanical response, but repeated solves across many loads, geometries, and design iterations can be prohibitively expensive. This chapter investigates learned surrogates that approximate the mapping from boundary and loading data to displacement, strain, and stress. The goal is to reduce wall-clock time while preserving the essential structure of the solution. We build an end-to-end image-to-solution workflow that begins with segmentation of the regions of interest, proceeds through mesh generation, and finishes with neural networks that predict mechanical fields directly from the discretized data.

We study two complementary surrogate families. The first is a grid U-Net that operates on a fixed Cartesian lattice and learns a force-to-displacement map, with strains and stresses recovered a posteriori from the predicted displacement. The second is a mesh-native graph U-Net, MAgNET, that works on unstructured meshes using local aggregation together with graph pooling and unpooling and skip connections, which avoids grid discretization effects and preserves the original discretization near boundaries and short traction patches. Using matched datasets, we compare both models on rectangular benchmarks and on irregular shapes produced by the image-based meshing pipeline, and we probe robustness with magnitude-extrapolation tests beyond the training range.

The chapter is organized as follows. Section 4.1 develops U-Net segmentation models that isolate the relevant structures in images. Section 4.2 converts the resulting masks into computational meshes using boundary sampling, Poisson-disk interior placement, Delaunay triangulation, and smoothing. Section 4.3 presents the gridded force-to-displacement formulation and evaluates the CNN U-Net on rectangular domains. The final section introduces MAgNET for mesh-based prediction on the native unstructured discretizations and compares it against the CNN under

matched loads and boundary conditions, including extrapolation to higher load magnitudes. Throughout, we report learning curves, field visualizations, absolute error maps, global relative L^2 metrics, and timing to enable like-for-like comparisons in accuracy and speed. These measures quantify surrogate fidelity to FEM fields and timing. We do not compute structural performance metrics used in engineering decisions, such as factors of safety, compliance or strain energy, energy-norm errors, or failure predictions. These depend on application-specific allowables and load cases and are outside the present scope.

4.1 Image Segmentation

The objective of this study is to perform segmentation on micrographs to isolate the structure of the material. To this purpose, various U-Net models were developed for different datasets. Because the number of elements varies in each image, each model performs binary classification and, for the best model, a threshold method was applied to identify regions and isolate the zones. The segmentation task was performed and evaluated in an image of an etched micro-structure of titanium powder, with grain boundaries (Figure 4.1) [26] and in an image of an isolated carbon fiber (Figure 4.2) [27].

The segmentation experiments using the second and third datasets were intentionally trained under limited data regimes to probe feasibility rather than to establish state-of-the-art performance. Results should therefore be interpreted as illustrative demonstrations of the pipeline rather than conclusive benchmarks of segmentation accuracy.

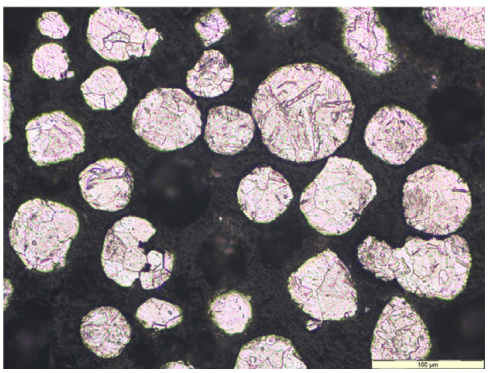


Figure 4.1: Etched micro-structure of titanium powder, with grain boundaries (source: [26]).

For the segmentation task, three datasets were used. The first dataset, created by Warren [28], consists of 481 images of grain boundaries of stainless steel 316L, and 801 artificial grains images. Figure 4.3 illustrates examples of grain boundaries in stainless steel and artificial grains and the corresponding masks.

The second dataset, created by Bertoldo et al. [29], is a collection of 101 x-ray computed

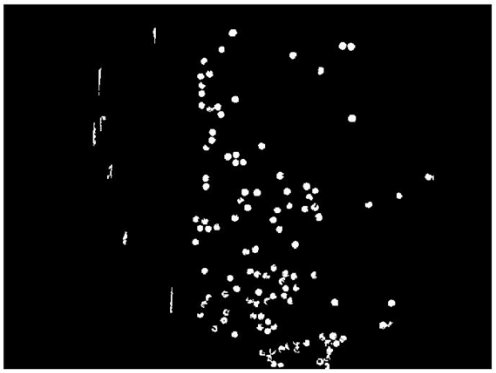


Figure 4.2: Isolated carbon fiber (source: [27]).

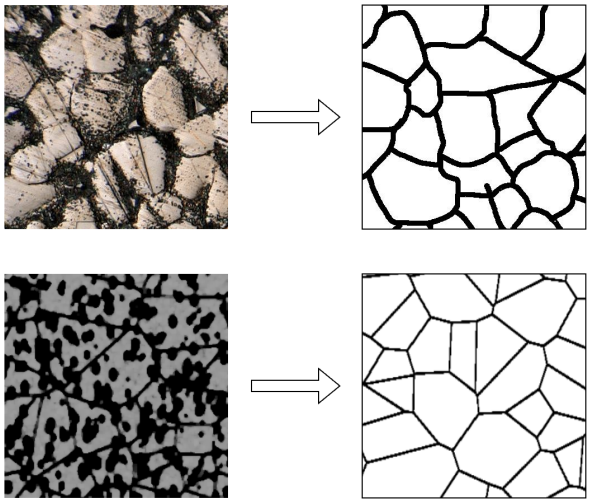


Figure 4.3: Grain boundaries and their masks. The upper panel displays an image of grain boundaries in stainless steel, while the lower panel displays artificial grain boundaries and the corresponding masks (source: [28]).

tomography images of a polyamide 66 reinforced by glass fibers. Figure 4.4 displays the two types of images of the polyamide and their corresponding masks.

The third dataset is composed by 101 images of the first dataset and 100 images of the second dataset.

The first U-Net design, shown in Figure 4.5, utilizes a 256x256 pixel image from the first dataset as its input layer. Initially, two convolution layers with the ReLu activation function was used to create 64 features maps. This was followed by maximum pooling, reducing the size of the feature maps by a factor of two in each dimension. This process was repeated two times, resulting in 128 features maps and then 256 features maps. Afterward, a sequence of up-convolutions, convolution layers with the ReLu activation function and concatenations with high-resolution features, from the contracting path, were applied, resulting in a layer with 128 features maps. The same process was repeated resulting in a a layer with 64 features maps. Finally, a layer with

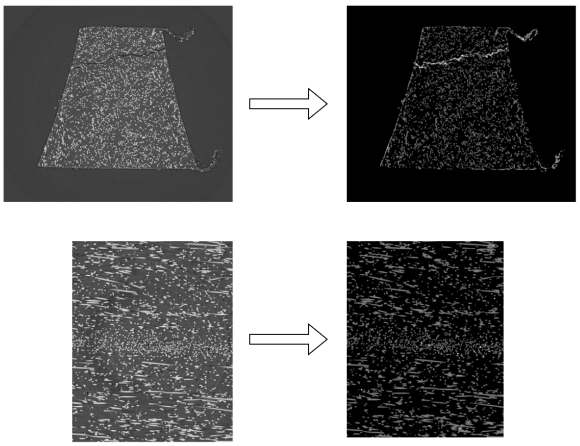


Figure 4.4: Two types of x-ray computed tomography images of a polyamide 66 reinforced by glass fibers and the corresponding masks (source: [29]).

sigmoid activation was employed to yield probabilities for each zone.

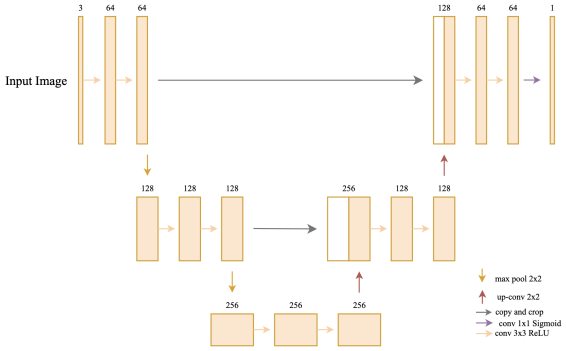


Figure 4.5: U-Net architecture for the first model.

The model was trained using 32 images per batch and a total of 15 epochs, resulting in a training accuracy of 0.8974 and a validation accuracy of 0.9002. Figure 4.6 shows the training and validation accuracy over the epochs. Accuracy values are reported for completeness, but they are not robust to class imbalance in segmentation and should be interpreted alongside the qualitative results. In this context, training accuracy reflects how well the model fits the data it learns from during optimization, while validation accuracy measures its ability to generalize to unseen data without updating the model's parameters. The close alignment of the two curves suggests stable learning with no strong indication of overfitting.

The segmentation results using the first model are displayed in Figure 4.10. The model struggles to capture the boundaries of the images accurately. In Figure 4.7b, the boundaries appear smoother than the original, while in Figure 4.7c, the model fails to distinguish the aggregation of combined circles.

The second U-Net design, displayed in Figure 4.8, utilizes a 256x256 pixel image from the second dataset as the input layer. The architecture of the U-Net is the same as the first one.

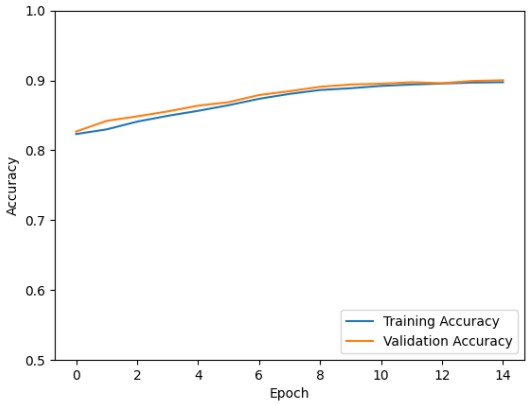
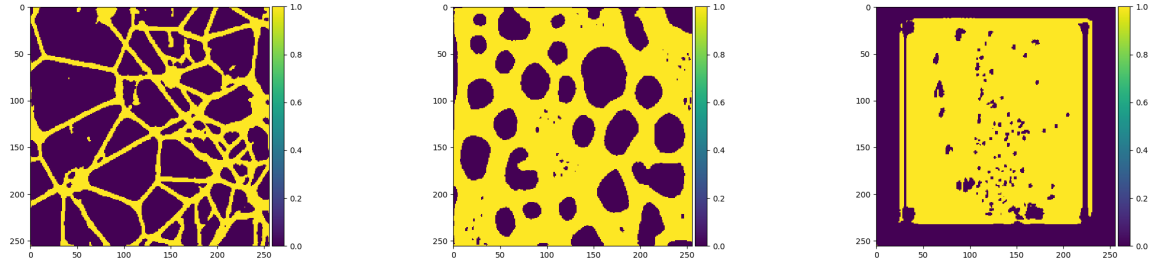


Figure 4.6: Training accuracy and validation accuracy of the first model. The blue line represents the training accuracy curve and the orange line represents the validation accuracy.



(a) Segmentation of an image of grain boundaries of stainless steel 316.

(b) Segmentation of an image of an etched micro-structure of titanium powder.

(c) Segmentation of an image of an isolated carbon fiber.

Figure 4.7: The three figures shown the segmentation task performed with the first model.

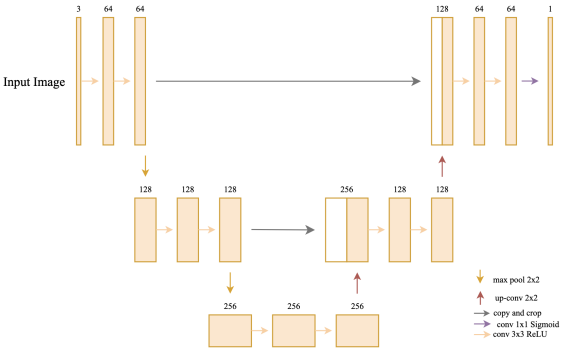


Figure 4.8: U-Net architecture for the second model.

Due to the smaller dataset size, the model was trained using a batch size of 2 images per epoch, with only 2 epochs to train the model. This regime is not robust, so the reported accuracy of 0.9035 and validation accuracy of 0.9181 (Figure 4.9) are indicative rather than conclusive.

Figure 4.10 displays the segmentation results obtained using the second model. The model performs well with the isolated carbon fiber image, but encounters difficulties in segmenting the image of an etched micro-structure of titanium powder. This is consistent with the limited training time and the small dataset size. The segmentation results from the two models clearly

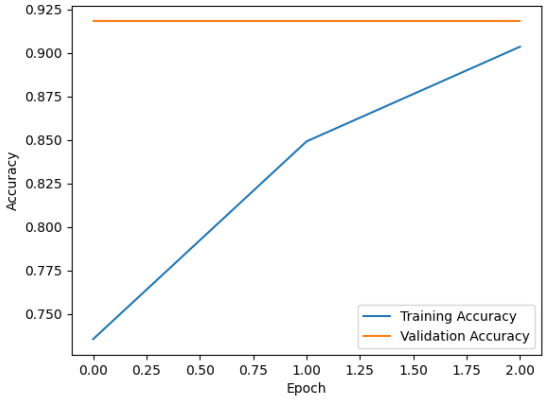
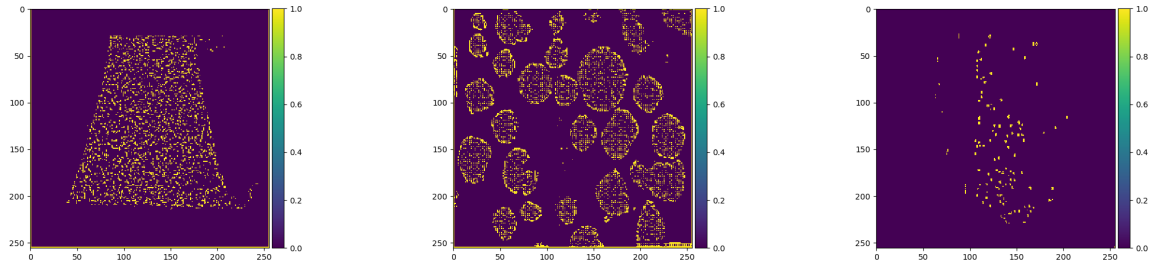


Figure 4.9: Training accuracy and validation accuracy of the second model. The blue line represents the training accuracy curve and the orange line represents the validation accuracy.

demonstrate the impact of dataset on segmentation accuracy.



(a) Segmentation of an image of a x-ray computed tomography images of a polyamide 66 reinforced by glass fibers.

(b) Segmentation of an image of an etched micro-structure of titanium powder.

(c) Segmentation of an image of an isolated carbon fiber.

Figure 4.10: The three figures shown the segmentation task performed with the second model.

The third U-Net design, exhibited in Figure 4.11, utilizes a 256x256 pixel image from the third dataset as the input layer. This U-Net architecture includes an additional layer compared to the other two models. It begins with two convolution layers with the ReLu activation function, resulting in 128 feature maps, followed by maximum pooling. This process is repeated three times, resulting in 1024 feature maps. Afterward, a sequence of up-convolutions, convolution layers with the ReLu activation function and concatenations were applied, resulting in 128 features maps. Finally, a layer with sigmoid activation was employed to yield probabilities for each zone.

The model was trained using a batch size of 2 images per epoch, with 10 epochs to train the model. This resulted in an accuracy of 0.9065 and a validation accuracy of 0.9099. Figure 4.12 shows the validation accuracy and training accuracy of the third model. As with the previous experiments, these values are presented for completeness and should be supported by stronger evaluation in future work.

Figure 4.13 presents the segmentation results achieved using the third model. On these

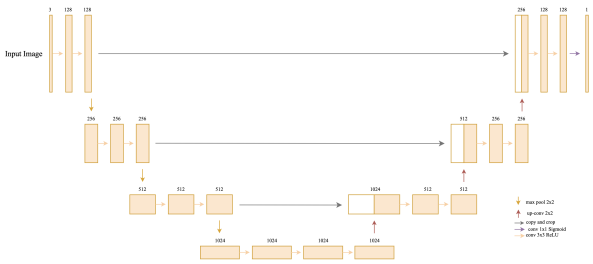


Figure 4.11: U-Net architecture for the third model.

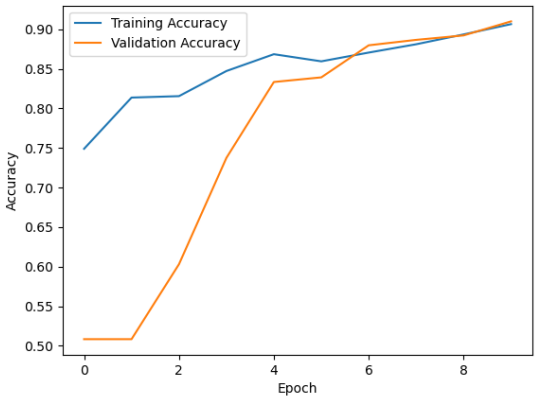
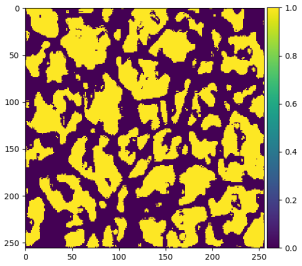
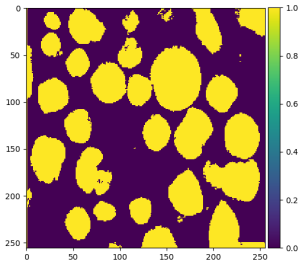


Figure 4.12: Training accuracy and validation accuracy for the third model. The blue line represents the training accuracy curve and the orange line represents the validation accuracy.

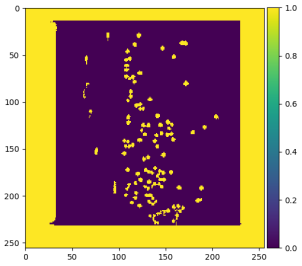
examples, the model performs well, successfully isolating both small parts of the image and larger objects, but broader conclusions require larger datasets and more robust metrics.



(a) Segmentation of an image of an artificial grains images, generated with a Voronoi Cell generation method.



(b) Segmentation of an image of an etched micro-structure of titanium powder.



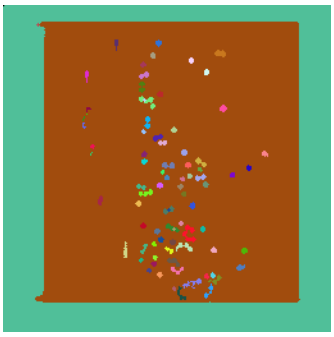
(c) Segmentation of an image of an isolated carbon fiber.

Figure 4.13: The three figures shown the segmentation task performed with the third model.

Given the segmentation applied to the images, using the third model, a unique color was assigned to each object in the image. In figure 4.14 are displayed the colored images obtained from the third model.

After applying colors to each object, it is possible to isolate one of them. Figure 4.15 displays one object isolated from each image.

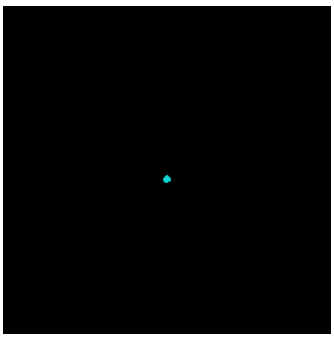
(a) Colored image of the segmentation of an etched micro-structure of titanium powder.



(b) Colored image of the segmentation of an isolated carbon fiber.

Figure 4.14: Colors assigned to the segmented images obtained from the third model.

(a) Isolated object from the colored image of an etched micro-structure of titanium powder.



(b) Isolated object from the colored image of an isolated carbon fiber.

Figure 4.15: Isolated objects from the colored images.

After thresholding and connected-component labeling, each object can be isolated from the colored overlays, ensuring that boundaries are well defined and small artifacts are suppressed. These masks are the geometric specification we need to move from pixels to elements.

In the next stage we convert each binary mask into a computational mesh suitable for finite element analysis. The result is a P1 triangular mesh that conforms to the segmented regions and can be used directly in downstream simulations. This image-to-mesh pipeline links the learned segmentations to the mechanics models developed later, enabling a consistent path from raw microscopy to analysis-ready discretizations.

4.2 Image-Based Mesh Generation

The generation of meshes is a central task in finite element analysis, as it provides the discretization of a given domain into simple geometric entities that can be used for numerical computation. In this work, particular attention is given to meshes obtained directly from image data, where segmented regions are converted into computational grids suitable for simulations.

Several approaches exist for mesh generation, ranging from structured grids to more flexible

unstructured methods. The present chapter, which follows the methodology proposed in [31], introduces the mathematical foundations required for building meshes from images, including image binarization, point distribution, Delaunay triangulation, and smoothing procedures.

The process begins with image binarization, in which a grayscale image is transformed into a binary one, separating the pixels into two regions based on their intensity. An image is mathematically represented as a two-dimensional array of pixels, each corresponding to a point in the Cartesian plane. For any pixel intensity function $I(x, y)$, the binarized version is defined by a threshold function T:

$$I_{\text{binary}}(x, y) = \begin{cases} 255 & \text{if } I(x, y) > T, \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

With the segmented region identified, the next step is grid generation. In the case of a structured grid, the image is discretized into a regular lattice of points. For an image of width W and height H, points are placed at uniform intervals of ten pixels, such that

$$x_i = i \times \Delta x, \quad y_j = j \times \Delta y, \tag{4.2}$$

with $\Delta x = \Delta y$. The resulting grid serves as a set of candidate vertices for triangulation. To ensure that only points belonging to the region of interest are retained, the binary mask is applied as a filter. A point $p = (x_p, y_p)$ is included if

$$I_{\text{binary}}(x_p, y_p) = 255, \tag{4.3}$$

leading to a subset of grid points that conform to the shape of the white zone. Using set notation, this filtering step can be written as

$$\text{Filtered Points} = \{p \in \text{Grid Points} \mid I_{\text{binary}}(x_p, y_p) = 255\}. \tag{4.4}$$

This process results in a structured mesh, where the domain is divided into a regular arrangement of triangular elements. Such meshes are simple to construct and particularly suitable for geometries that align with a Cartesian grid. An example of a structured mesh obtained from a segmented square-shaped region, with $\Delta x = \Delta y = 10$, is shown in Figure 4.16.

Although straightforward, structured meshes have limitations: the spacing between points is fixed and cannot adapt to arbitrary geometries. For more complex shapes, an unstructured mesh is preferable, since it allows full control over the number and spacing of points both along the boundary and within the interior. Typically, a higher density of points is placed near the

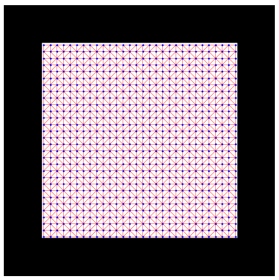


Figure 4.16: Example of a structured mesh generated from a segmented square region.

boundary to capture geometric detail, while the interior can be represented more sparsely.

Formally, let $B = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\}$ denote an ordered set of points along the boundary of the segmented region. To evenly distribute new points, each boundary segment $[b_i, b_{i+1}]$ of length $L_i = \|b_{i+1} - b_i\|$ is considered. A new point $\mathbf{p}_i$ is placed inside this segment according to

$$\mathbf{p}_i = b_i + t(b_{i+1} - b_i), \tag{4.5}$$

where

$$t = \frac{d - \text{current\_length}}{L_i}. \tag{4.6}$$

Here, d is the target spacing between consecutive points, and current_length is the accumulated distance along the boundary up to the beginning of the segment.

After distributing boundary points, the interior of the region is filled using Poisson disk sampling, which ensures that points are well spread out and respect a prescribed minimum spacing. The algorithm, summarized in Algorithm 1, takes as input the region of interest R, a minimum distance d_{min} between interior points, a minimum separation d_{boundary} between interior and boundary points, and the desired number of samples. Candidate points are generated randomly inside the bounding box of R and are only accepted if they satisfy the distance constraints relative to already-placed points and to the boundary.

With both boundary and interior points defined, the next step is Delaunay triangulation, a method that generates triangles such that no mesh point lies inside the circumcircle of any triangle. For a point set $P = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n\}$, this condition ensures well-shaped elements and avoids sliver triangles, which is particularly important for finite element computations.

Even with Delaunay triangulation, mesh quality can be further enhanced through Laplacian smoothing. Let B denote the set of boundary points of the polygon Polygon(B), where each edge is given by

Algorithm 1 Constrained Poisson Disk Sampling for Interior Points

Input: Region R, minimum interior distance $d_{\min}$, minimum boundary distance d_{boundary}, and number of samples.
Output: Uniformly distributed interior points in R.
1. Compute the bounding box of R.
2. Initialize `points` $= \emptyset$.
while size of `points` $<$ `num_samples`
  Generate a random candidate $q = (x, y)$ within the bounding box.
  if $q \in R$ and $\min_{p \in \texttt{points}} \|q - p\| > d_{\min}$
    if $\min_{b \in \texttt{boundary}} \|q - b\| > d_{\text{boundary}}$
      Add q to `points`.
    end if
  end if
end whilereturn `points`.

$$e_i = \{(1-t)b_i + tb_{i+1} \mid t \in [0,1]\}, \quad i = 1, \ldots, n, \tag{4.7}$$

with $b_{n+1} = b_1$ closing the loop. A set of points $P \subseteq \text{Polygon}(B)$ is used to generate the triangulation, and only those triangles whose centroids

$$\text{centroid} = \frac{\mathbf{p}_1 + \mathbf{p}_2 + \mathbf{p}_3}{3} \tag{4.8}$$

lie inside the polygon are retained. The mesh is then smoothed iteratively: each interior point is updated to the average position of its neighbors, while boundary points remain fixed. Formally, for a point $\mathbf{p}_i$ with neighbors $N(i)$, the update rule is

$$\mathbf{p}_i^{\text{new}} = \frac{1}{|N(i)|} \sum_{j \in N(i)} \mathbf{p}_j, \tag{4.9}$$

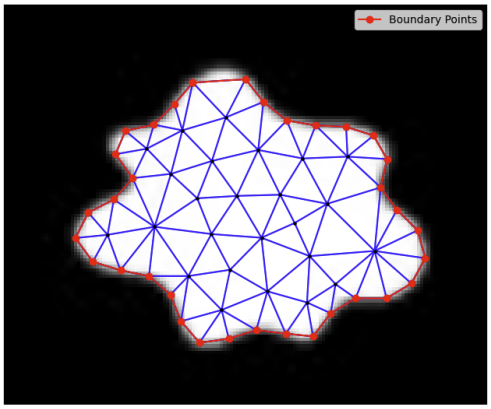
and this process is repeated for k iterations.

The result is a triangular mesh that conforms to the segmented white zone of the image. Structured and unstructured approaches each have advantages: structured grids are simpler and can yield finer triangulations but at higher computational cost, while unstructured grids allow adaptive control of density and are better suited for complex geometries.

Figures 4.17 and 4.18 show examples of Delaunay triangulation applied to irregular segmented shapes, producing meshes that successfully capture the boundaries while avoiding sharp angles. An application to a segmented femur bone is illustrated in Figure 4.19, where the generated mesh can be used in finite element simulations to compute deformations.

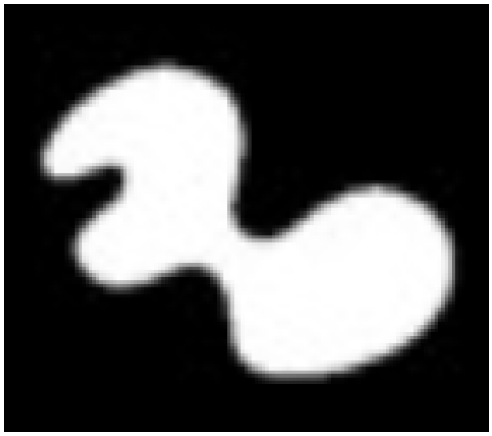
Building on the segmented outputs presented in Section 4.1 for the etched micro-structure of titanium powder and the isolated carbon fiber, we generated unstructured meshes for each

(a) Segmented image illustrating an irregular white shape.

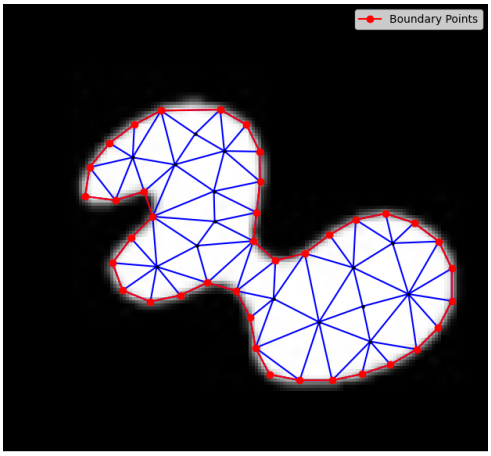


(b) Mesh generated for the irregular shape using Delaunay triangulation.

Figure 4.17: Results obtained from applying Delaunay triangulation to segmented images with irregular shapes.



(a) Segmented image of a second distinct irregular white shape.



(b) Resulting mesh for the second shape, created with Delaunay triangulation.

Figure 4.18: Additional results showcasing Delaunay triangulation applied to different irregular shapes.

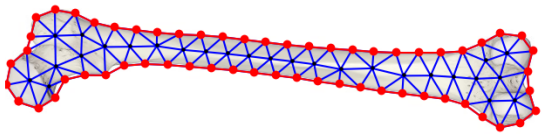
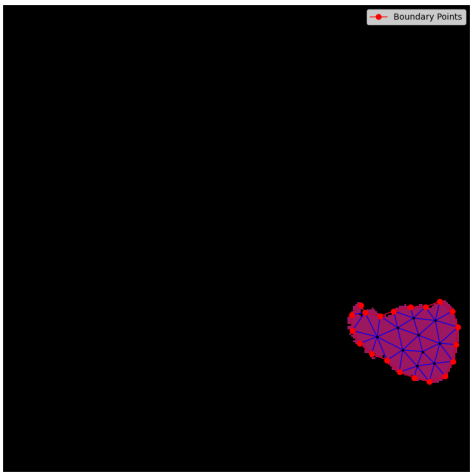


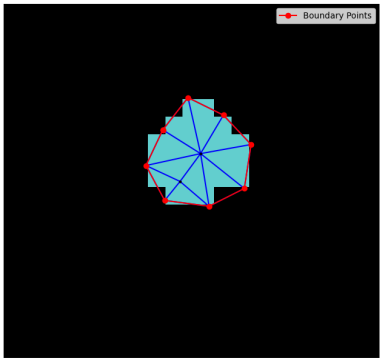
Figure 4.19: Generated mesh overlay for a segmented image of a femur bone, illustrating boundary and interior triangulation.

isolated object. Boundary samples are shown in red and Delaunay elements in blue; the filled blue region is slightly magnified to improve visual clarity.

Table 4.1 compiles the mesh-generation parameters used to produce the examples in Figures 4.16–4.20. For each image we report the number of boundary samples n_{boundary}, the number of interior samples n_{interior}, the minimum interior spacing d_{min}, and the distance between points in the boundary d_{boundary}.



(a) Generated mesh overlay for an isolated titanium-powder micro-structure.



(b) Resulting mesh for an isolated carbon fiber.

Figure 4.20: Generated mesh overlays corresponding to the isolated objects in Fig. 4.15.

Table 4.1: Mesh-generation settings for the examples in Figures 4.16–4.20.

Image (ref)	n_{boundary}	n_{interior}	d_{min}	d_{bdry}
Irregular shape 1 (Fig. 4.17)	35	25	10	10
Irregular shape 2 (Fig. 4.18)	40	15	10	10
Femur overlay (Fig. 4.19)	61	29	22	20
Titanium micro-structure (Fig. 4.20a)	20	10	7	7
Carbon fiber (Fig. 4.20b)	8	2	2	2

As shown in the examples above, the unstructured meshes provide explicit control via n_{boundary}, n_{interior}, d_{min}, and d_{boundary}, however, both quality and runtime are sensitive to these choices. Poisson sampling exhibits run-to-run variability unless a seed is fixed. Aggressive configurations, such as requesting many interior samples while enforcing a large d_{min} or using very dense boundary sampling, can stall the sampler or produce unnecessarily large meshes, increasing computational cost, whereas coarse settings risk missing thin features. In practice, modest smoothing and triangle-quality filtering are often needed to remove slivers and preserve small details.

The image-to-mesh pipeline completes the geometric stage of the workflow. Starting from segmentations, we extract clean boundaries, sample the contour and interior under spacing controls, and build Delaunay triangulations with light smoothing. The result is a set of analysis-ready discretizations that respect the native image geometry and can be queried at nodal and elemental level.

With meshes in hand we move to fast surrogates for the elastic response. We begin with a grid-based model. Finite element loads and displacements are rasterized onto a fixed Cartesian

lattice so a convolutional U-Net can learn the mapping from forces to displacements. This keeps all samples on a common resolution, enables efficient batching, and yields very rapid inference once trained. Section 4.3 presents the gridded force-to-displacement formulation, the U-Net configuration, and a quantitative comparison against the finite element reference on a set of rectangular domains.

4.3 Convolutional U-Net for Force-to-Displacement Mapping

This section introduces a grid-based surrogate for the mechanical response. Building on the regular meshes used for the finite element simulations, we project the FEM inputs and outputs (forces and displacements) onto a fixed Cartesian grid and train a U-Net to approximate the mapping from gridded forces to the corresponding displacement field. This approach leverages the approximate translation invariance of convolutions and keeps all data on a single, consistent grid, enabling fast predictions of the displacement field.

As ground truth we use the linear–elastic model from the FEM chapter (Hooke's law and weak form; see Sec. 2.2). The domain Ω is a rectangular plate in plane stress (Sec. 2.1); rigid–body modes are removed by clamping the four corner nodes, and tractions are applied on short four–node patches along the right and top edges. For each randomized traction profile we solve the forward problem on a $P2$ mesh, take the nodal displacements as reference, rasterize the tractions into input channels, and interpolate the displacements onto the Cartesian grid to form the CNN targets. When reporting derived fields, strains and stresses are computed a posteriori from the (FEM or CNN–predicted) displacement using the standard $P2$ post–processing (Eqs. (2.21)–(2.22)) under plane stress, with the engineering shear convention $\gamma_{xy} = 2\varepsilon_{xy}$.

U-Net–like architectures have demonstrated strong performance on large-scale inputs and have recently been applied to mechanical response prediction as well [32]. Earlier in this thesis, a U-Net was used for image segmentation; here, the same family of architectures is adapted to learn forward mappings from loads to deformations.

A practical limitation is that conventional CNNs operate naturally on regular grids and do not straightforwardly accommodate unstructured mesh inputs [30]. A simple workaround is to insert a structured grid over the domain and define a naïve mapping between unstructured-mesh quantities and grid-aligned arrays [30]. In this chapter, the proposed CNN U-Net is trained on force–displacement FEM datasets available in mesh format; once trained, the CNN can quickly and accurately approximate mechanical responses to new external forces [30]. In practice, we rasterize tractions (and, if needed, geometry masks) into input channels, interpolate FEM nodal displacements onto the grid for supervision, and at evaluation time sample the predicted grid

field back at FEM nodes to compute errors on the native mesh.

The U-Net used, developed by Saurabh D. et al. [30] and shown in Figure 4.21, follows an encoder–decoder pattern with skip connections. Initially, two convolutional layers with ReLU activation produce 64 feature maps, followed by max pooling that halves the spatial resolution. This pattern is repeated twice, yielding 128 and then 256 feature maps. After another max pooling operation, two convolutional layers with ReLU produce 256 feature maps at the bottleneck. The expansive path applies a sequence of up-convolutions and convolutional layers with ReLU, concatenating high-resolution encoder features at each level to reconstruct spatial detail, first resulting in 128 feature maps and then 64 feature maps. The input–output pairs consist of applied forces and the corresponding displacement fields from the training set.

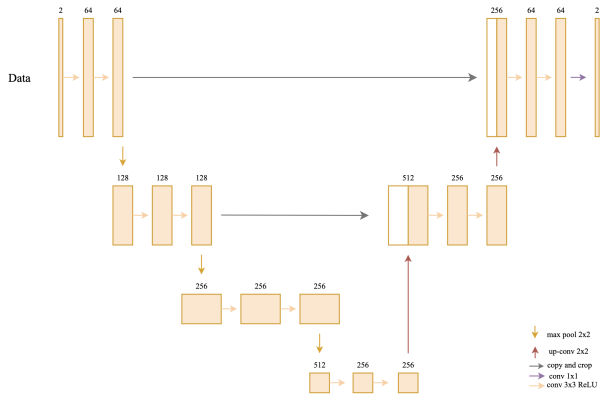


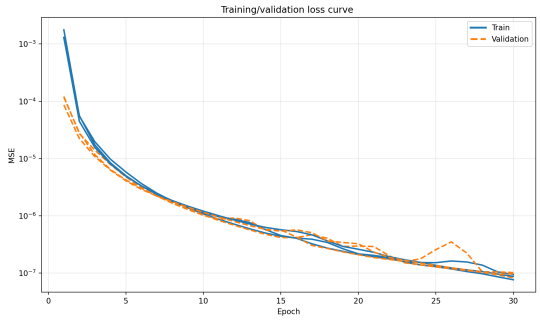
Figure 4.21: U-Net model architecture to accelerate simulations in mechanics [30].

Across the datasets considered below, the number of randomized cases is sufficiently large to promote generalization to unseen force configurations [30]. In this section we evaluate three examples using the same U-Net configuration; for each, the data are split 95%/5% for training/testing and the model is trained for 30 epochs with batch size 16. The material, mesh, and loading details for the three examples are summarized in Table 4.2.

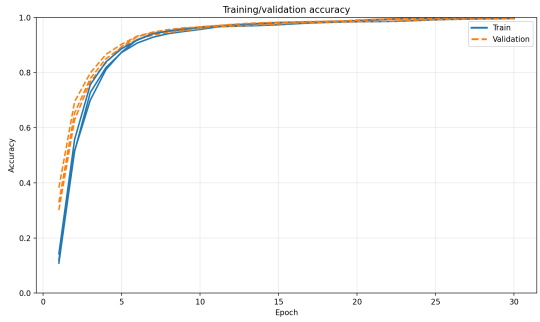
Table 4.2: Material, mesh, dataset, and loading used in the three examples.

Parameter	**Units**	**Mesh A**	**Mesh B**	**Mesh C**
Young's modulus, E	MPa	1e5	1e5	1e5
Poisson's ratio, ν	–	0.30	0.30	0.30
Domain size ($L_x \times L_y$)	m	0.8×3.2	3.2×0.8	3.2×0.8
Grid (vertices) ($n_x \times n_y$)	–	8×32	32×8	32×8
FEM element type	–	Q2 (quadrilaterals)	Q2 (quadrilaterals)	Q2 (quadrilaterals)
Samples (train / test)	–	95% / 5%	95% / 5%	95% / 5%
Traction boundary (4-node patch)	–	Right edge	Top edge	Top edge
Traction direction	–	x (horizontal)	y (vertical)	y (vertical)
Traction range per node	N/m	$[-12, -8]$	$[-12, -8]$	$[-12, -8]$

Figure 4.22 summarizes the training dynamics across the three mesh settings, reporting training/validation MSE and accuracy over epochs, where solid lines denote training and dashed lines denote validation. In the loss plot (left), the MSE decreases by several orders of magnitude on a logarithmic scale, where there is a rapid drop within the first few epochs followed by a slower, steady decay. Validation curves closely track the training curves, with only mild oscillations around mid-training and occasional small bumps near later epochs, indicating limited overfitting. In the accuracy plot (right), performance rises quickly from low initial values to above 0.95 within a few epochs and then saturates near 0.99; again, validation mirrors training. Overall, the learning curves are stable and consistent across meshes, suggesting the chosen architecture and schedule are sufficient for reliable generalization in the subsequent experiments.



(a) Training and validation MSE over epochs (y-axis in log scale).



(b) Training and validation accuracy over epochs.

Figure 4.22: Learning curves for the U-Net across the three mesh settings in Figs. 4.23, 4.25, and 4.27.

After confirming stable training, field quality is assessed visually. For each test case we display three scalar fields: displacement magnitude $|\mathbf{u}|$, a scalarized strain magnitude $\|\boldsymbol{\varepsilon}\|_2$ and a

scalarized stress magnitude $\|\boldsymbol{\sigma}\|_2$ obtained under plane stress, together with companion panels showing the corresponding absolute error maps.

Errors are reported per node as absolute differences, which is what the figures render. Displacement error is computed directly at nodes,

$$e_u(i) = \left\|\hat{\mathbf{u}}_i - \mathbf{u}_i^{\mathrm{FEM}}\right\|_2. \tag{4.10}$$

Strain and stress are first evaluated per element and then averaged to nodes; letting $\mathcal{T}(i)$ be the set of elements incident to node i,

$$e_\varepsilon(i) = \frac{1}{|\mathcal{T}(i)|} \sum_{e \in \mathcal{T}(i)} \left\|\hat{\boldsymbol{\varepsilon}}_e - \boldsymbol{\varepsilon}_e\right\|_2, \tag{4.11a}$$

$$e_\sigma(i) = \frac{1}{|\mathcal{T}(i)|} \sum_{e \in \mathcal{T}(i)} \left\|\hat{\boldsymbol{\sigma}}_e - \boldsymbol{\sigma}_e\right\|_2. \tag{4.11b}$$

To provide a compact scalar summary per case, we also report the overall relative L^2 error for each quantity $q \in \{\, \mathbf{u},\, \boldsymbol{\varepsilon},\, \boldsymbol{\sigma} \,\}$,

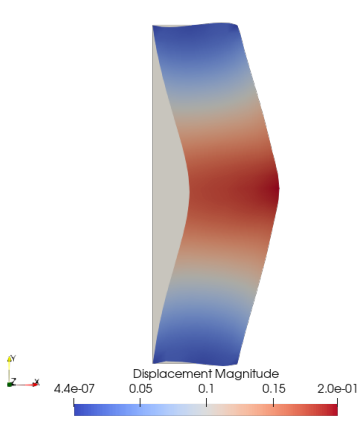
$$\mathcal{E}_{\mathrm{rel}}(q) = \frac{\left\|\hat{q} - q\right\|_2}{\left\|q\right\|_2} \times 100\%, \tag{4.12}$$

where the norm is taken over all nodes and components for displacement, and over all elements and components for strain and stress, matching the implementation. Here $\|\cdot\|_2$ is the discrete ℓ^2 norm over nodes/elements.

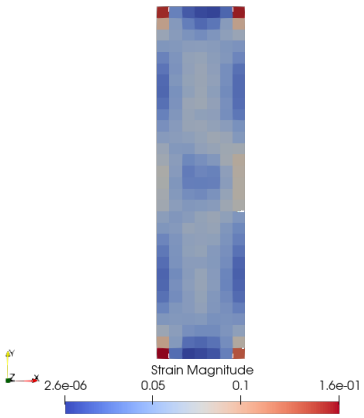
The metrics above quantify field fidelity (displacement, strain, stress) with respect to a finite-element reference. In engineering practice, structural performance is often summarized by energetic quantities (e.g., strain energy or compliance), safety factors, or failure criteria. These are straightforward to compute from the same fields when material models and design allow, and we view integrating such task-level metrics as complementary work to the present study, whose focus is on end-to-end field prediction fidelity and runtime.

Figures 4.23a, 4.23b and 4.23c show results obtained with the CNN U-Net trained on the force–displacement dataset from Saurabh D. et al. [30]. The domain (Mesh-A) is a rectangle discretized by an 8×32 mesh with 217 quadrilateral elements where the four corners are constrained.

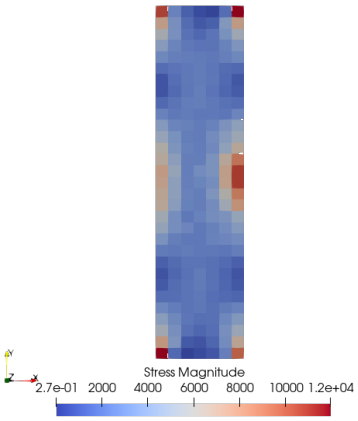
Figure 4.24 summarizes a representative test of the CNN U-Net on Mesh-A. The panels show absolute errors with respect to the FEniCS reference: (a) nodal displacement magnitude, (b) element-wise strain magnitude and (c) element-wise stress magnitude. Errors are concentrated near the loaded edge and along bands of large gradients, and remain small away from the traction.



(a) Predicted horizontal displacement (mm), shown on deformed mesh.



(b) Predicted strain magnitude, element-wise.

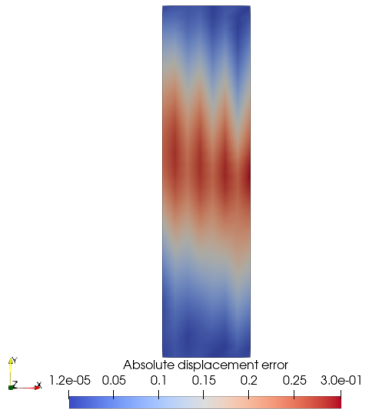


(c) Predicted stress magnitude, element-wise (Pa).

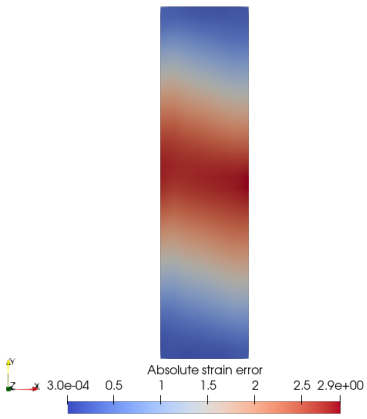
Figure 4.23: Predictions on Mesh-A plate (8×32 nodes) with four corner vertices fixed and traction applied on the right edge.

For this case, the global relative L^2 errors are approximately 19.82% for displacement, 20.63% for strain, and 20.90% for stress.

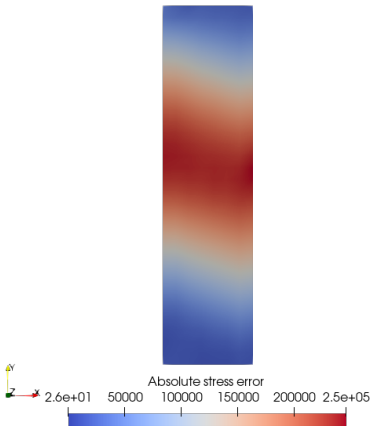
These larger percentages are consistent with the resolution and boundary conditions of Mesh-A, where the right-edge traction induces steep gradients across the short (x) direction, which is sampled by only 8 cells on the 8×32 grid, so the boundary layer and traction patch are under-resolved. Corner constraints combined with side-edge loading create localized stress concentrations that are difficult to capture on this coarse cross-section.



(a) Absolute displacement error (mm).



(b) Absolute strain error.

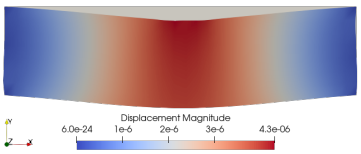


(c) Absolute stress error (Pa).

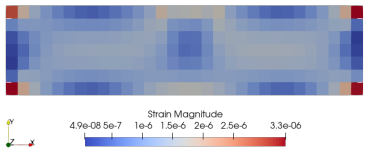
Figure 4.24: Absolute error fields on Mesh-A relative to the FEM reference.

Figures 4.25a, 4.25b, and 4.25c illustrate results from a CNN U-Net trained for Mesh-B, a rectangular plate discretized with a 32×8 grid (217 quadrilateral elements), with the four corner vertices fixed.

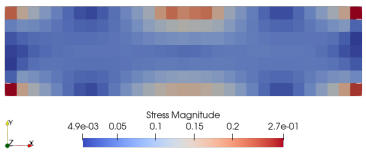
Figure 4.26 summarizes a representative test of the CNN U-Net on Mesh-B. Panels (a)–(c)



(a) Predicted horizontal displacement (mm), shown on deformed mesh.



(b) Predicted strain magnitude, element-wise.

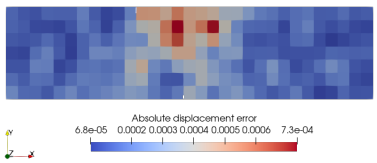


(c) Predicted stress magnitude, element-wise (Pa).

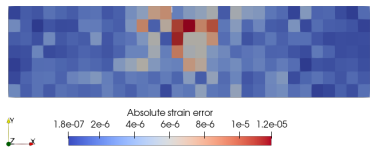
Figure 4.25: Predictions on Mesh-B plate (32×8 nodes), with four corner vertices fixed and traction applied on the top edge.

show absolute errors relative to the FEniCS reference: (a) nodal displacement magnitude, (b) element-wise strain magnitude, and (c) element-wise stress magnitude. Errors concentrate near the loaded edge and along bands of large gradients, and remain small away from the traction. For this case, the global relative L^2 errors are approximately 0.23% (displacement), 6.05% (strain), and 7.26% (stress).

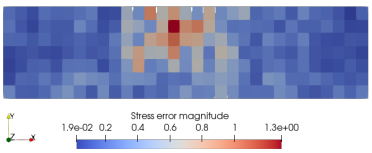
The lower displacement error here is consistent with how this case is sampled, where the top-edge traction and its short patch lie along the long edge of the rectangle, which is discretized more finely along x. This provides better in-plane resolution of the loaded boundary and of the lateral variation around the patch, reducing grid-induced aliasing and improving supervision near the critical region.



(a) Absolute displacement error (mm).



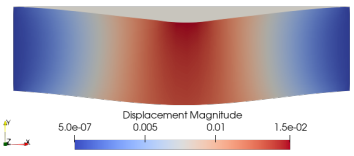
(b) Absolute strain error.



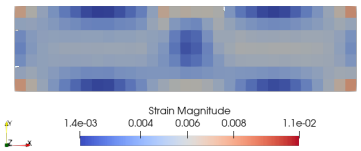
(c) Absolute stress error (Pa).

Figure 4.26: Absolute error fields on Mesh-B relative to the FEM reference.

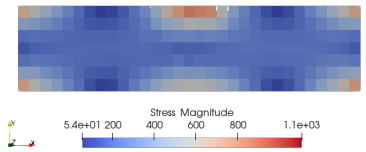
Figures 4.27a, 4.27b, and 4.27c present results from a CNN U-Net trained for Mesh-C, a rectangular plate discretized into a 32×8 grid (217 quadrilateral elements), where the left and right edges are clamped.



(a) Predicted horizontal displacement (mm), shown on deformed mesh.



(b) Predicted strain magnitude, element-wise.

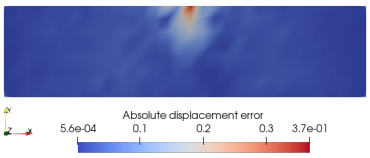


(c) Predicted stress magnitude, element-wise (Pa).

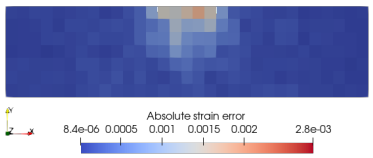
Figure 4.27: Predictions on Mesh-C plate (8×32 nodes) with left and right edges clamped and traction applied on the top edge.

Figure 4.28 summarizes a representative test of the CNN U-Net on Mesh-C. The panels show absolute errors with respect to the FEniCS reference: (a) nodal displacement magnitude, (b) element-wise strain magnitude and (c) element-wise stress magnitude. Errors are concentrated near the loaded edge and along bands of large gradients, and remain small away from the traction. For this case, the global relative L^2 errors are approximately 0.45% for displacement, 5.76% for strain, and 6.01% for stress.

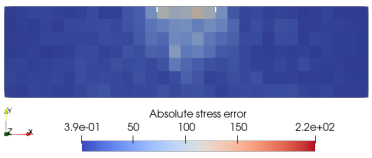
These values are consistent with how Mesh-C is sampled and constrained. The top-edge traction creates a dominant gradient across y; on the 8×32 grid this direction is well resolved, so the near-boundary layer beneath the load is captured with limited aliasing in the rasterized inputs and interpolated targets. In addition, clamping the left and right edges suppresses lateral rigid-body motion and reduces corner–load interactions. As expected, the derived fields show larger relative errors than displacement (about 6%), because strain differentiates $\mathbf{u}$ and stress scales with strain, which amplifies small nodal discrepancies in the boundary layer.



(a) Absolute displacement error (mm).



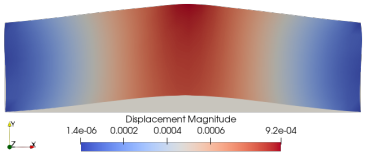
(b) Absolute strain error.



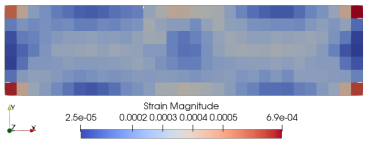
(c) Absolute stress error (Pa).

Figure 4.28: Absolute error fields on Mesh-C relative to the FEM reference.

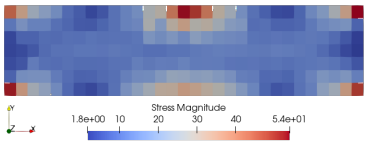
Using the model trained on the left–right clamped rectangular dataset, we apply a 30N equivalent nodal load on the matched 8×32 mesh. Predicted displacement, strain, and stress fields together with their absolute error maps are shown in Fig. 4.29 and Fig. 4.30. The global relative errors are 0.45% (displacement), 9.85% (strain), and 11.76% (stress).



(a) Predicted horizontal displacement (mm), shown on deformed mesh.



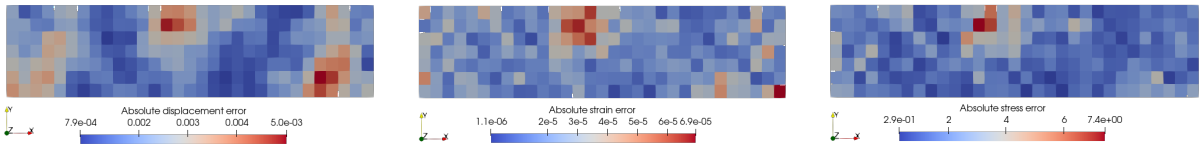
(b) Predicted strain magnitude, element-wise.



(c) Predicted stress magnitude, element-wise (Pa).

Figure 4.29: Predictions on matched rectangle (Mesh-B, 8×32 nodes), with left and right edges clamped and a **30 N** equivalent nodal load applied on the top edge (vertical), outside the 8–12 N training range.

Using the model trained on the corners-fixed rectangular dataset, we apply a 30 N load on the matched 32×8 mesh. Predictions and error maps are given in Fig. 4.31 and Fig. 4.32. The global relative errors are 1.72% (displacement), 33.35% (strain), and 42.53% (stress). The larger



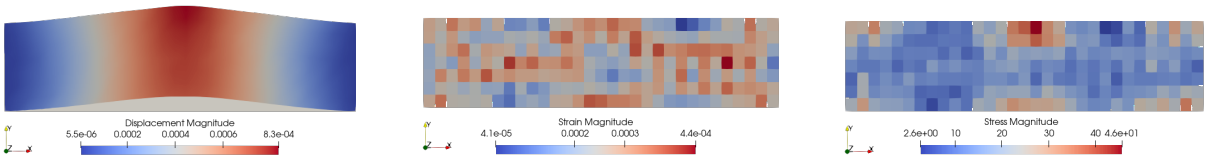
(a) Absolute displacement error (mm).

(b) Absolute strain error.

(c) Absolute stress error (Pa).

Figure 4.30: Absolute error fields for the 30 N case on Mesh-B (matched rectangle) relative to the FEM reference.

percentages for strain and stress reflect the amplification of small displacement discrepancies by spatial differentiation and by the constitutive relation.

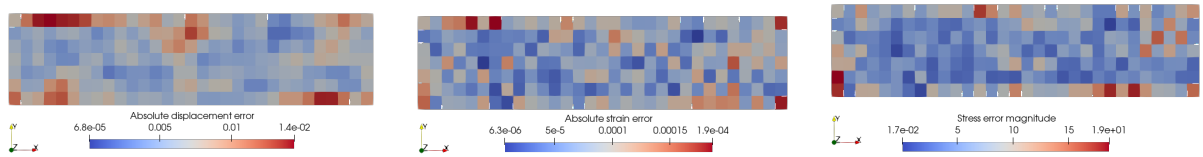


(a) Predicted horizontal displacement (mm), shown on deformed mesh.

(b) Predicted strain magnitude, element-wise.

(c) Predicted stress magnitude, element-wise (Pa).

Figure 4.31: Predictions on matched rectangle (Mesh-C, 8×32 nodes), with left and right edges clamped and a **30 N** equivalent nodal load applied on the top edge (vertical), outside the 8–12 N training range.



(a) Absolute displacement error (mm).

(b) Absolute strain error.

(c) Absolute stress error (Pa).

Figure 4.32: Absolute error fields for the 30 N case on Mesh-C (matched rectangle) relative to the FEM reference.

These two magnitude-extrapolation tests show that displacement remains accurate, while the derivative-based fields are more sensitive to loads outside the training range; the spatial pattern of the error consistently peaks near the applied traction and along high-gradient paths.

Across the three benchmarks the U-Net preserves the displacement field and tracks the FEM reference on regular layouts. Mesh A shows larger values with relative L^2 errors around 19.8% for displacement and about 21% for strain and stress. This matches the sampling of the short direction by only eight cells which under resolves the boundary layer beneath the right edge traction and amplifies the effect of corner clamps. On Mesh B and Mesh C the displacement error drops to about 0.2–0.5% and strain and stress sit near 6–7%. Error maps peak at the loaded edge and along paths of high gradient and decay in the interior. The difference between Mesh A and Mesh B reflects how the main gradient aligns with the grid and with the boundary

conditions rather than a pure rotation of the same case.

Out of range loads at 30 N confirm the same trend. Displacement remains within about 0.5–1.7% while strain reaches about 10% on the matched rectangle and can rise above 30% on the configuration with corner clamps and short cross section. The spatial pattern does not change and the largest discrepancies stay near the traction patch and in narrow boundary layers. Differentiation of $\mathbf{u}$ magnifies small nodal differences which explains the gap between displacement and derivative based fields.

Sensitivity to grid resolution is monotone with coarser grids increasing error. Perturbations in the segmentation affect mainly the band near the boundary while the interior stays stable. Inference runs in milliseconds on a single GPU and supports rapid sweeps of geometry and loading patterns. Training for 30 epochs with batch size 16 is enough for smooth convergence and the validation curves track training which suggests limited overfitting.

These observations motivate the use of a mesh native surrogate. Grid discretization blurs sharp boundaries and short traction patches and this is acceptable for coarse displacement summaries yet it becomes visible in strains and stresses near boundary layers. A mesh based encoder–decoder such as MAgNET keeps computation on the native discretization and sustains accuracy around complex geometry and steep gradients while keeping inference fast. This clarifies when a grid model is sufficient and when a mesh model is preferable and it aligns with the error maps and the extrapolation tests reported above.

4.4 MAgNET for Mesh-Based Force-to-Displacement Mapping

The convolutional U-Net demonstrates strong performance on structured grids but cannot directly operate on irregular, unstructured meshes such as those arising from image-driven meshing pipelines. This motivates MAgNET framework [33, 43], which extends the U-Net concept to graphs by replacing regular convolutions with graph-based aggregations, so that the network can process arbitrary mesh topologies without embedding them into a grid. Architecturally, MAgNET follows the encoder–decoder principle of U-Net entirely on graphs, where the encoder alternates multi-channel aggregation (MAg) layers with graph pooling (gPool) to progressively contract the graph and capture long-range dependencies with reduced computational cost; the decoder mirrors this process with graph unpooling (gUnpool) and further aggregation to restore resolution, with skip connections between matching encoder and decoder stages to preserve fine-scale structure. The final prediction head uses a linear activation so that the output lives on the original graph with task-specific channels. In Fig. 4.33 we illustrate the MAgNET used in this work.

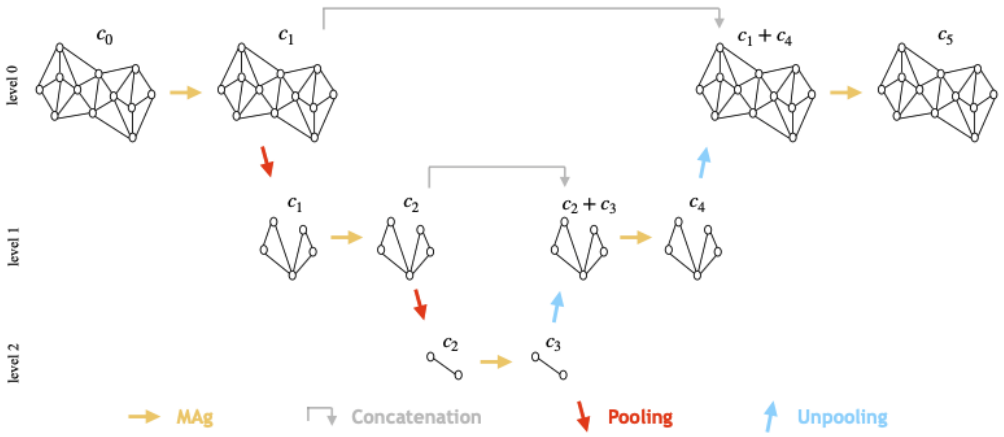


Figure 4.33: MAgNET architecture. Yellow segments denote multi-channel aggregation (MAg) layers; red arrows indicate graph pooling (gPool); blue arrows indicate graph unpooling (gUnpool); grey brackets mark skip concatenations. Channel counts per stage are annotated as $c_0, \ldots, c_5$. A linear node-wise head follows the final block (source: [33]).

Formally, the Graph U-Net network $\mathcal{G}$ is built as a stack of transformations on a sequence of graph-structured feature fields. The input layer d^0 consists of N nodes, each provided with a feature vector of length c^0. For each layer d^l, the transformation from d^{l-1} is

$$d^l = T^l(d^{l-1}; \theta^l), \tag{4.13}$$

where T^l denotes either a MAg, gPool, or gUnpool operation (with skip concatenations at appropriate decoder levels), and θ^l are the corresponding trainable parameters. The final output d^L has the same graph connectivity as the input, possibly with a different number of channels c^L, and is produced with a linear activation. The overall mapping is

$$\mathcal{G}(d^0, \theta) = d^L = T^L(T^{L-1}(\ldots T^1(d^0; \theta^1)); \theta^L), \qquad \theta = \bigcup_{l=1}^{L} \theta^l. \tag{4.14}$$

The graph structure used by MAgNET is encoded by a sparse Boolean, symmetric adjacency matrix $A \in \{0,1\}^{N \times N}$ with self-loops ($A_{ii} = 1$). Beyond the mesh wire-frame, all nodes belonging to the same finite element are made mutually adjacent, i.e., intra-element cliques are added to reflect strong local coupling. No edge attributes are used, only node features and the adjacency enter the computation. This adjacency construction simplifies aggregation and pooling, supports stable training, and enables optional use of higher adjacency powers. The partitioning used by pooling is computed once from A and reused across the hierarchy.

The multi-channel aggregation (MAg) layer generalizes convolution by aggregating information from each node's neighborhood defined by A. Writing $\mathcal{N}_i = \{j : A_{ij} = 1\}$ for the neighbors of

node i, the update of node i at channel α can be expressed as

$$d_{i,\alpha}^{l+1} \;=\; \sigma\left(b_{i,\alpha}^{l+1} \;+\; \sum_{\beta=1}^{c_l} \sum_{j\in\mathcal{N}_i} k_{i,j,\alpha,\beta}^{l+1}\, d_{j,\beta}^{l}\right), \tag{4.15}$$

with learnable weights $k_{i,j,\alpha,\beta}^{l+1}$ and biases $b_{i,\alpha}^{l+1}$, and a pointwise nonlinearity $\sigma(\cdot)$. Unlike conventional convolutions that share kernels across spatial locations, MAg uses input-independent but non-shared local weights (no global weight sharing), yielding high capacity on irregular topologies while remaining purely local. To accelerate information propagation on graphs with large diameter, MAgNET optionally enlarges the aggregation support by replacing A with a higher power A^k ($k \geq 2$) inside the aggregation, thereby expanding each node's effective neighborhood without requiring excessively deep stacks. From an information-passing viewpoint, stacking MAg layers and/or using A^k grows the receptive field, while pooling reduces the effective graph diameter.

Pooling and unpooling operate purely at the topological level. The encoder partitions the graph into non-overlapping fully connected subgraphs (cliques) using a static, data-independent scheme determined from A, and contracts each clique to a single node (gPool). Feature aggregation is applied per channel (e.g., max or average) and preserves the channel count. The decoder then replicates pooled features back to the original nodes of each stored clique (gUnpool), after which the unpooled features are concatenated with the corresponding encoder features via the skip connection at that level. This concatenation increases the channel dimension at the decoder stage, mirroring the behavior of U-Net on images while remaining fully graph-based. The clique pooling/unpooling mechanics are depicted in Fig. 4.34.

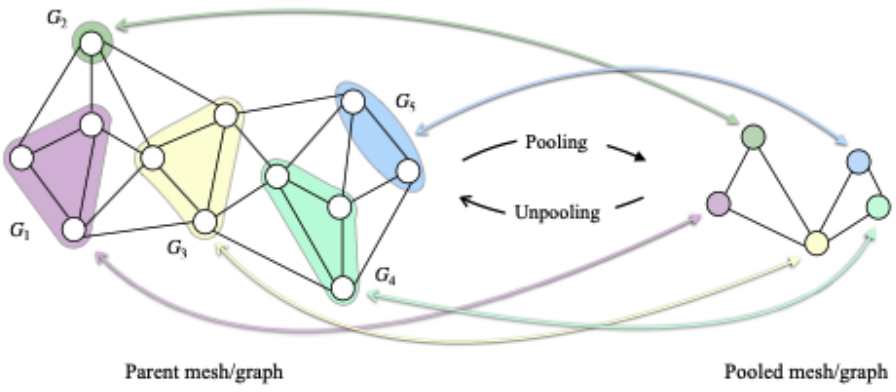


Figure 4.34: An example partition of the graph into disjoint subgraphs used for pooling.

MAg's input-independent, non-shared local weights provide a flexible alternative that fits naturally within the U-Net encoder–decoder with skip connections, and the static clique pooling and unpooling preserve a clear geometric interpretation throughout the hierarchy. In contrast

with Graph U-Nets that rely on top-k pooling [34], MAgNET's static, disjoint clique pooling aligns more directly with mesh elements and avoids feature-dependent pooling decisions, which motivates a controlled evaluation to assess how these choices affect performance.

In this section we train on four datasets, two of which coincide with the last two rectangular problems introduced in Sec. 4.3, where Mesh-B is a rectangular domain with the four corner nodes constrained and Mesh-C is a rectangular domain with the left and right edges constrained, and the remaining meshes are unstructured and consist of $P1$ triangles.

The MAgNET network, shown in Fig. 4.35, is the same for the four examples. It is trained end-to-end with mean-squared error (MSE) on the target fields and the Adam optimizer (initial learning rate 10^{-4}) together with the default monotone scheduler, which is constant for the first 10 epochs and then linearly decays until epoch 100. For the four cases considered here we use a batch size of 4 and run for 10 epochs each, and we retain the best checkpoint with the lowest validation MSE (see A for details). Inputs are vectors of nodal forces assembled from the traction patches and a file with the connectivity of the nodes, and outputs are vectors of nodal displacements. Strains and stresses are recovered a posteriori from the predicted displacement via standard $P1$ finite-element post-processing under plane stress (Eqs. (2.21)–(2.22)), adopting the engineering-shear convention $\gamma_{xy} = 2\varepsilon_{xy}$. Unlike the CNN pipeline, no **grid mapping** or grid interpolation is performed, and all quantities remain on the native unstructured mesh throughout training and evaluation.

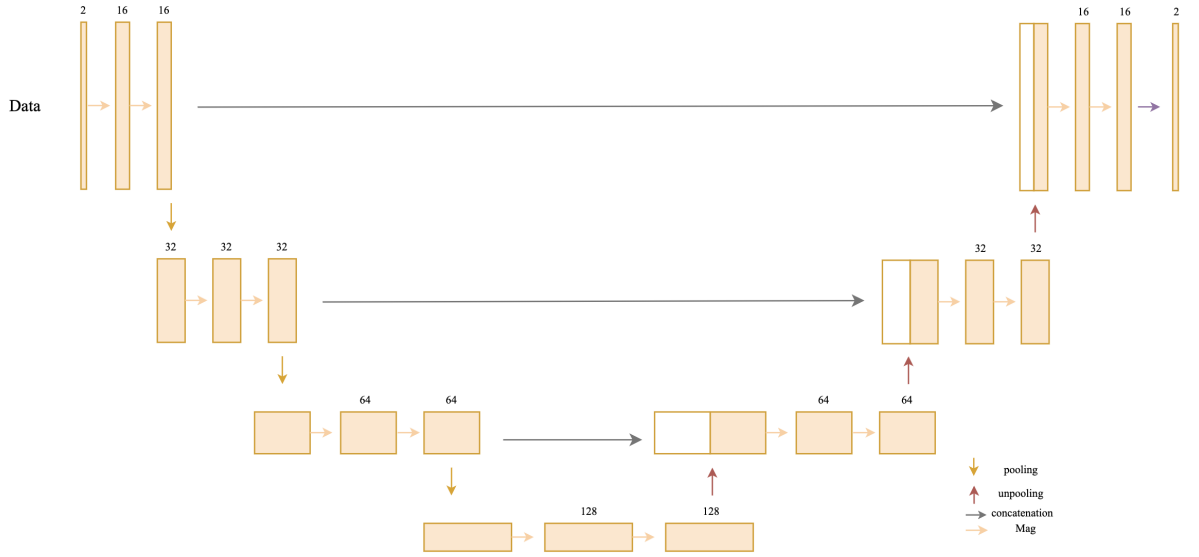
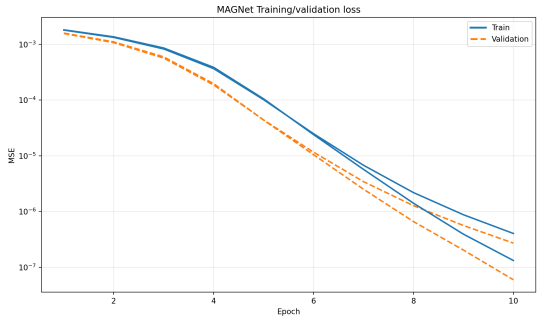


Figure 4.35: MAgNET architecture used in the experiments.

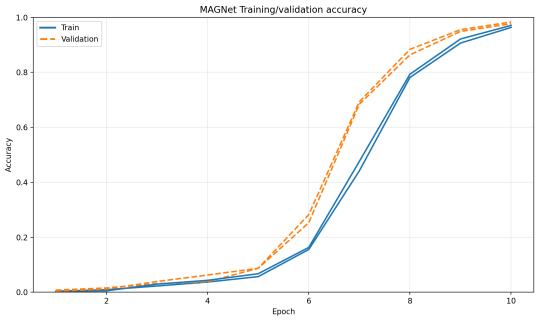
To enable a like-for-like comparison with the CNN baseline we use the same datasets as in the CNN section, and the geometry, loading edge, and loading direction match the CNN setup in

Table 4.2.

Figures 4.36a and 4.36b report MAgNET's training (solid) and validation (dashed) performance on the two rectangular meshes (Mesh-B and Mesh-C). In the loss plot both curves fall by roughly four orders of magnitude, from about 10^{-3} at epoch 1 to around 10^{-7} by epoch 10. The largest decrease occurs between epochs 4 and 8, and the validation loss stays close to or slightly below the training loss after epoch 6, which suggests good generalization. In the accuracy plot both curves remain near zero through epoch 5, then rise steeply with a clear knee around epochs 6–7, and approach 0.98–0.99 by epoch 10, with validation marginally higher. Taken together, these trends indicate fast early optimization and stable convergence without signs of overfitting.



(a) Training and validation MSE (log-scale).



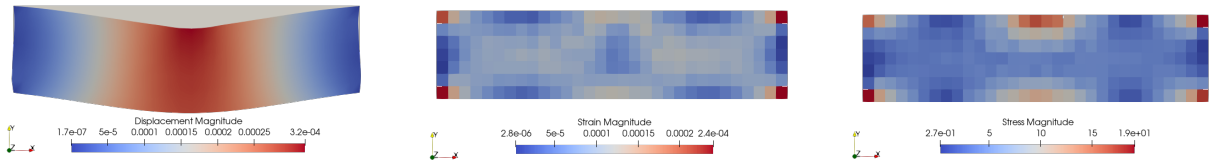
(b) Training and validation accuracy.

Figure 4.36: MAgNET learning curves across the two mesh settings in Table 4.2.

Figure 4.37 summarizes a representative Mesh B test with MAgNET. Panels 4.37a–4.37c show the predicted horizontal displacement on the deformed mesh together with the element-wise strain and stress magnitudes. The largest responses appear near the loaded top edge and along narrow bands where the solution varies most rapidly, while the interior away from the load remains comparatively smooth.

Figure 4.38 shows the absolute error maps for the same sample. Displacement, strain, and stress errors form compact hotspots around the short traction patch and follow the high-gradient directions, with low residuals elsewhere. The footprint is similar to the CNN U-Net on the same setup, which indicates that both surrogates confine discrepancies to the physically demanding regions.

For this case MAgNET yields global relative L^2 errors of **0.31%** for displacement, **6.8%** for strain, and **7.9%** for stress. The CNN U-Net on Mesh B attains **0.23%**, **6.05%**, and **7.26%**. The gap is modest but consistent, especially for the derivative-based fields. A likely reason is that graph pooling and multi-hop aggregation can smooth sharp boundary layers on small meshes, whereas the grid-aligned CNN benefits from a sampling pattern that resolves the traction patch more finely along the dominant gradient direction.

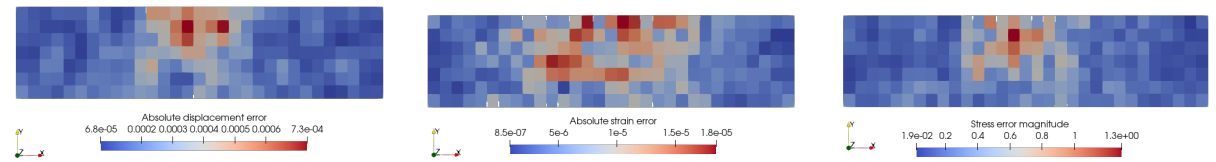


(a) Predicted horizontal displacement (mm), shown on deformed mesh.

(b) Predicted strain magnitude, element-wise.

(c) Predicted stress magnitude, element-wise (Pa).

Figure 4.37: MAgNET predictions on Mesh B (32×8 nodes) with corner vertices fixed and a top-edge vertical traction; displacement is shown on the deformed mesh.



(a) Absolute displacement error (mm).

(b) Absolute strain error.

(c) Absolute stress error (Pa).

Figure 4.38: Absolute error fields on Mesh B (MAgNET) relative to the FEM reference.

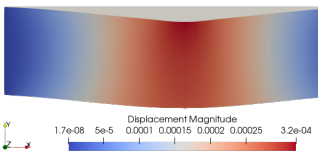
Figure 4.39 presents a representative Mesh C trial with MAgNET. Panels 4.39a–4.39c display the predicted horizontal displacement on the deformed mesh together with element-wise strain and stress magnitudes. Strong responses cluster beneath the loaded edge and along thin bands of rapid spatial variation, with comparatively smooth behavior away from the traction.

Figure 4.40 shows the corresponding absolute error fields. Residuals in displacement, strain, and stress concentrate around the short traction patch and align with high-gradient paths, remaining small elsewhere. This pattern mirrors the CNN case and highlights the same physically challenging zones.

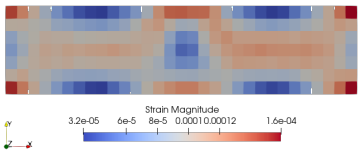
For this case MAgNET reaches global relative L^2 errors of **0.42%** for displacement, **7.9%** for strain, and **9.3%** for stress. The CNN U-Net on the same setup achieves **0.06%**, **2.8%**, and **3.5%**. Two effects likely contribute to the higher MAgNET values, namely clique pooling and unpooling that diffuse steep boundary layers, and enlarged aggregation neighborhoods that help long-range communication but can blur gradients near the traction patch. The CNN, in contrast, operates on a grid that aligns with the principal gradient direction on Mesh C, which helps preserve edge detail under grid discretization and yields sharper strain and stress reconstructions.

We now repeat the magnitude–extrapolation test with MAgNET on the two rectangular benchmarks, applying a **30 N** top-edge traction that lies outside the 8–12 N training range. The qualitative fields and the absolute error maps are reported alongside the corresponding plots for the CNN baseline to enable a like-for-like comparison.

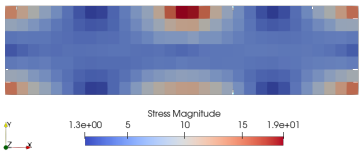
Figure 4.41 shows the prediction on the matched rectangle with corner nodes fixed (Mesh B), together with the element-wise strain and stress magnitudes. The companion maps in Figure 4.42



(a) Predicted horizontal displacement (mm), shown on deformed mesh.

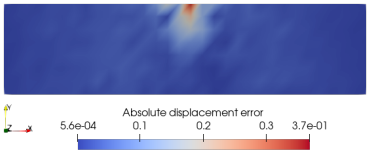


(b) Predicted strain magnitude, element-wise.

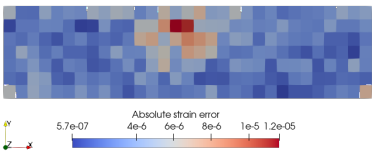


(c) Predicted stress magnitude, element-wise (Pa).

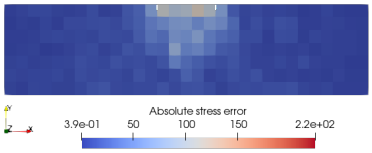
Figure 4.39: MAgNET predictions on Mesh C (8×32 nodes) with left and right edges clamped and a top-edge vertical traction; displacement is shown on the deformed mesh.



(a) Absolute displacement error (mm).



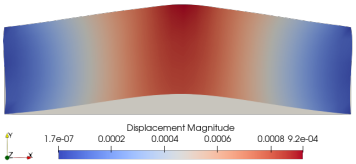
(b) Absolute strain error.



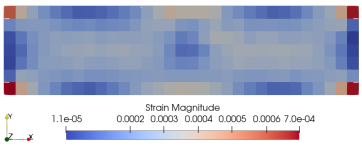
(c) Absolute stress error (Pa).

Figure 4.40: Absolute error fields on Mesh C (MAgNET) relative to the FEM reference.

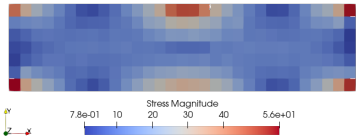
display the absolute errors. Residuals concentrate at the short traction patch and along high-gradient bands; away from the applied load the fields remain comparatively smooth. For this extrapolation case MAgNET attains global relative L^2 errors of **0.72%** in displacement, **14.8%** in strain, and **18.9%** in stress. On the same setup the CNN U-Net reports **0.45%**, **9.85%**, and **11.76%**. The gap is most visible in the derivative-based quantities, where small differences near the boundary layer are amplified by spatial differentiation and the constitutive mapping.



(a) Predicted horizontal displacement (mm) on the deformed mesh.

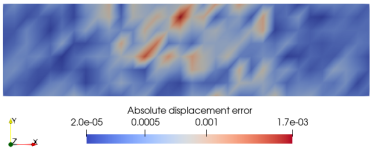


(b) Predicted strain magnitude, element-wise.

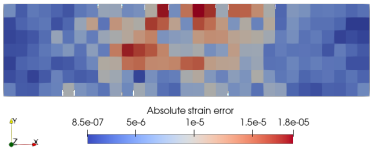


(c) Predicted stress magnitude, element-wise (Pa).

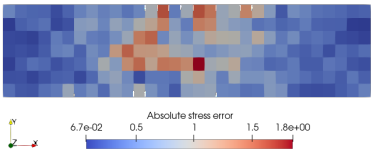
Figure 4.41: MAgNET predictions for Example A under a 30 N top-edge traction (outside the 8–12 N training range).



(a) Absolute displacement error (mm).



(b) Absolute strain error.

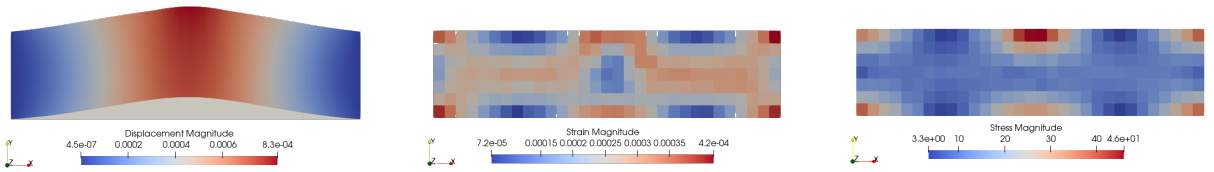


(c) Absolute stress error (Pa).

Figure 4.42: Absolute error fields for Example A under a 30 N top-edge traction relative to the FEM reference.

Figure 4.43 presents the corresponding **30 N** trial on the rectangle with left and right edges

clamped (Mesh C). Absolute error fields in Figure 4.44 again peak beneath the loaded edge and follow narrow paths of steep variation. MAgNET reaches global relative L^2 errors of **2.40%** for displacement, **45.0%** for strain, and **56.0%** for stress, whereas the CNN U-Net yields **1.72%**, **33.35%**, and **42.53%**. The larger percentages for MAgNET are consistent with a sharper boundary layer in the short direction for this constraint configuration, where graph pooling and enlarged aggregation neighborhoods can smooth steep gradients and slightly blur the traction imprint, which is then magnified in strain and stress.

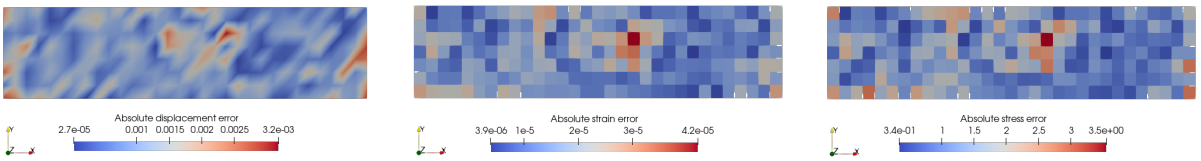


(a) Predicted horizontal displacement (mm) on the deformed mesh.

(b) Predicted strain magnitude, element-wise.

(c) Predicted stress magnitude, element-wise (Pa).

Figure 4.43: MAgNET predictions for Mesh C under a 30 N top-edge traction (outside the 8–12 N training range).



(a) Absolute displacement error (mm).

(b) Absolute strain error.

(c) Absolute stress error (Pa).

Figure 4.44: Absolute error fields for Mesh C under a 30 N top-edge traction relative to the FEM reference.

Taken together, these **30 N** tests confirm that displacement remains accurate for both surrogates, while strain and stress are more sensitive beyond the training range. Error hotspots stay localized at the applied traction and along high-gradient bands. Relative to the CNN on these rectangles, MAgNET shows higher derivative errors under extrapolation, a behavior that aligns with the interplay between mesh pooling, multi-hop aggregation, and the amplification inherent in computing strain and stress from the predicted displacement.

After comparing the CNN and MAgNET surrogates, Table 4.3 reports the wall-clock time required for training and for producing a single prediction. The computing environment is described in Appendix A. In our setup, the CNN trains in 850 s (14.17 min), whereas MAgNET requires 1800 s (30.00 min). At inference, the CNN attains 0.8 s per prediction and MAgNET 1.2 s per prediction; relative to the finite element method (FEM) baseline of 4.0 s, this corresponds to speed-ups of 5.00× and 3.33×, respectively.

Considering time-to-first prediction (training plus one inference), the CNN reaches 850.8 s

versus 1801.2 s for MAgNET, while FEM delivers a single solution in 4.0 s without any training phase. For workloads involving many queries, the training cost of the learned surrogates is quickly amortized. The break-even number of predictions N_{be} after which a surrogate becomes faster than repeatedly running FEM is

$$N_{\text{be}} \;\geq\; \frac{T_{\text{train}}}{t_{\text{FEM}} - t_{\text{infer}}},$$

yielding $N_{\text{be}} \approx 266$ for the CNN and $N_{\text{be}} \approx 643$ for MAgNET with the timings above. We emphasize that absolute times depend on hardware and implementation details, but the relative trend, faster training and inference for CNN, and substantial inference-time speed-ups over FEM for both surrogates, is consistent across runs.

Table 4.3: Training and inference time comparison (per-sample inference).

Method	Training [s]	Training [min]	Inference [s]	Speed-up vs FEM	Time-to-first [s]
MAgNET	1800.00	30.00	1.20	3.33	1801.20
CNN U-Net	850.00	14.17	0.80	5.00	850.80
FEM (ref.)	0	0	4.00	1.00	4.00

Inference times are per prediction. Speed-up is computed as FEM time/inference time.

Beyond the rectangular benchmark, all remaining domains are represented by unstructured P1 triangulations. As ground truth we again use the linear–elastic FEM model from the FEM chapter with Hooke's law and the weak form described in Sec. 2.2, in plane stress as in Sec. 2.1. Rigid-body modes are removed using the same clamping strategy described there, and tractions are prescribed on short four-node patches as detailed in Table 4.4. For the unstructured cases the domain boundary is extracted from a binary mask and triangulated by Poisson-disk interior sampling followed by Delaunay meshing and Laplacian smoothing, and the resulting point set is converted to a FEniCS P1 triangular mesh in the physical orientation. Rigid-body modes are then removed by Dirichlet constraints on a designated boundary, for example the left edge or the corner nodes in line with the case definition, and tractions are applied as equivalent nodal loads on short three to four node patches along the specified edge, as in Table 4.4. For each sample a force magnitude is drawn uniformly in [-12,-8] and applied with the prescribed sign and direction, for example negative y for top-edge vertical loading, which yields nodal force features on the affected DOFs. We pre-assemble the stiffness matrix once and reuse it across solves to accelerate dataset generation. Labels are the nodal displacement vectors, and compliance-style targets u/F are optionally stored. When reporting derived fields, strains and stresses are computed a

posteriori from the displacement via the standard P1 post-processing under plane stress, see Eqs. (2.21)–(2.22), adopting the engineering shear convention $\gamma_{xy} = 2\varepsilon_{xy}$. All quantities remain on the native unstructured mesh with no grid mapping, which aligns with MAgNET's mesh-native processing. For the matched rectangular case the geometry and boundary conditions are identical to the CNN setup, so any differences isolate the effect of the architecture (see Sec. 4.3).

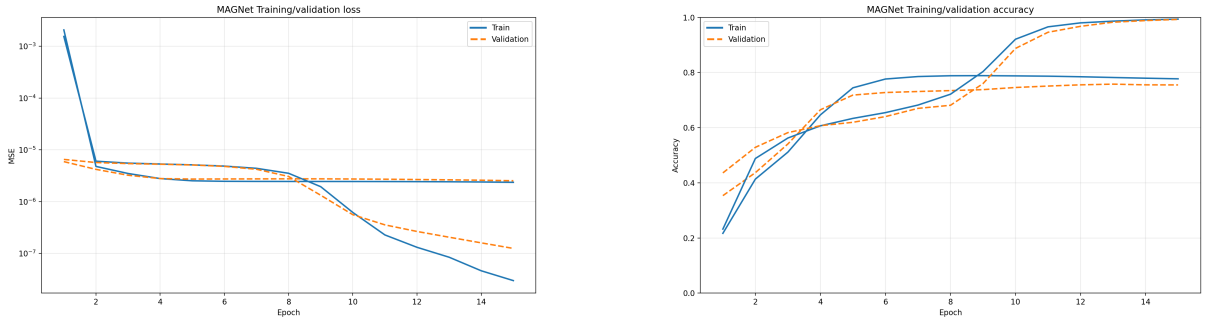
Table 4.4: Material, mesh, dataset, and loading used in the two MAgNET examples.

Parameter	**Units**	**Mesh D**	**Mesh E**
Internal case (MAgNET)	–	4.17	4.20a
Young's modulus, E	MPa	500	500
Poisson's ratio, ν	–	0.30	0.30
Domain size ($L_x \times L_y$)	m	–	–
Element type	–	$P1$ (triangles)	$P1$ (triangles)
Nodes / DOFs	–	59 / 118	29 / 58
Elements $\lvert\mathcal{T}\rvert$	–	–	–
BCs (constraints)	–	(as FEM setup)	(as FEM setup)
Traction boundary (patch)	–	Top edge (4 nodes)	Top edge (4 nodes)
Traction direction	–	y (vertical)	y (vertical)
Traction range per node	N/m	$[-8, 12]$	$[-8, 12]$
Samples (train / test)	–	80%/20%	80%/20%
Graph / training (MAgNET):			
gPool levels / channels	–	3 / $[16, 32, 64, 128]$	3 / $[16, 32, 64, 128]$
Aggregation support	–	A^2 (MAg, hop= 2)	A^2 (MAg, hop= 2)
Epochs / batch size	–	10 / 4	10 / 4
Loss / optimizer	–	MSE / Adam	MSE / Adam

The four channel counts $[16, 32, 64, 128]$ correspond to the encoder/decoder stages $c_0, \ldots, c_3$ in Fig. 4.35. The three pooling operations listed here match the three downsampling steps shown there.

Figures 4.45a and 4.45b report training (solid) and validation (dashed) performance for the two MAgNET cases. In the loss plot, both curves drop by roughly three orders of magnitude within the first two epochs, then remain nearly flat around 3–4 $\times 10^{-6}$ up to epoch 8. From epoch 9 onward, the training loss continues to decrease to $\approx 10^{-7}$, while the validation loss plateaus near 3×10^{-6}, opening a modest gap. In the accuracy plot, the training curve climbs

steadily to ≈ 1.0 by epochs 12–15; the validation curve increases more gradually and saturates around 0.75–0.80. Overall, these dynamics indicate rapid early optimization on the unstructured $P1$ meshes.



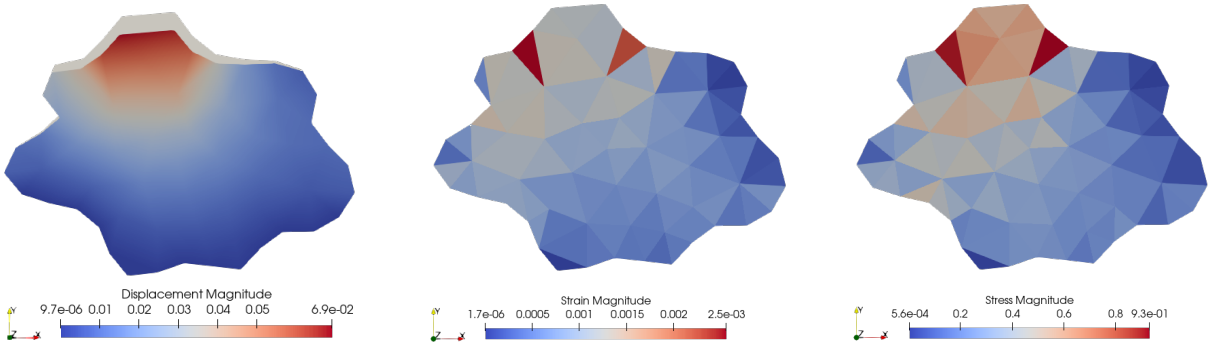
(a) Training and validation MSE (log-scale).

(b) Training and validation accuracy.

Figure 4.45: MAgNET learning curves across the two mesh settings in Figs. 4.46 and 4.48

To enable a direct comparison with the CNN baseline, we adopt exactly the same visualization protocol and error definitions as in Sec. 4.3 (Eqs. (2.21)–(2.22) and the nodal/element-wise error maps). Concretely, for each test case we plot the displacement magnitude $|\mathbf{u}|$, strain $\|\boldsymbol{\varepsilon}\|_2$, and stress $\|\boldsymbol{\sigma}\|_2$, together with their absolute error fields. The nodal errors for displacement and the element-wise errors for strain and stress, as well as the global relative L^2 error $\mathcal{E}_{\text{rel}}(q)$, are computed identically to the CNN section.

Figures 4.46 and 4.47 show a representative test for Example A. The first row displays (a) the predicted horizontal displacement u_x on the deformed mesh, and (b–c) the element-wise strain and stress error magnitudes. The largest errors appear near the right-hand tip of the shape, not exactly on the load patch. This is where the true field changes fastest, so small differences lead to a visible error pocket. Away from that region the errors are small. The global relative L^2 errors are 0.26% (displacement), 1.15% (strain), and 1.21% (stress).

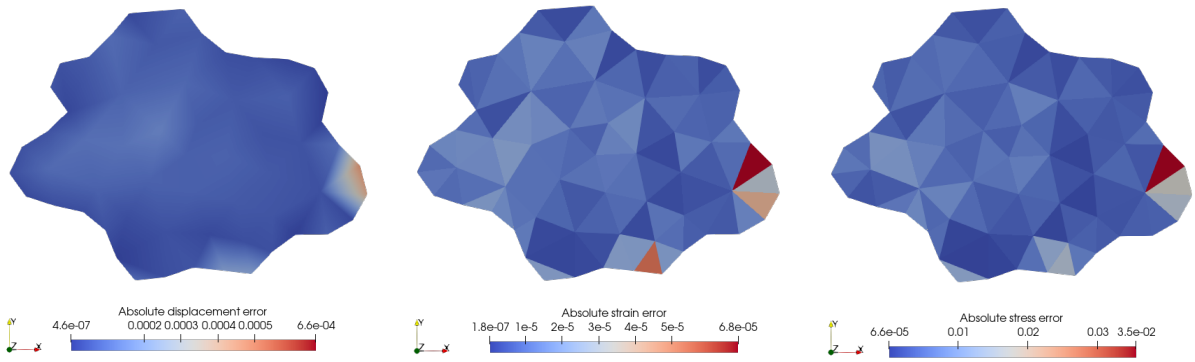


(a) Predicted horizontal displacement (mm), shown on deformed mesh.

(b) Predicted strain magnitude, element-wise.

(c) Predicted stress magnitude, element-wise (Pa).

Figure 4.46: MAgNET predictions on Example A: unstructured $P1$ triangular mesh, with traction applied on the top edge in y (vertical).



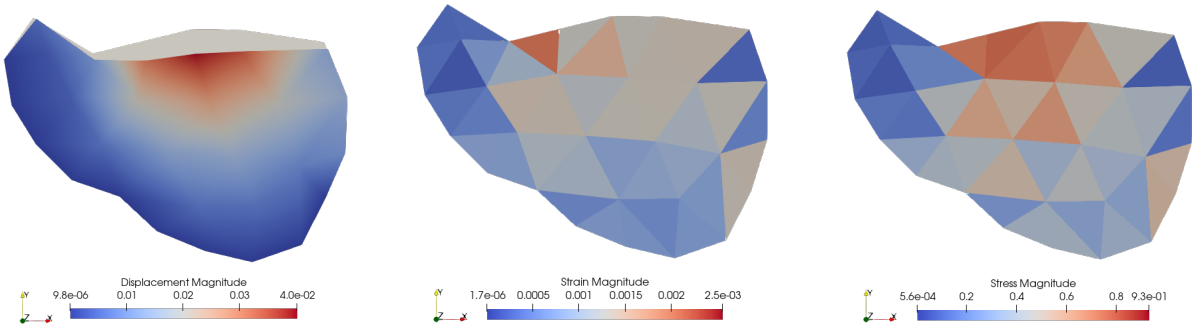
(a) Absolute displacement error (mm).

(b) Absolute strain error.

(c) Absolute stress error (Pa).

Figure 4.47: Absolute error fields for Example A relative to the FEM reference.

Figures 4.48 and 4.49 report Example B under the same vertical top-edge loading. Here the largest errors sit along the lower rim and near the bottom-right notch, where the field bends the most. As in Example A, the rest of the domain shows small errors. The global relative L^2 errors are 0.09% (displacement), 0.28% (strain), and 0.26% (stress).

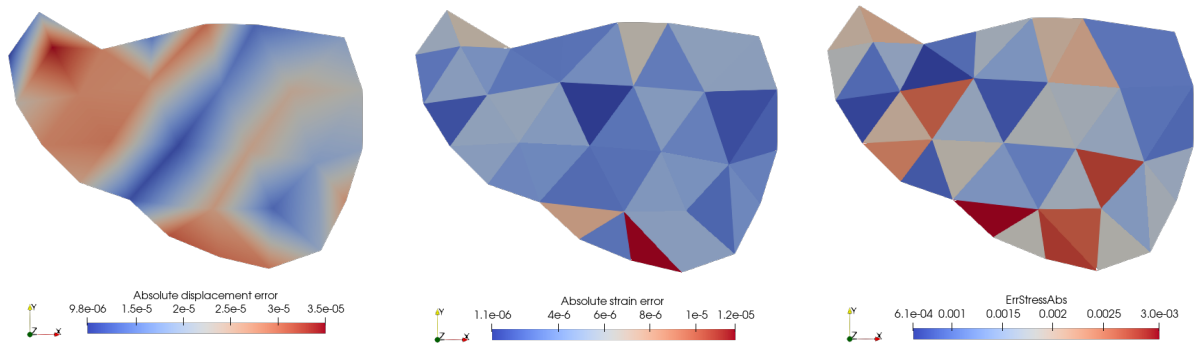


(a) Predicted horizontal displacement (mm), shown on deformed mesh.

(b) Predicted strain magnitude, element-wise.

(c) Predicted stress magnitude, element-wise (Pa).

Figure 4.48: MAgNET predictions on Example B: unstructured $P1$ triangular mesh with traction applied on the top edge in y (vertical).



(a) Absolute displacement error (mm).

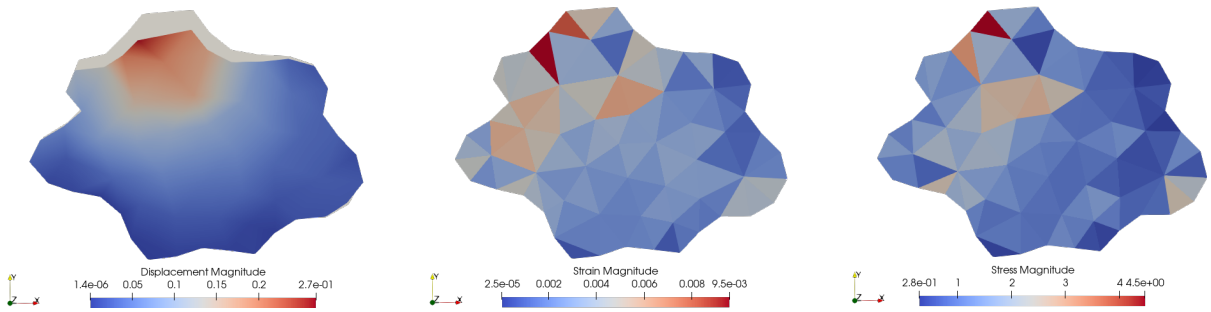
(b) Absolute strain error.

(c) Absolute stress error (Pa).

Figure 4.49: Absolute error fields for Example B relative to the FEM reference.

Next, we test how the two models trained on the irregular shapes behave when the load exceeds the training range. We keep the loading edge and direction fixed and apply a 30 N traction, which lies outside the 8–12 N range used during training. For Mesh D, the predicted fields and the corresponding error maps are shown in Figs. 4.50 and 4.51; for Mesh E, see Figs. 4.52 and 4.53.

On Example A, the error location shifts with load magnitude. For forces within the 8–12 N training range (Figs. 4.46 and 4.47), the largest discrepancies appear on the right-hand arm, where curvature is highest, while the load patch itself shows only moderate error. When the traction is increased to 30 N (Figs. 4.50 and 4.51), a prominent hotspot emerges at and around the top-edge application zone, and the overall error grows. This behavior appears because pushing the load beyond the 8–12 N range amplifies small boundary-layer mismatches near the applied traction, and the effect is more pronounced in strain and stress because they depend on spatial derivatives of the displacement. For this example, the relative L^2 terms, the errors are 15.61% for displacement, 55.41% for strain, and 63.75% for stress.

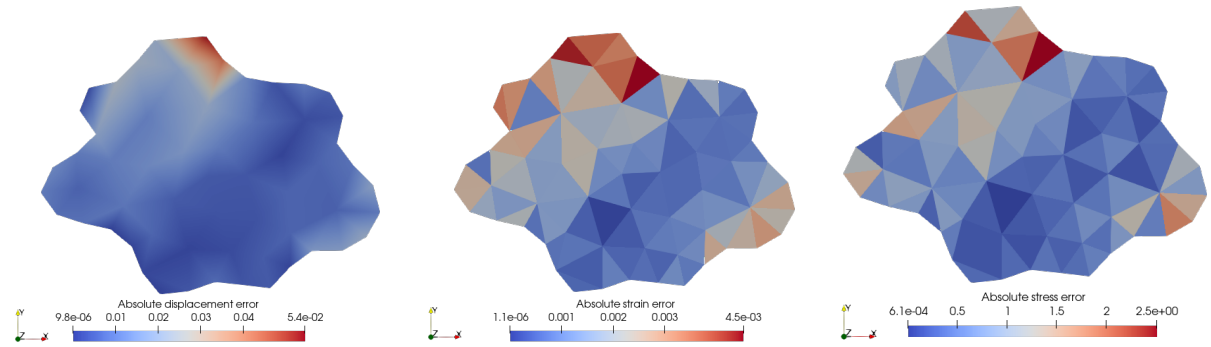


(a) Predicted horizontal displacement (mm) on the deformed mesh.

(b) Predicted strain magnitude, element-wise.

(c) Predicted stress magnitude, element-wise (Pa).

Figure 4.50: MAgNET predictions for Example A under a 30 N top-edge traction (outside the 8–12 N training range).



(a) Absolute displacement error (mm).

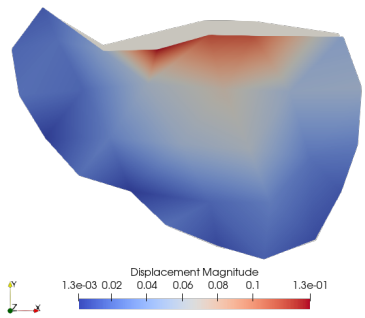
(b) Absolute strain error.

(c) Absolute stress error (Pa).

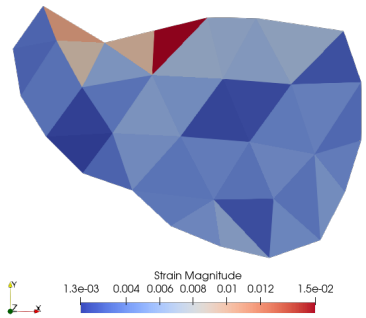
Figure 4.51: Absolute error fields for Example A under a 30 N top-edge traction relative to the FEM reference.

On Example B, the same pattern holds. As seen in Fig. 4.52, displacement is well captured,

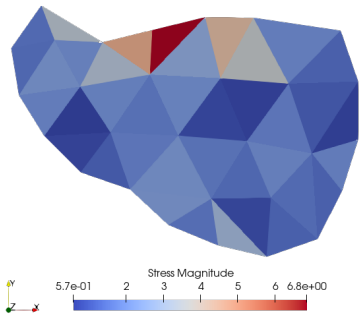
and the error maps in Fig. 4.53 peak along the lower rim and near the bottom-right notch, where the top-edge traction induces the largest bending and shear. The relative L^2 errors are 21.72% for displacement, 80.15% for strain, and 80.04% for stress.



(a) Predicted horizontal displacement (mm), shown on deformed mesh.

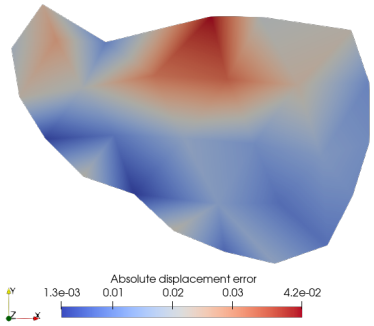


(b) Predicted strain magnitude, element-wise.

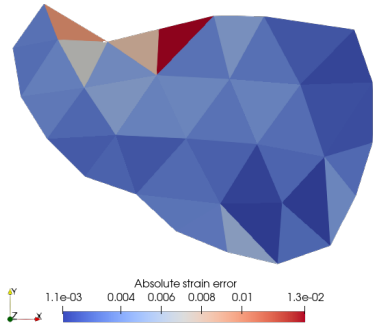


(c) Predicted stress magnitude, element-wise (Pa).

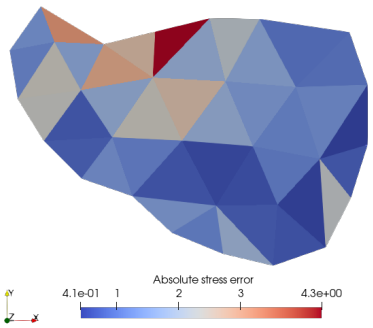
Figure 4.52: MAgNET predictions for Example B under a 30 N top-edge traction (outside the 8–12 N training range).



(a) Absolute displacement error (mm).



(b) Absolute strain error.



(c) Absolute stress error (Pa).

Figure 4.53: Absolute error fields for Example B under a 30 N top-edge traction relative to the FEM reference.

In short, going to 30 N increases the relative errors more for strain and stress than for displacement. This is expected because strain differentiates the displacement field and stress scales with strain, so small differences get amplified. Even so, errors remain concentrated near high-gradient regions, and the overall deformation pattern is preserved.

Across the rectangular benchmarks MAgNET keeps the computation on the native mesh and preserves geometric detail around boundaries. Displacement stays accurate and errors remain localized near the traction patch and along narrow bands of steep variation. On Mesh B the global relative L^2 errors are about 0.31% for displacement and around 6.8% and 7.9% for strain and stress. On Mesh C they are about 0.42% for displacement and around 7.9% and 9.3% for strain and stress. The grid model keeps a small edge on these rectangles especially for derivative fields. This is consistent with graph pooling and enlarged aggregation neighborhoods that can

smooth sharp boundary layers while the grid aligns with the principal gradient direction and retains a slightly sharper imprint of the load.

On unstructured shapes the mesh native approach shows strong results with small and tightly localized residuals. Example A reaches about 0.26% in displacement and about 1.15% and 1.21% in strain and stress. Example B reaches about 0.09% in displacement and about 0.28% and 0.26% in the derivative fields. These values confirm that operating directly on the mesh avoids grid discretization effects and keeps the topology and the boundary description intact.

Extrapolating the load to 30 N increases errors more in strain and stress than in displacement. On the matched rectangle with corner clamps displacement remains close to one or two percent while strain and stress can reach tens of percent. The spatial pattern does not change and the largest discrepancies stay near the load and in boundary layers. This matches the amplification introduced by differentiation of the displacement and by the constitutive mapping.

Runtime follows the same trade off. The grid model trains faster and gives shorter inference while MAgNET remains fast and practical and removes pre and post processing between mesh and grid. Taken together these results show when each surrogate is preferable. Use the grid model when a regular lattice is natural and aligned with the dominant gradients. Use the mesh native model when geometry arrives as an unstructured mesh or when preserving boundary detail and native connectivity matters.

Chapter 5

Conclusion

5.1 Conclusions and future works

This work assembled an image-to-solution pipeline that extracts geometry from segmented images, generates structured and unstructured meshes, and replaces repeated finite-element solves with learned surrogates. Two families were examined. A U-Net on a Cartesian grid offered simple data handling and fast inference on rectangles. A mesh-native graph U-Net (MAgNET) operated directly on unstructured triangulations and removed the need for rasterization.

Across matched rectangular cases the CNN trained faster, produced predictions more quickly, and in our runs achieved lower errors for displacement while keeping strain and stress within single-digit percentages in the nominal range. On meshes obtained from images the graph model preserved geometric detail at the traction patch and kept all computations on native nodes and elements, avoiding interpolation steps. Timing results showed per-sample speed-ups over FEM at inference for both surrogates, with the CNN around 5× and MAgNET around 3.3× on our hardware, whereas FEM remained competitive for a single query with no training cost. Break-even counts indicate where the training expense of the learned models is amortized under repeated use.

The study shows that data driven surrogates can approximate linear elastic responses from images with meaningful speed advantages at inference time. U-Net performs competitively on regular domains and in regions where the field varies smoothly, whereas a mesh native network such as MAgNET preserves geometric detail on unstructured meshes and sustains accuracy near sharp features and high gradients. The finite element reference anchors the evaluation and enables rigorous measurement of discrepancies.

The guiding questions receive coherent answers. A grid based model can match a mesh native model while the geometry remains close to rectangular layouts and the gradients are moderate,

but accuracy degrades near complex boundaries and stress concentrations where the mesh based approach retains quality. The balance between the two depends on mesh density and on load amplitude, with the mesh native model showing advantages under sparser or non uniform meshes and in out of distribution load levels, and the grid based model excelling when the geometry aligns with the sampling grid and variations are smooth. Inference is markedly faster than a fresh FEM solve on a per case basis, and errors remain within acceptable ranges for the test cases considered, both in relative L^2 and in energy terms.

These findings support the use of learned surrogates as companions to finite elements in design and screening workflows that require rapid iteration. They also clarify the conditions under which each family of models is preferable, providing practical guidance for future deployments.

Looking ahead, the priority is stronger generalization so a single surrogate can handle new geometries, resolutions, and load cases without retraining. A first step is mesh-agnostic modeling that operates on graphs of varying size and topology while conditioning on boundary data, material parameters, and simple local geometric descriptors, with validation on held-out meshes and refinements. Operator-learning approaches that learn the force-to-displacement map as a discretization-independent operator offer a practical route to transfer across families of meshes. Equivariance and normalization that factor out rigid motions, global scale, and coordinate reparameterization can further reduce dependence on a particular mesh layout. On the data side, broader training distributions with systematic sampling of shapes, boundary locations, and traction magnitudes, together with explicit out-of-distribution tests for larger loads, should make cross-mesh behavior more reliable. Architecturally, multi-resolution and adaptive graph schemes with shared weights across levels would enable inference at arbitrary resolution and reduce the need to retrain after refinement. A small suite of cross-mesh benchmarks that report accuracy under refinement, sensitivity to mesh quality, and stability on thin features would clarify limits and guide model choices. Finally, automating the path from segmentation to mesh to surrogate with consistent boundary encoding and cached factorizations would support a train-once, apply-anywhere workflow, with a natural extension to three dimensions when the same generalization principles hold on tetrahedral meshes.

Bibliography

[1] Hibbeler, R. C. (2018). *Mechanics of materials* (10th ed.). Pearson Education.

[2] Embry–Riddle Aeronautical University. (2025). *Advanced computational methods* [E-book]. Embry–Riddle Aeronautical University.

[3] Ansys. (n.d.). What is finite element analysis (FEA)? Retrieved January 10, 2025, from `https://www.ansys.com/simulation-topics/what-is-finite-element-analysis`

[4] Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378, 686–707. `https://doi.org/10.1016/j.jcp.2018.10.045`

[5] Li, Z., Kovachki, N. B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A. M., & Anandkumar, A. (2020). Fourier neural operator for parametric partial differential equations. *arXiv preprint* arXiv:2010.08895. `https://arxiv.org/abs/2010.08895`

[6] Li, Z., Zheng, H., Kovachki, N. B., Jin, D., Chen, H., Liu, B., Azizzadenesheli, K., & Anandkumar, A. (2021). Physics-informed neural operator for learning partial differential equations. *arXiv preprint* arXiv:2111.03794. `https://arxiv.org/abs/2111.03794`

[7] Deshpande, S., Sosa, R. I., Bordas, S. P. A., & Lengiewicz, J. (2023). Convolution, aggregation and attention based deep neural networks for accelerating simulations in mechanics. *Frontiers in Materials*, 10, 1128954. `https://doi.org/10.3389/fmats.2023.1128954`

[8] Capuano, G., & Rimoli, J. J. (2019). Smart finite elements: A novel machine learning application. *Computer Methods in Applied Mechanics and Engineering*, 345, 363–381. `https://doi.org/10.1016/j.cma.2018.10.046`

[9] Hastie, T., Tibshirani, R., & Friedman, J. H. (2021). Deep learning. In *The elements of statistical learning with applications in R* (pp. 403–461). Springer.

[10] Johnson, D. (2000). *Advanced structural mechanics: An introduction to continuum mechanics and structural mechanics*. Thomas Telford Publishing.

[11] COMSOL. (n.d.). Mesh refinement. Retrieved January 2025, from `https://www.comsol.com/multiphysics/mesh-refinement`

[12] Zienkiewicz, O. C., Taylor, R. L., & Zhu, J. Z. (2013). *The finite element method: Its basis and fundamentals* (7th ed.). Elsevier.

[13] Hughes, T. J. R. (2000). *The finite element method: Linear static and dynamic finite element analysis*. Dover Publications.

[14] Ciarlet, P. G. (2002). *The finite element method for elliptic problems*. SIAM.

[15] Adams, R. A., & Fournier, J. J. F. (2003). *Sobolev spaces* (2nd ed.). Academic Press.

[16] Brezis, H. (2010). *Functional analysis, Sobolev spaces and partial differential equations*. Springer.

[17] Roth, D., Cheng, C., & Cervantes, C. (2016, October). Neural networks [Lecture notes]. CS 446 Machine Learning, University of Pennsylvania. Retrieved July 2025, from `https://www.cis.upenn.edu/~danroth/Teaching/CS446-17/LectureNotesNew/neuralnet1/main.pdf`

[18] Hastie, T., Tibshirani, R., & Friedman, J. H. (2009). Neural networks. In *The elements of statistical learning: Data mining, inference, and prediction* (pp. 389–416). Springer.

[19] Stuckey, P. J., Sulzmann, M., & Wazny, M. (2009). *A foundation for argumentation semantics in logic programming*. University of Melbourne. Retrieved August 2025, from `https://people.eng.unimelb.edu.au/pstuckey/papers/argmax.pdf`

[20] Wang, Z., Li, T., & Ermon, S. (2023). On the differentiability of the argmax operation in deep learning. *Science Advances*, 9(46), eado8999. `https://doi.org/10.1126/sciadv.ado8999`

[21] Aggarwal, C. C. (2020). *Linear algebra and optimization for machine learning: A textbook*. Springer Nature Switzerland. `https://doi.org/10.1007/978-3-030-40344-7`

[22] Prince, S. J. D. (2023). *Understanding deep learning*. The MIT Press.

[23] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems* (Vol. 25).

[24] O'Shea, K., & Nash, R. (2015). An introduction to convolutional neural networks. *arXiv preprint* arXiv:1511.08458. https://arxiv.org/abs/1511.08458

[25] Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional networks for biomedical image segmentation. *arXiv preprint* arXiv:1505.04597. http://arxiv.org/abs/1505.04597

[26] Marin, E., Pressacco, M., Fusi, S., Lanzutti, A., Turchet, S., & Fedrizzi, L. (2014). *Materials Science and Engineering C – 2013 (Grade 2) – IF2689* [Dataset].

[27] Graça, R. J. R., Rodrigues, J. A., Loja, M. A. R., & Jorge, P. M. (2017). Multiscale stress analysis in CFRC using microscope image data of carbon fibres. *Composite Structures*, 176, 471–480. https://doi.org/10.1016/j.compstruct.2017.05.020

[28] Warren, P. (2023, May). *Artificial grains and real grains* [Dataset]. Retrieved August 25, 2025, from https://www.kaggle.com/datasets/peterwarren/voronoi-artificial-grains-gen

[29] Bertoldo, J. P. C., Decencière, E., Ryckelynck, D., & Proudhon, H. (2021). A modular U-Net for automated segmentation of X-ray tomography images in composite materials. *Frontiers in Materials*, 8, 761229. https://doi.org/10.3389/fmats.2021.761229

[30] Deshpande, S., Sosa, R. I., Bordas, S. P. A., & Lengiewicz, J. (2023). Convolution, aggregation and attention based deep neural networks for accelerating simulations in mechanics. *Frontiers in Materials*, 10, 1128954. https://doi.org/10.3389/fmats.2023.1128954

[31] Rodrigues, J. A., & Vieira, B. (2024, October). A mathematical analysis of image mesh generation using Delaunay triangulation and image processing techniques. In *Proceedings of CEMEERS'24b*.

[32] Deshpande, S., Sosa, R., Bordas, S., & Lengiewicz, J. (2023). Convolution, aggregation and attention based deep neural networks for accelerating simulations in mechanics. *Frontiers in Materials*, 10, 1128954. https://doi.org/10.3389/fmats.2023.1128954

[33] Deshpande, S., Bordas, S. P. A., & Lengiewicz, J. (2024). MAgNET: A graph U-Net architecture for mesh-based simulations. *Engineering Applications of Artificial Intelligence*, 133, 108055. https://doi.org/10.1016/j.engappai.2024.108055

[34] Gao, H., & Ji, S. (2022). Graph U-Nets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9), 4948–4960. https://doi.org/10.1109/TPAMI.2021.3081010

[35] Logg, A., Mardal, K.-A., & Wells, G. N. (2012). *Automated solution of differential equations by the finite element method: The FEniCS book*. Springer.

[36] Bradski, G. (2000). The OpenCV library. *Dr. Dobb's Journal of Software Tools*.

[37] Harris, C. R., Millman, K. J., van der Walt, S. J., et al. (2020). Array programming with NumPy. *Nature*, 585(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[38] Reback, J., McKinney, W., jbrockmendel, Van den Bossche, J., Augspurger, T., Hawkins, S., et al. (2020). *pandas-dev/pandas: Pandas* [Software]. Zenodo. https://doi.org/10.5281/zenodo.3509134

[39] Pedregosa, F., Varoquaux, G., Gramfort, A., et al. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.

[40] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

[41] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., ... Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems* [Software]. Retrieved August 25, 2025, from https://www.tensorflow.org/

[42] Chollet, F., et al. (2015). *Keras* [Software]. Retrieved August 25, 2025, from https://keras.io

[43] Vieira, B., Rodrigues, J., Deshpande, S., & Bordas, S. (2025, April 6). *Simulating stresses and strains in solid mechanics directly from images using convolutional neural networks* [Conference paper]. Zenodo. Retrieved November 16, 2025, from https://zenodo.org/records/15164468

[44] Colomés, D., Rocha, K., Van der Meer, F., & Lesueur, D. (2023). *3.1 Strong form for linear elasticity*. In *Continuum mechanics* (CiTG Jupyter Book). Delft University of Technology. Retrieved November 16, 2025, from https://teachbooks.tudelft.nl/computational-modelling/continuum_linear/continuum_mechanics.html

Appendix A

Computational Implementation

This appendix consolidates the practical details needed to reproduce the results in Chapter 4. It follows the same pipeline: image segmentation (Section 4.1), image-based mesh generation (Section 4.2), and learned surrogates for elastic response on grids (Section 4.3) and on meshes (MAgNET).

A.1 Environment and Reproducibility

All experiments were run on a personal Apple MacBook Air with 8,GB of unified memory using the CPU only, with development in Visual Studio Code. To respect the hardware limits, datasets were streamed or cached to disk when helpful, batch sizes were kept modest, and intermediate results were checkpointed. Exact package versions are pinned in environment files, and random seeds are fixed consistently across Python, NumPy, and TensorFlow/Keras to support reproducibility. The simulation stack relies on FEniCS/dolfin for finite elements [35], while OpenCV handles image I/O and preprocessing [36]. Data handling uses NumPy, Pandas, and scikit-learn [37, 38, 39]; figures are produced with Matplotlib [40]; and learning models are implemented in TensorFlow/Keras [41, 42]. For graph-based surrogates, the architecture follows the MAgNET design [32, 33].

A.2 Datasets and Versioning

The segmentation datasets mirror Section 4.1 and include stainless-steel grain boundaries and artificial grains with provided masks [28], polyamide-66 x-ray CT images with masks [29], and a combined subset drawing from both. For the mechanics experiments, force–displacement datasets are generated via finite-element solves. All resulting CSV files—features, labels, and, when applicable, node coordinates—preserve a fixed vertex ordering and are split into training and

test sets using a consistent random seed.

A.3 Metrics and Evaluation Protocol

For each test case we report:

- **Nodal displacement error** at node i:

$$e_u(i) = \left\|\hat{\mathbf{u}}_i - \mathbf{u}_i^{\mathrm{FEM}}\right\|_2.$$

- **Element-wise strain and stress errors:** fields are computed per element and averaged to nodes for visualization. Let $\mathcal{T}(i)$ be elements incident to node i,

$$e_\varepsilon(i) = \frac{1}{|\mathcal{T}(i)|}\sum_{e\in\mathcal{T}(i)} \left\|\hat{\boldsymbol{\varepsilon}}_e - \boldsymbol{\varepsilon}_e\right\|_2, \quad e_\sigma(i) = \frac{1}{|\mathcal{T}(i)|}\sum_{e\in\mathcal{T}(i)} \left\|\hat{\boldsymbol{\sigma}}_e - \boldsymbol{\sigma}_e\right\|_2.$$

- **Global relative** L^2 for a quantity $q \in \{\mathbf{u}, \boldsymbol{\varepsilon}, \boldsymbol{\sigma}\}$:

$$\mathcal{E}_{\mathrm{rel}}(q) = \frac{\left\|\hat{q} - q\right\|_2}{\left\|q\right\|_2} \times 100\%.$$

All definitions match Chapter 4 to allow like-for-like comparisons between CNN and MAgNET.

A.4 Timing and Practical Considerations

On the CPU-only setup, the compact CNN trains faster and infers slightly faster than MAgNET on the matched rectangles, while both are substantially faster than running a fresh FEM solve at inference time. Absolute times depend on hardware and implementation, but the relative trends reported in Chapter 4 hold across runs. For workloads with many queries, the one-time training cost is amortized quickly; for occasional queries, direct FEM may remain competitive.

A.5 Image Segmentation: Models and Training

We use three binary U-Nets to isolate regions of interest, matching Chapter 4:

- **Model I** (dataset 1): encoder channels $64 \to 128 \to 256$ with mirrored decoder and skip connections; input size 256×256; sigmoid head and binary cross-entropy. Batch 32, 15 epochs.
- **Model II** (dataset 2): same architecture as Model I, trained with batch 2 for 2 epochs due to the dataset size.

- **Model III** (combined dataset): one extra level ($128 \to 256 \to 512 \to 1024$), same decoder pattern and sigmoid head. Batch 2, 10 epochs.

Preprocessing comprised resizing all images to a fixed resolution, normalizing intensities, and converting the masks to single-channel binary arrays. The same light augmentation was applied across all models—small rotations and flips, modest translations and zoom, and mild adjustments to brightness and contrast—to mitigate overfitting while preserving the semantics of edges and regions. The resulting learning curves and qualitative segmentations align with the results reported in Chapter 4, ensuring that differences arise from model architecture and data regime rather than preprocessing.

The snippets below are the exact implementations used for Models I–III in Section 4.1. Paths are placeholders and should be adapted to your local dataset layout.

```
# Parameters (shared)
IMG_H, IMG_W = 256, 256

# --- Paths (edit to match your data) ---
image_dir_A = "/path/to/datasetA/images"
mask_dir_A  = "/path/to/datasetA/masks"
image_dir_B = "/path/to/datasetB/images"
mask_dir_B  = "/path/to/datasetB/masks"

# --- Minimal I/O and preprocessing ---
import os, cv2, numpy as np
def load_images(folder, img_w=IMG_W, img_h=IMG_H):
    xs = []
    for fn in sorted(os.listdir(folder)):
        if fn.lower().endswith((".png",".jpg",".jpeg")):
            im = cv2.imread(os.path.join(folder, fn),
    ↪ cv2.IMREAD_COLOR)
            xs.append(cv2.resize(im, (img_w, img_h)))
    return np.asarray(xs, dtype=np.float32) / 255.0

def load_binary_masks(folder, img_w=IMG_W, img_h=IMG_H, thresh=127):
    ms = []
    for fn in sorted(os.listdir(folder)):
        if fn.lower().endswith((".png",".jpg",".jpeg")):
            m = cv2.imread(os.path.join(folder, fn),
    ↪ cv2.IMREAD_GRAYSCALE)
```

```
            m = cv2.resize(m, (img_w, img_h),
    ↪ interpolation=cv2.INTER_NEAREST)
            _, m = cv2.threshold(m, thresh, 255, cv2.THRESH_BINARY)
            ms.append(m[..., None])
    return (np.asarray(ms, dtype=np.uint8) > 0).astype(np.float32)

# --- Light augmentation (shared) ---
from keras.preprocessing.image import ImageDataGenerator
augment = ImageDataGenerator(
    rotation_range=15, width_shift_range=0.12,
    ↪ height_shift_range=0.12,
    shear_range=10, zoom_range=0.6, horizontal_flip=True,
    ↪ vertical_flip=True,
    channel_shift_range=0.10, brightness_range=[0.8, 1.2]
)
```

Listing A.1: Common setup: parameters, I/O, preprocessing, and data augmentation

Listing A.1 collects the shared components. Images and masks are resized to a fixed 256×256 canvas and normalized to keep training well conditioned on a CPU-only machine. Masks are converted to single-channel binaries through thresholding and stored with an explicit channel dimension, which matches the one-channel sigmoid head used by all models. The augmentation policy is deliberately light so that small rotations and flips, modest shifts and zoom, and mild brightness changes reduce overfitting while preserving edges and thin regions.

```
from keras import Input, Model
from keras.layers import Conv2D, MaxPooling2D, Conv2DTranspose,
    ↪ concatenate
from keras.optimizers import Adam
from sklearn.model_selection import train_test_split

def unet_model_1(img_h=IMG_H, img_w=IMG_W, in_ch=3):
    x = inputs = Input((img_h, img_w, in_ch))
    # Down
    c1 = Conv2D(64, 3, activation='relu', padding='same')(x)
    c1 = Conv2D(64, 3, activation='relu', padding='same')(c1); p1 =
    ↪ MaxPooling2D(2)(c1)
    c2 = Conv2D(128, 3, activation='relu', padding='same')(p1)
    c2 = Conv2D(128, 3, activation='relu', padding='same')(c2); p2 =
    ↪ MaxPooling2D(2)(c2)
```

```
    c3 = Conv2D(256, 3, activation='relu', padding='same')(p2)
    c3 = Conv2D(256, 3, activation='relu', padding='same')(c3)
    # Up
    u2 = Conv2DTranspose(128, 2, strides=2, padding='same')(c3)
    u2 = concatenate([u2, c2])
    u2 = Conv2D(128, 3, activation='relu', padding='same')(u2)
    u2 = Conv2D(128, 3, activation='relu', padding='same')(u2)
    u1 = Conv2DTranspose(64, 2, strides=2, padding='same')(u2)
    u1 = concatenate([u1, c1])
    u1 = Conv2D(64, 3, activation='relu', padding='same')(u1)
    u1 = Conv2D(64, 3, activation='relu', padding='same')(u1)
    out = Conv2D(1, 1, activation='sigmoid')(u1)
    m = Model(inputs, out)
    m.compile(optimizer=Adam(2e-4), loss='binary_crossentropy',
    ↪ metrics=['accuracy'])
    return m

# Data (dataset A) and train
X_A = load_images(image_dir_A);  Y_A = load_binary_masks(mask_dir_A)
Xtr, Xte, Ytr, Yte = train_test_split(X_A, Y_A, test_size=0.10,
   ↪ random_state=42, shuffle=True)
BATCH, EPOCHS = 32, 15
m1 = unet_model_1()
hist1 = m1.fit(
    augment.flow(Xtr, Ytr, batch_size=BATCH),
    steps_per_epoch=max(1, len(Xtr)//BATCH),
    validation_data=(Xte, Yte),
    epochs=EPOCHS,
    verbose=1
)
```

Listing A.2: Model I: compact U-Net (64→128→256) and training on dataset A

Model I in Listing A.2 is a compact U-Net that climbs from 64 to 256 channels and mirrors back with skip connections. The depth balances capacity with memory limits and keeps training stable on CPU. The single sigmoid head with binary cross-entropy predicts foreground probability per pixel. Training uses batch size 32 for 15 epochs with a standard Adam optimizer and the same split reported in Chapter 4, so the accuracy curves and qualitative masks can be reproduced exactly.

```
# Minimal deltas: new paths, tiny batch/epochs
X_B = load_images(image_dir_B);  Y_B = load_binary_masks(mask_dir_B)
from sklearn.model_selection import train_test_split
Xtr, Xte, Ytr, Yte = train_test_split(X_B, Y_B, test_size=0.10,
    ↪ random_state=42, shuffle=True)

BATCH, EPOCHS = 2, 2
m2 = unet_model_1()
hist2 = m2.fit(
    augment.flow(Xtr, Ytr, batch_size=BATCH),
    steps_per_epoch=max(1, len(Xtr)//BATCH),
    validation_data=(Xte, Yte),
    epochs=EPOCHS,
    verbose=1
)
```

Listing A.3: Model II: same network, limited-data regime on dataset B

Model II in Listing A.3 keeps the architecture and preprocessing fixed and changes only the data regime. Batches of size 2 and two epochs reflect the scarcity of annotated x-ray images. Holding the network constant isolates the effect of sample count and schedule on convergence and validation behaviour. The aim is to illustrate sensitivity to data rather than to draw broad conclusions about segmentation accuracy.

```
from keras import Input, Model
from keras.layers import Conv2D, MaxPooling2D, Conv2DTranspose,
    ↪ concatenate
from keras.optimizers import Adam
from sklearn.model_selection import train_test_split
import numpy as np, cv2

def unet_model_3(img_h=IMG_H, img_w=IMG_W, in_ch=3):
    x = inputs = Input((img_h, img_w, in_ch))
    # Down
    c1 = Conv2D(128, 3, activation='relu', padding='same')(x)
    c1 = Conv2D(128, 3, activation='relu', padding='same')(c1); p1 =
    ↪ MaxPooling2D(2)(c1)
    c2 = Conv2D(256, 3, activation='relu', padding='same')(p1)
    c2 = Conv2D(256, 3, activation='relu', padding='same')(c2); p2 =
    ↪ MaxPooling2D(2)(c2)
```

```
    c3 = Conv2D(512, 3, activation='relu', padding='same')(p2)
    c3 = Conv2D(512, 3, activation='relu', padding='same')(c3); p3 =
    ↪ MaxPooling2D(2)(c3)
    bn = Conv2D(1024, 3, activation='relu', padding='same')(p3)
    bn = Conv2D(1024, 3, activation='relu', padding='same')(bn)
    # Up
    u3 = Conv2DTranspose(512, 2, strides=2, padding='same')(bn)
    u3 = concatenate([u3, c3]); u3 = Conv2D(512, 3,
    ↪ activation='relu', padding='same')(u3)
    u3 = Conv2D(512, 3, activation='relu', padding='same')(u3)
    u2 = Conv2DTranspose(256, 2, strides=2, padding='same')(u3)
    u2 = concatenate([u2, c2]); u2 = Conv2D(256, 3,
    ↪ activation='relu', padding='same')(u2)
    u2 = Conv2D(256, 3, activation='relu', padding='same')(u2)
    u1 = Conv2DTranspose(128, 2, strides=2, padding='same')(u2)
    u1 = concatenate([u1, c1]); u1 = Conv2D(128, 3,
    ↪ activation='relu', padding='same')(u1)
    u1 = Conv2D(128, 3, activation='relu', padding='same')(u1)
    out = Conv2D(1, 1, activation='sigmoid')(u1)
    m = Model(inputs, out)
    m.compile(optimizer=Adam(2e-4), loss='binary_crossentropy',
    ↪ metrics=['accuracy'])
    return m

# Combine subsets from datasets A and B (example)
X_c = np.concatenate([X_A[:101], X_B[:100]], axis=0)
Y_c = np.concatenate([Y_A[:101], Y_B[:100]], axis=0)
Xtr, Xte, Ytr, Yte = train_test_split(X_c, Y_c, test_size=0.10,
    ↪ random_state=42, shuffle=True)

BATCH, EPOCHS = 2, 10
m3 = unet_model_3()
hist3 = m3.fit(
    augment.flow(Xtr, Ytr, batch_size=BATCH),
    steps_per_epoch=max(1, len(Xtr)//BATCH),
    validation_data=(Xte, Yte),
    epochs=EPOCHS,
    verbose=1
)
```

```
47
48 # Post-processing: threshold and color connected components (for
       ↪ visualization)
49 prob = m3.predict(Xte[:1], verbose=0)[0, ..., 0]
50 mask = (prob > 0.5).astype(np.uint8) * 255
51 num, labels = cv2.connectedComponents(mask)
52 palette = (np.random.rand(num, 3) * 255).astype(np.uint8); palette[0]
       ↪ = 0
53 colored = palette[labels]  # H x W x 3
```

Listing A.4: Model III: deeper U-Net (128→256→512→1024) and post-processing

Model III in Listing A.4 adds one encoder–decoder stage, reaching 1024 channels at the bottleneck. The extra depth enlarges the receptive field and often produces cleaner delineations on the combined dataset. Training remains conservative with batch size 2 and ten epochs to limit overfitting under CPU-only constraints. After inference, probabilities are thresholded and connected components are coloured for the figures reported in the chapter.

Across the three models, random seeds are fixed and the same augmentation policy is applied so that runs are comparable. The implementation favours clarity and determinism over squeezing out the last percentage points of accuracy, which keeps the segmentation results aligned with the mechanics surrogates and the reproducibility goals of this thesis.

A.6 Image-Based Mesh Generation

Consistent with Section 4.2, binary masks are converted to analysis-ready 2D meshes as follows:

- **Binarization and boundary extraction:** grayscale thresholding produces a binary image; the largest external contour is taken as the boundary (OpenCV).
- **Boundary sampling:** points are distributed at near-uniform arc length along the extracted contour to control perimeter resolution.
- **Interior sampling:** Poisson-disk sampling places interior points with a minimum spacing; a small safety margin is enforced from the boundary samples.
- **Triangulation and filtering:** Delaunay triangulation is built on the union of boundary and interior points; triangles whose centroids fall outside the polygon are removed.
- **Smoothing:** Laplacian smoothing moves interior nodes to the average of neighbors while keeping boundary nodes fixed. A few iterations improve element quality without eroding sharp features.

This procedure produces unstructured P1 triangular meshes that accurately follow the segmented geometry. For structured toy examples we also use regular Cartesian grids and split quads into triangles when needed.

To make these steps reproducible, the exact implementation is provided below (following the outline in [31]), with brief explanations interleaved so each choice can be audited.

```python
import cv2, numpy as np

def read_image(image_path, thresh=240):
    image = cv2.imread(image_path)
    if image is None:
        raise ValueError(f"Could not read '{image_path}'")
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    _, binary = cv2.threshold(gray, thresh, 255, cv2.THRESH_BINARY)
    return image, binary

def largest_contour(binary):
    cnts, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL,
    ↪ cv2.CHAIN_APPROX_SIMPLE)
    if not cnts:
        raise ValueError("No external contour found.")
    contour = max(cnts, key=cv2.contourArea).reshape(-1,
    ↪ 2).astype(float)
    return contour
```

Listing A.5: Read image, create a binary mask, and extract the largest external contour

Listing A.5 reads the image, converts it to grayscale, thresholds to obtain a binary mask, and extracts the largest external contour as a polygonal boundary. The threshold should yield a solid foreground and a clean background. When illumination varies, an adaptive method can be substituted without changing later steps.

```python
import numpy as np

def distribute_uniform_boundary_points(boundary, n_points):
    b = np.vstack([boundary, boundary[0]])      # close the loop
    seg = np.linalg.norm(np.diff(b, axis=0), axis=1)
    perim = seg.sum()
    spacing = perim / n_points

    pts, cur = [], 0.0
```

```
    i, start = 0, b[0]
    while len(pts) < n_points and i < len(b) - 1:
        end = b[i+1]
        seglen = np.linalg.norm(end - start)
        while cur + seglen >= spacing and len(pts) < n_points:
            t = (spacing - cur) / seglen
            newp = start + t * (end - start)
            pts.append(newp)
            cur = 0.0
            start = newp
            seglen = np.linalg.norm(end - start)
        cur += seglen
        i += 1
        start = end
    return np.array(pts, dtype=float)
```

Listing A.6: Evenly spaced samples along the detected boundary

Listing A.6 places samples with approximately uniform arc-length spacing along the boundary. Specifying the number of points yields predictable perimeter resolution. These samples remain fixed to preserve geometric fidelity at the edges.

```
from shapely.geometry import Polygon, Point
from scipy.spatial.distance import cdist
import numpy as np

def poisson_disk_sampling(boundary, boundary_points,
                          min_dist, num_samples,
    ↪ min_dist_to_boundary):
    poly = Polygon(boundary)
    x0, y0, x1, y1 = poly.bounds
    pts = []
    while len(pts) < num_samples:
        x = np.random.uniform(x0, x1)
        y = np.random.uniform(y0, y1)
        if not poly.contains(Point(x, y)):
            continue
        cand = np.array([[x, y]])
        if pts and float(cdist(cand, np.asarray(pts)).min()) <=
    ↪ float(min_dist):
```

```
            continue
        if float(cdist(cand, boundary_points).min()) <=
    ↪ float(min_dist_to_boundary):
            continue
        pts.append([x, y])
    return np.asarray(pts, dtype=float)
```

Listing A.7: Poisson–disk sampling of interior points with a safety margin to the boundary

Listing A.7 seeds the interior with a Poisson–disk pattern. A minimum separation controls element size, and a small exclusion band from the boundary prevents sliver triangles and preserves the contour. For larger meshes, faster blue-noise samplers can replace this routine without altering downstream interfaces.

```
from scipy.spatial import Delaunay
from shapely.geometry import Polygon, Point

def triangulate_inside(points, boundary):
    poly = Polygon(boundary)
    tri = Delaunay(points)
    keep = []
    for simplex in tri.simplices:
        centroid = points[simplex].mean(axis=0)
        if poly.contains(Point(float(centroid[0]),
    ↪ float(centroid[1]))):
            keep.append(simplex)
    return tri, np.array(keep, dtype=int)
```

Listing A.8: Delaunay triangulation of all points and filtering to triangles whose centroids lie inside the shape

Listing A.8 builds a Delaunay triangulation over all samples and retains only those triangles whose centroids fall inside the boundary polygon. This centroid test is robust for simply connected shapes; if holes are present, the same approach applies with a polygon that encodes holes explicitly.

```
def find_neighbors(n_points, triangles):
    adj = [set() for _ in range(n_points)]
    for a, b, c in triangles:
        adj[a].update([b, c]); adj[b].update([a, c]);
    ↪ adj[c].update([a, b])
    return [sorted(list(s)) for s in adj]

```

```
def laplacian_smoothing(points, triangles, boundary_points, iters=10):
    boundary_set = {tuple(p) for p in np.asarray(boundary_points)}
    neigh = find_neighbors(points.shape[0], triangles)
    P = points.copy()
    for _ in range(iters):
        Q = P.copy()
        for i, nb in enumerate(neigh):
            if tuple(P[i]) in boundary_set or not nb:
                continue
            Q[i] = P[nb].mean(axis=0)
        P = Q
    return P
```

Listing A.9: Laplacian smoothing with boundary nodes fixed

Listing A.9 performs a few iterations of Laplacian smoothing to regularise element shapes while keeping the boundary fixed. Interior nodes move to the average of their neighbours, which improves minimum angles without eroding sharp features.

```
def mesh_from_image(image_path, n_boundary=200, n_interior=500,
                    min_dist=8.0, min_dist_to_boundary=6.0,
                    thresh=240, smooth_iters=10):
    image, binary = read_image(image_path, thresh=thresh)
    boundary = largest_contour(binary)
    bpts = distribute_uniform_boundary_points(boundary, n_boundary)
    ipts = poisson_disk_sampling(boundary, bpts, min_dist,
    ↪ n_interior, min_dist_to_boundary)
    all_pts = np.vstack([bpts, ipts])
    _, tris_inside = triangulate_inside(all_pts, boundary)
    nodes_smoothed = laplacian_smoothing(all_pts, tris_inside, bpts,
    ↪ iters=smooth_iters)
    return image, boundary, nodes_smoothed, tris_inside

# Example (commented):
# image, boundary, nodes, tris =
    ↪ mesh_from_image("Isolated_Color_img2.png")
# import matplotlib.pyplot as plt
# plt.figure(figsize=(6,6))
# plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
# plt.triplot(nodes[:,0], nodes[:,1], tris, color="tab:blue")
```

```
# plt.scatter(nodes[:,0], nodes[:,1], s=3, c="k")
# plt.axis("off"); plt.tight_layout(); plt.show()
```

Listing A.10: End-to-end driver from image to smoothed interior mesh

The driver in Listing A.10 wires the stages together. The defaults produce meshes with uniform boundary resolution and approximately uniform interior spacing. Increasing the number of boundary samples improves edge fidelity, whereas decreasing the interior separation increases element density. The plotting lines are commented to keep the appendix fast to compile; uncommenting them renders the triangulation overlay used in the figures. The implementation assumes a simply connected foreground. When images contain holes or multiple disjoint objects, the contour extraction can be extended to return a composite polygon; the remainder of the pipeline applies unchanged.

A.7 FEM Dataset Generation

We assemble supervised datasets for linear elasticity in plane stress. Let E and ν denote Young's modulus and Poisson's ratio. With small strains $\varepsilon(u) = \frac{1}{2}(\nabla u + \nabla u^\top)$ and plane-stress constitutive law

$$\sigma(u) = \lambda \operatorname{tr}(\varepsilon(u))\, I + 2\mu\, \varepsilon(u), \quad \mu = \frac{E}{2(1+\nu)}, \quad \lambda = \frac{2\mu\,\nu}{1-\nu},$$

the weak form is $a(u, v) = \int_\Omega \sigma(u) : \varepsilon(v)\, dx$. Dirichlet clamps are applied either at the four corners or along selected edges, as in Chapter 4. Tractions are applied on short, four-node boundary patches in a prescribed direction. For each randomized load sample, FEniCS/dolfin [35] solves for the nodal displacement. Features are the per-node forces (f_x, f_y) in N/m; labels are the corresponding nodal displacements (u_x, u_y) in millimeters. The same fixed vertex order is used for all CSV outputs.

The next two listings give the implementation used to generate supervised datasets for 2D, linear, plane-stress elasticity and to evaluate predictions by exporting error fields to VTU. The generator assembles a single stiffness matrix and reuses it across randomized traction samples, which makes dataset creation deterministic and fast on CPU. Features are per-node forces on the mesh boundary patch, labels are nodal displacements in millimetres in a fixed vertex order that matches all CSV outputs.

```
from dolfin import *
import numpy as np, random
from sklearn.model_selection import train_test_split
```

```
4
5  # --- helpers: vertices, boundary facets, ordering ---
6  def collect_vertices(mesh):
7      verts = sorted(vertices(mesh), key=lambda v: v.index())
8      coords = np.array([[v.point().x(), v.point().y()] for v in
       ↪ verts], dtype=float)
9      return verts, coords
10
11 def guess_top_boundary_marker(mesh, tol=1e-8, value=1):
12     """Mark exterior facets whose midpoint y is near the mesh's max
       ↪ y."""
13     mf = MeshFunction("size_t", mesh, mesh.topology().dim()-1, 0)
14     ymax = mesh.coordinates()[:,1].max()
15     for f in facets(mesh):
16         if f.exterior():
17             mp = f.midpoint()
18             if abs(mp.y() - ymax) <= tol or mp.y() > ymax - 1e-6:
19                 mf.array()[f.index()] = value
20     return mf
21
22 def boundary_vertices_from_marker(mesh, marker, id_val):
23     """Unique vertex indices belonging to facets with
       ↪ marker==id_val."""
24     bset = set()
25     for f in facets(mesh):
26         if marker[f] == id_val:
27             bset.update(f.entities(0))
28     return sorted(list(bset))
29
30 def order_nodes_along_axis(coords, ids):
31     """Order a boundary node set by the first PCA axis (1D
       ↪ parameter)."""
32     pts = coords[ids]
33     c = pts.mean(axis=0)
34     U, S, Vt = np.linalg.svd(pts - c, full_matrices=False)
35     t = (pts - c) @ Vt[0]
36     return [ids[i] for i in np.argsort(t)]
37
38 def center_window(ids_ordered, k):
```

```
    if k >= len(ids_ordered): return ids_ordered
    m = len(ids_ordered) // 2
    a = max(0, m - k//2)
    b = a + k
    return ids_ordered[a:b]

# --- main generator ---
def generate_dataset_2d(
    mesh,
    E_val=1e5, nu_val=0.3,
    fixed_nodes=None,                  # list of vertex indices for
    ↪ point clamps
    fixed_marker=None, fixed_id=None, # boundary marker + id for
    ↪ Dirichlet clamp
    traction_marker=None, traction_id=1,  # marker + id for traction
    ↪ edge
    patch_size=4,                      # number of boundary nodes in
    ↪ the patch
    traction_axis='y',                 # 'x' or 'y'
    T_min=-800.0, T_max=-200.0,        # load range per node (N/m)
    num_samples=10000,
    out_prefix="meshX"                 # file name stem
):
    # FE space and material (P1, plane stress)
    V = VectorFunctionSpace(mesh, "Lagrange", 1)
    u, v = TrialFunction(V), TestFunction(V)
    E, nu = Constant(E_val), Constant(nu_val)
    mu = E / (2.0*(1.0+nu))
    lam = E*nu/((1.0+nu)*(1.0-2.0*nu)); lam = 2.0*mu*lam/(lam+2.0*mu)

    def eps(w): return sym(grad(w))
    def sigma(w): return lam*tr(eps(w))*Identity(2) + 2.0*mu*eps(w)
    a = inner(sigma(u), eps(v)) * dx

    # vertex order and coordinates
    verts, coords = collect_vertices(mesh)
    nverts = len(verts)

    # traction boundary marker (auto-top if none is provided)
```

```
    if traction_marker is None:
        traction_marker = guess_top_boundary_marker(mesh,
↪ value=traction_id)
    ds = Measure("ds", domain=mesh, subdomain_data=traction_marker)

    # boundary vertex candidates and centered patch
    cand = boundary_vertices_from_marker(mesh, traction_marker,
↪ traction_id)
    cand_ordered = order_nodes_along_axis(coords, cand)
    patch = center_window(cand_ordered, patch_size)

    # Dirichlet clamps
    bcs = []
    if fixed_nodes is not None and len(fixed_nodes) > 0:
        class FixedNodes(SubDomain):
            def __init__(self, pts, tol=1e-12):
                self.key = { (round(float(x),12), round(float(y),12))
↪ for x,y in pts }
                self.tol = tol; super().__init__()
            def inside(self, x, on_boundary):
                key = (round(float(x[0]),12), round(float(x[1]),12))
                return key in self.key
        fixed_pts = coords[np.asarray(fixed_nodes, dtype=int)]
        bcs.append(DirichletBC(V, Constant((0.0,0.0)),
↪ FixedNodes(fixed_pts), method="pointwise"))
    if fixed_marker is not None and fixed_id is not None:
        bcs.append(DirichletBC(V, Constant((0.0,0.0)), fixed_marker,
↪ fixed_id))

    # traction field tied to nearest vertex loads
    class SpecificForce(UserExpression):
        def __init__(self, combined, **kwargs):
            self.combined = combined; self.N = len(combined)//2
            super().__init__(**kwargs)
        def eval(self, value, x):
            i = int(np.argmin(np.linalg.norm(coords - (x[0], x[1]),
↪ axis=1)))
            value[0] = float(self.combined[i])            # fx
            value[1] = float(self.combined[i + self.N])   # fy
```

```
        def value_shape(self): return (2,)

    combined = np.zeros(2*nverts)
    f = SpecificForce(combined, degree=0)
    L_form = inner(f, v) * ds(traction_id)

    # assemble once; reuse solver
    A = assemble(a)
    for bc in bcs: bc.apply(A)
    solver = LUSolver(A)
    u_sol = Function(V, name="Displacement")

    # dataset loop
    disp_scale = 1e3  # m -> mm for labels
    features_list, labels_list = [], []
    random.seed(0); np.random.seed(0)

    for _ in range(num_samples):
        # mirror profile on the patch nodes
        t1 = random.uniform(T_min, T_max)
        t2 = random.uniform(T_min, T_max)
        fx = np.zeros(nverts); fy = np.zeros(nverts)
        vals = [t1, t2, t2, t1] if len(patch) >= 4 else
    ↪ [t1]*len(patch)
        for node, vload in zip(patch, vals):
            if traction_axis.lower() == 'x': fx[node] = vload
            else:                            fy[node] = vload

        combined[:nverts] = fx
        combined[nverts:] = fy

        b = assemble(L_form)
        for bc in bcs: bc.apply(b)
        solver.solve(u_sol.vector(), b)

        ux = np.array([u_sol(p)[0] for p in (vv.point() for vv in
    ↪ verts)])  # m
        uy = np.array([u_sol(p)[1] for p in (vv.point() for vv in
    ↪ verts)])  # m
```

```
143
144         features_list.append(np.concatenate([fx, fy]))
    ↪ # N/m
145         labels_list.append(np.concatenate([ux, uy]) * disp_scale)
    ↪ # mm
146
147     X = np.asarray(features_list); Y = np.asarray(labels_list)
148     Xtr, Xte, Ytr, Yte = train_test_split(X, Y, test_size=0.2,
    ↪ random_state=42, shuffle=True)
149
150     np.savetxt(f"features_train_2D_{out_prefix}.csv", Xtr,
    ↪ delimiter=",")
151     np.savetxt(f"features_test_2D_{out_prefix}.csv",  Xte,
    ↪ delimiter=",")
152     np.savetxt(f"labels_train_2D_{out_prefix}.csv",   Ytr,
    ↪ delimiter=",")
153     np.savetxt(f"labels_test_2D_{out_prefix}.csv",    Yte,
    ↪ delimiter=",")
154
155     print(f"[OK] Saved CSVs with prefix '{out_prefix}'.")
156     print(f"#verts={nverts}, #cells={mesh.num_cells()},
    ↪ DoF={V.dim()}")
157     print("Patch node indices:", patch)
158     return dict(patch_nodes=patch, nverts=nverts)
159
160 # --- example (comment): rectangle with side clamps and top traction
    ↪ ---
161 # mesh = RectangleMesh(Point(0.,0.), Point(3.2,0.8), 31, 7)
162 # result = generate_dataset_2d(mesh,
163 #                              traction_axis='y',
164 #                              patch_size=4,
165 #                              out_prefix="meshC")
```

Listing A.11: General dataset generator for 2D plane-stress elasticity

Listing A.11 constructs a P_1 vector space for plane stress, defines the small-strain tensor and Cauchy stress, and assembles the bilinear form once. Boundary conditions can be set either as pointwise clamps on selected vertices or via a boundary marker, which keeps the setup flexible across meshes. If no traction boundary is supplied, a small helper marks the top edge

by detecting exterior facets near the maximum y. Candidate vertices on that edge are ordered along the dominant geometric axis and a centered patch is selected. Random tractions are drawn within a user-defined range and mirrored across the patch to mimic a short, symmetric load. A 'UserExpression' maps the per-vertex loads to a continuous traction over the marked edge. The right-hand side is reassembled for each sample while the factorised stiffness is reused, which makes the loop efficient. Displacements are sampled at the vertex coordinates and converted from metres to millimetres for storage. CSVs are written for train and test splits, and the vertex order is fixed by construction, which guarantees that learning features and labels always align.

```
import os, math, numpy as np, pandas as pd
from dolfin import *

def vertices_and_tris(mesh):
    verts = sorted(vertices(mesh), key=lambda v: v.index())
    pts = np.array([[v.point().x(), v.point().y()] for v in verts],
    ↪ dtype=float)
    tris = []
    for c in cells(mesh):
        v = c.entities(0)
        if len(v) == 3:
            tris.append(tuple(map(int, v)))
        else:
            # for quads/polygons: fan triangulation about the first
    ↪ vertex
            for k in range(1, len(v)-1):
                tris.append((int(v[0]), int(v[k]), int(v[k+1])))
    return pts, np.array(tris, dtype=int)

def compute_strain_CST(disp_m, pts, tri_nodes):
    """CST strain [exx, eyy, gxy] with gxy = 2*exy (engineering
    ↪ shear)."""
    out = np.empty((len(tri_nodes), 3), dtype=float)
    for e, (i1,i2,i3) in enumerate(tri_nodes):
        x1,y1 = pts[i1]; x2,y2 = pts[i2]; x3,y3 = pts[i3]
        A = 0.5*((x2-x1)*(y3-y1)-(x3-x1)*(y2-y1))
        if abs(A) < 1e-14:
            out[e] = 0.0; continue
        B = np.array([
            [y2-y3, 0,     y3-y1, 0,     y1-y2, 0    ],
```

```
            [0,     x3-x2, 0,     x1-x3, 0,     x2-x1],
            [x3-x2, y2-y3, x1-x3, y3-y1, x2-x1, y1-y2]
        ], dtype=float)/(2.0*A)
        u_loc = np.hstack([disp_m[i1,:2], disp_m[i2,:2],
    ↪ disp_m[i3,:2]])
        out[e] = B @ u_loc
    return out

def constitutive_plane_stress(E, nu):
    return (E/(1-nu**2))*np.array([[1,nu,0],[nu,1,0],[0,0,(1-nu)/2]],
    ↪ dtype=float)

def write_vtu_tri(filename, pts, tri_nodes, point_arrays=None,
    ↪ cell_arrays=None):
    os.makedirs(os.path.dirname(filename) or ".", exist_ok=True)
    npts = pts.shape[0]; ncells = len(tri_nodes)
    with open(filename, "w", encoding="utf-8") as f:
        f.write('<?xml version="1.0"?>\n')
        f.write('<VTKFile type="UnstructuredGrid" version="0.1"
    ↪ byte_order="LittleEndian">\n')
        f.write('  <UnstructuredGrid>\n')
        f.write(f'    <Piece NumberOfPoints="{npts}"
    ↪ NumberOfCells="{ncells}">\n')
        # points
        f.write('      <Points>\n')
        f.write('        <DataArray type="Float32"
    ↪ NumberOfComponents="3" format="ascii">\n')
        for x,y in pts: f.write(f'          {x} {y} 0\n')
        f.write('        </DataArray>\n')
        f.write('      </Points>\n')
        # cells
        f.write('      <Cells>\n')
        f.write('        <DataArray type="Int32" Name="connectivity"
    ↪ format="ascii">\n')
        f.write('          ' + " ".join(" ".join(map(str, e)) for e
    ↪ in tri_nodes) + '\n')
        f.write('        </DataArray>\n')
        f.write('        <DataArray type="Int32" Name="offsets"
    ↪ format="ascii">\n')
```

```
        offs = np.cumsum([3]*ncells); f.write('          ' + "
    ↪ ".join(map(str, offs)) + '\n')
        f.write('        </DataArray>\n')
        f.write('        <DataArray type="UInt8" Name="types"
    ↪ format="ascii">\n')
        f.write('          ' + " ".join(['5']*ncells) + '\n')  # 5 =
    ↪ VTK_TRIANGLE
        f.write('        </DataArray>\n')
        f.write('      </Cells>\n')
        # point data
        if point_arrays:
            f.write('      <PointData>\n')
            for name, arr in point_arrays.items():
                f.write(f'        <DataArray type="Float32"
    ↪ Name="{name}" NumberOfComponents="1" format="ascii">\n')
                for v in arr: f.write(f'          {float(v)}\n')
                f.write('        </DataArray>\n')
            f.write('      </PointData>\n')
        # cell data
        if cell_arrays:
            f.write('      <CellData>\n')
            for name, arr in cell_arrays.items():
                f.write(f'        <DataArray type="Float32"
    ↪ Name="{name}" NumberOfComponents="1" format="ascii">\n')
                for v in arr: f.write(f'          {float(v)}\n')
                f.write('        </DataArray>\n')
            f.write('      </CellData>\n')
        f.write('    </Piece>\n')
        f.write('  </UnstructuredGrid>\n')
        f.write('</VTKFile>\n')

def evaluate_to_vtu(mesh, labels_csv, predicts_csv, sample_idx=0,
                    E_val=1e5, nu_val=0.3, out_path="errors_tri.vtu"):
    pts, tri_nodes = vertices_and_tris(mesh)
    num_nodes = pts.shape[0]

    # read rows (vertex order) in mm
    df_ref = pd.read_csv(labels_csv, header=None, dtype=float)
    df_hat = pd.read_csv(predicts_csv, header=None, dtype=float)
```

```
    u_ref = df_ref.iloc[sample_idx].to_numpy(float)
    u_hat = df_hat.iloc[sample_idx].to_numpy(float)
    assert u_ref.size == 2*num_nodes == u_hat.size

    ux_ref_mm, uy_ref_mm = u_ref[:num_nodes], u_ref[num_nodes:]
    ux_hat_mm, uy_hat_mm = u_hat[:num_nodes], u_hat[num_nodes:]

    # to meters for mechanics
    disp_ref = np.column_stack([ux_ref_mm, uy_ref_mm,
    ↪ np.zeros(num_nodes)]) * 1e-3
    disp_hat = np.column_stack([ux_hat_mm, uy_hat_mm,
    ↪ np.zeros(num_nodes)]) * 1e-3

    # strains/stresses
    e_ref = compute_strain_CST(disp_ref, pts, tri_nodes)
    e_hat = compute_strain_CST(disp_hat, pts, tri_nodes)
    C = constitutive_plane_stress(E_val, nu_val)
    s_ref = e_ref @ C.T
    s_hat = e_hat @ C.T

    # absolute errors
    err_u_mm = np.linalg.norm(disp_hat[:,:2]-disp_ref[:,:2], axis=1)
    ↪ * 1e3
    err_e   = np.linalg.norm(e_hat-e_ref, axis=1)
    err_s   = np.linalg.norm(s_hat-s_ref, axis=1)

    write_vtu_tri(out_path, pts, tri_nodes,
                  point_arrays={"ErrDispAbs_mm": err_u_mm},
                  cell_arrays ={"ErrStrainAbs": err_e,
    ↪ "ErrStressAbs_Pa": err_s})
    print("[OK] wrote", out_path)

# --- example (comment): rectangle mesh and evaluation ---
# mesh = RectangleMesh(Point(0.,0.), Point(3.2,0.8), 31, 7)
# evaluate_to_vtu(mesh,
#                 labels_csv="labels_test_2D_meshC.csv",
#                 predicts_csv="predicts.csv",
#                 sample_idx=0,
#                 out_path="errors_meshC_tri.vtu")
```

Listing A.12: Generic evaluator with VTU output (triangles)

Listing A.12 provides a compact evaluator for visual diagnostics. The mesh is converted to a triangle list (with a fan triangulation fallback for non-triangle cells) so that constant-strain triangle (CST) formulas can be applied elementwise. Displacements read from CSV are interpreted in millimetres and converted back to metres for mechanics. The CST 'B' matrices produce the strain triplets $[\varepsilon_{xx}, \varepsilon_{yy}, \gamma_{xy}]$ with engineering shear $\gamma_{xy} = 2\,\varepsilon_{xy}$, and a plane-stress constitutive tensor turns them into stresses. Absolute errors are computed for displacements at points and for strains and stresses over cells, then written to a VTU file with clear array names and units. Paraview can load this file to display field and error maps exactly as shown in the chapter.

A.8 Convolutional U-Net for Force-to-Displacement Mapping

Section 4.3 uses a compact U-Net on a fixed Cartesian lattice to learn the mapping from rasterized forces to rasterized displacements. The pipeline is:

- **Grid mapping:** FEM nodal forces (f_x, f_y) are mapped to two grid channels on a fixed Cartesian lattice by depositing each node's contribution at its corresponding pixel (via an affine letterbox map from physical to grid coordinates), with a small Gaussian applied for anti-aliasing. A binary mask channel encodes the domain. Targets are obtained by interpolating the nodal displacements (u_x, u_y) onto the same grid.
- **Normalization:** robust per-channel scales (e.g., 95th percentile) keep values well conditioned on CPU.
- **Network:** U-Net with encoder channels 64, 128, 256, mirrored decoder with skip connections, and a linear two-channel head for (u_x, u_y).
- **Loss and weights:** masked mean absolute error over in-domain pixels (mask as sample weights). A small total-variation regularizer can be added for stability.
- **Back-projection:** for metrics on the FEM mesh, grid predictions are sampled back at node pixels and de-normalized to millimeters.

Training uses Adam with learning rate 2×10^{-4}, batch size 16, and 30 epochs for the three rectangular examples summarized in Table 4.2. Quantitative and qualitative results (fields and error maps) match Chapter 4.

To make the pipeline reproducible, the listings below provide the exact implementation with brief explanations interleaved. File paths assume the same fixed vertex ordering used when

exporting the FEM CSVs so rasterization and back–projection are consistent.

```
import os, random, numpy as np, pandas as pd
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"

IMG_H, IMG_W = 256, 256
BATCH = 16
EPOCHS = 30  # matches Chapter schedule

prefix = "meshC"  # example: matches previous generator's out_prefix
feat_tr = f"features_train_2D_{prefix}.csv"
feat_te = f"features_test_2D_{prefix}.csv"
lab_tr  = f"labels_train_2D_{prefix}.csv"
lab_te  = f"labels_test_2D_{prefix}.csv"

# optional: coordinates exported once from the mesh (same vertex order)
# e.g. coords saved via: np.savetxt("coords_meshC.csv", coords, delimiter=",")
coords_csv = f"coords_{prefix}.csv"

random.seed(0); np.random.seed(0)
try:
    import tensorflow as tf
    tf.random.set_seed(0)
except Exception:
    pass
```

Listing A.13: Parameters, paths, and fixed seeds

The rasterization stage uses an aspect-preserving letterbox map to send physical coordinates to pixel indices. Nodal values are "splatted" as impulses and lightly diffused with a small Gaussian so that the inputs and targets become dense grid fields. The same affine map is reused to gather predictions back at node locations for mesh-level metrics.

```
import numpy as np
from scipy.ndimage import gaussian_filter

def letterbox_affine(pts, H=IMG_H, W=IMG_W, pad=2.0):
    """Return scale s and offsets (dx,dy) s.t. (x,y)->(px,py) fits inside HxW."""
```

```
    x0,y0 = pts.min(0); x1,y1 = pts.max(0)
    # small padding in physical coords
    dxp = (x1-x0); dyp = (y1-y0)
    x0 -= pad*dxp/IMG_W; x1 += pad*dxp/IMG_W
    y0 -= pad*dyp/IMG_H; y1 += pad*dyp/IMG_H
    sx = (W-1)/(x1-x0+1e-12); sy = (H-1)/(y1-y0+1e-12)
    s = min(sx, sy)
    # center the box
    px_min, py_min = 0.5*(W-1 - s*(x1-x0)), 0.5*(H-1 - s*(y1-y0))
    return (s, -s*x0 + px_min, -s*y0 + py_min)  # (s, dx, dy)

def nodes_to_pixels(pts_xy, H=IMG_H, W=IMG_W):
    """Compute integer pixel indices for each node and a binary
    ↪ occupancy mask."""
    s, dx, dy = letterbox_affine(pts_xy, H, W)
    px = np.clip(np.round(s*pts_xy[:,0] + dx).astype(int), 0, W-1)
    py = np.clip(np.round(s*pts_xy[:,1] + dy).astype(int), 0, H-1)
    mask = np.zeros((H,W), dtype=np.float32); mask[py, px] = 1.0
    return (px, py, mask)

def splat_to_grid(values, px, py, H=IMG_H, W=IMG_W, sigma=1.5):
    """Splat nodal values (N,) to an HxW image via impulse + Gaussian
    ↪ blur."""
    img = np.zeros((H,W), dtype=np.float32)
    # accumulate: if multiple nodes land on same pixel, sum
    ↪ (area-free convention)
    np.add.at(img, (py, px), values.astype(np.float32))
    if sigma and sigma>0:
        img = gaussian_filter(img, sigma=float(sigma))
    return img

def fields_to_grid(fx, fy, px, py, mask, sigma=1.5,
    ↪ include_mask=True):
    """Build input tensor: [fx_img, fy_img, mask] with shape
    ↪ (H,W,C)."""
    gx = splat_to_grid(fx, px, py, sigma=sigma)
    gy = splat_to_grid(fy, px, py, sigma=sigma)
    if include_mask:
        return np.stack([gx, gy, mask], axis=-1)
```

```
    return np.stack([gx, gy], axis=-1)

def grid_to_nodes(grid_uv, px, py):
    """Sample grid predictions (H,W,2) at node pixels -> (N,2) in
    ↪ image units."""
    ux = grid_uv[py, px, 0]; uy = grid_uv[py, px, 1]
    return np.stack([ux, uy], axis=-1)
```

Listing A.14: Affine map, splat to grid, and gather back to nodes

We load the vertex coordinates (to define the rasterization) together with the train/test CSV files. Robust per-channel scales based on the 95th percentile keep forces and displacements well conditioned on CPU. Forces and displacements are normalized independently.

```
# coords are in the same vertex order used to write the CSVs
coords = pd.read_csv(coords_csv, header=None).to_numpy(dtype=float)
    ↪ # (N,2)
px, py, domain_mask = nodes_to_pixels(coords)

# read train/test
Xtr = pd.read_csv(feat_tr, header=None).to_numpy(dtype=float)  #
    ↪ [Fx(0..N-1), Fy(0..N-1)]
Ytr = pd.read_csv(lab_tr,  header=None).to_numpy(dtype=float)  #
    ↪ [Ux(mm), Uy(mm)]
Xte = pd.read_csv(feat_te, header=None).to_numpy(dtype=float)
Yte = pd.read_csv(lab_te,  header=None).to_numpy(dtype=float)

N = coords.shape[0]
assert Xtr.shape[1] == 2*N and Ytr.shape[1] == 2*N

# robust scales (95th percentile across training set) and shifts
    ↪ (zero)
def robust_scale(vals):
    s = np.percentile(np.abs(vals), 95) + 1e-8
    return s

sx_fx = robust_scale(np.abs(Xtr[:,:N]).ravel())
sx_fy = robust_scale(np.abs(Xtr[:,N:]).ravel())
sy_ux = robust_scale(np.abs(Ytr[:,:N]).ravel())
sy_uy = robust_scale(np.abs(Ytr[:,N:]).ravel())
```

```
23
24 def norm_forces(sample):
25     fx = sample[:N]/sx_fx; fy = sample[N:]/sx_fy
26     return fx.astype(np.float32), fy.astype(np.float32)
27
28 def norm_displ(sample_mm):
29     ux = sample_mm[:N]/sy_ux; uy = sample_mm[N:]/sy_uy
30     return ux.astype(np.float32), uy.astype(np.float32)
```

Listing A.15: Data loading and robust per-channel scaling

A small `tf.data` generator converts each CSV row into an input tensor with three channels—force f_x, force f_y, and a binary mask—and a two-channel target with normalized displacements u_x and u_y. The mask is carried alongside and used as a per-pixel weight so only in-domain pixels contribute to the loss.

```
1  import tensorflow as tf
2
3  H, W = IMG_H, IMG_W
4  MASK = domain_mask.astype(np.float32)
5
6  def example_to_tensors(x_row, y_row, blur_sigma=1.5):
7      fx, fy = norm_forces(x_row); ux, uy = norm_displ(y_row)
8      x_img = fields_to_grid(fx, fy, px, py, MASK, sigma=blur_sigma,
       ↪ include_mask=True)    # (H,W,3)
9      y_img = np.stack([
10         splat_to_grid(ux, px, py, sigma=blur_sigma),
11         splat_to_grid(uy, px, py, sigma=blur_sigma)
12     ], axis=-1)  # (H,W,2)
13     w_img = MASK  # (H,W)
14     return x_img.astype(np.float32), y_img.astype(np.float32),
       ↪ w_img.astype(np.float32)
15
16 def make_tf_dataset(X, Y, batch=BATCH, shuffle=True):
17     def gen():
18         idx = np.arange(len(X))
19         if shuffle: np.random.shuffle(idx)
20         for i in idx:
21             yield example_to_tensors(X[i], Y[i])
22     sig = (
```

```
        tf.TensorSpec(shape=(H,W,3), dtype=tf.float32),
        tf.TensorSpec(shape=(H,W,2), dtype=tf.float32),
        tf.TensorSpec(shape=(H,W),   dtype=tf.float32),
    )
    ds = tf.data.Dataset.from_generator(gen, output_signature=sig)
    if shuffle: ds = ds.shuffle(buffer_size=8*batch)
    return ds.batch(batch).prefetch(tf.data.AUTOTUNE)

ds_tr = make_tf_dataset(Xtr, Ytr, shuffle=True)
ds_te = make_tf_dataset(Xte, Yte, shuffle=False)
```

Listing A.16: Dataset generator with sample weights (mask)

The model is a compact U-Net that mirrors the segmentation Model I for architectural consistency: two convolutional blocks at each scale with batch normalization, down to 256 channels at the bottleneck, and a linear two-channel head that predicts normalized displacements.

```
from keras import Input, Model
from keras.layers import Conv2D, MaxPooling2D, Conv2DTranspose,
    ↪ concatenate, BatchNormalization
from keras.optimizers import Adam

def conv_block(x, ch):
    x = Conv2D(ch, 3, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)
    x = Conv2D(ch, 3, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)
    return x

def unet_fx2u(h=IMG_H, w=IMG_W, in_ch=3, out_ch=2):
    inp = Input((h, w, in_ch))
    # Down
    c1 = conv_block(inp, 64);  p1 = MaxPooling2D(2)(c1)
    c2 = conv_block(p1, 128);  p2 = MaxPooling2D(2)(c2)
    c3 = conv_block(p2, 256)
    # Up
    u2 = Conv2DTranspose(128, 2, strides=2, padding='same')(c3)
    u2 = concatenate([u2, c2]); u2 = conv_block(u2, 128)
    u1 = Conv2DTranspose(64, 2, strides=2, padding='same')(u2)
    u1 = concatenate([u1, c1]); u1 = conv_block(u1, 64)
```

```
    out = Conv2D(out_ch, 1, activation='linear')(u1)  # normalized displacements
    return Model(inp, out)
```

Listing A.17: Compact U-Net for force-to-displacement

Training minimizes a masked mean absolute error, augmented with a light total-variation term to discourage spurious high-frequency oscillations. Both operate in normalized units, and the binary mask enters the loss as a per-pixel weight via the custom training loop.

```
from keras import backend as K

TV_W = 1e-4  # small TV weight

def masked_mae(y_true, y_pred, w):
    # y: (B,H,W,2); w: (B,H,W,1) broadcasted
    diff = K.abs(y_pred - y_true) * w
    denom = K.maximum(K.mean(w), K.epsilon())
    return K.sum(diff) / (2.0 * K.cast(denom * tf.shape(y_true)[0], K.floatx()))

@tf.function
def train_loss(y_true, y_pred, w):
    # TV on each displacement channel (inside mask)
    tv = tf.image.total_variation(y_pred)  # shape (B,)
    return masked_mae(y_true, y_pred, w) + TV_W * tf.reduce_mean(tv)

# Keras wrapper so we can pass sample_weight=mask and reuse MAE for logs
def mae_metric(y_true, y_pred):
    return tf.reduce_mean(tf.abs(y_pred - y_true))
```

Listing A.18: Loss: masked MAE + TV regularization

The custom loop exposes masking and logging explicitly. The optimizer is Adam with learning rate 2×10^{-4}, batch size 16, and 30 epochs as specified in the chapter.

```
model = unet_fx2u()
opt = Adam(2e-4)

@tf.function
def train_step(x, y, w):
```

```
    with tf.GradientTape() as tape:
        yhat = model(x, training=True)
        loss = train_loss(y, yhat, w[...,None])  # (B,H,W,1)
    grads = tape.gradient(loss, model.trainable_variables)
    opt.apply_gradients(zip(grads, model.trainable_variables))
    return loss, mae_metric(y, yhat)

@tf.function
def val_step(x, y, w):
    yhat = model(x, training=False)
    loss = train_loss(y, yhat, w[...,None])
    return loss, mae_metric(y, yhat)

for epoch in range(1, EPOCHS+1):
    # train
    tr_losses, tr_mae = [], []
    for xb, yb, wb in ds_tr:
        l, m = train_step(xb, yb, wb)
        tr_losses.append(l.numpy()); tr_mae.append(m.numpy())
    # val
    va_losses, va_mae = [], []
    for xb, yb, wb in ds_te:
        l, m = val_step(xb, yb, wb)
        va_losses.append(l.numpy()); va_mae.append(m.numpy())
    print(f"[{epoch:02d}/{EPOCHS}] "
          f"loss={np.mean(tr_losses):.4f} mae={np.mean(tr_mae):.4f} |
    ↪ "
          f"val_loss={np.mean(va_losses):.4f}
    ↪ val_mae={np.mean(va_mae):.4f}")
```

Listing A.19: Compile and training loop with masked sample weights

Finally, grid predictions are sampled back at the original node pixels and de-normalized to millimetres. The resulting `predicts.csv` is compatible with the VTU evaluator (Listing A.12) to compute displacement, strain, and stress error fields on the mesh.

```
# build tensors for the entire test set (no shuffle)
x_imgs = []
for i in range(len(Xte)):
    fx, fy = norm_forces(Xte[i])
```

```
    x_imgs.append(fields_to_grid(fx, fy, px, py, MASK, sigma=1.5,
    ↪ include_mask=True))
x_imgs = np.asarray(x_imgs, dtype=np.float32)  # (T,H,W,3)

# predict on CPU
yhat_imgs = model.predict(x_imgs, verbose=0)   # (T,H,W,2)

# sample at node pixels and de-normalize back to mm, row-wise
rows = []
for k in range(yhat_imgs.shape[0]):
    uv_norm = grid_to_nodes(yhat_imgs[k], px, py)  # (N,2) normalized
    ux_mm = (uv_norm[:,0] * sy_ux).astype(np.float32)
    uy_mm = (uv_norm[:,1] * sy_uy).astype(np.float32)
    rows.append(np.concatenate([ux_mm, uy_mm], axis=0))
rows = np.asarray(rows, dtype=np.float32)

out_csv = "predicts.csv"
np.savetxt(out_csv, rows, delimiter=",")
print("[OK] wrote", out_csv, "with shape", rows.shape)
```

Listing A.20: Project grid predictions back to nodes and save CSV

A.9 MAgNET for Mesh-Based Force-to-Displacement Mapping

MAgNET operates directly on the FEM mesh to avoid rasterization, following [32, 33]:

- **Adjacency:** a symmetric Boolean adjacency is built with self-loops; in addition to mesh edges, nodes belonging to the same element are connected (intra-element cliques).
- **Aggregation:** multi-channel aggregation layers propagate information over the neighborhood defined by the adjacency (optionally using A^2 to enlarge the effective receptive field).
- **Pooling/unpooling:** static partitions of the graph produce disjoint cliques for pooling; unpooling restores features to the finer graph. Skip connections concatenate encoder features with decoder features at matching levels.
- **Head:** a linear node-wise head outputs two channels per node (u_x, u_y) in the mesh DOF layout.

We use three pooling levels with channel widths $[16, 32, 64, 128]$ across stages, Adam with initial learning rate 10^{-4} and a simple schedule that is constant for the first 10 epochs and then decays. Batch size is 4 and training runs for 10 epochs, retaining the best validation checkpoint. Inputs are per-DOF force vectors; outputs are per-DOF displacements, later reordered to per-node (x, y) for saving and comparison. Strains and stresses are recovered a posteriori under plane stress as in the CNN section, so metrics are comparable.

This section presents the mesh-native surrogate built on MAgNET. Unlike grid CNNs, MAgNET operates directly on the finite-element graph, which preserves the original topology and connectivity. We begin with a lightweight driver that selects the case, loads train/test arrays, and looks up the total degrees of freedom (DoF) and spatial dimension from a small configuration registry.

```
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"
import numpy as np
import matplotlib.pyplot as plt
from layers.pool_utils import pooled_adjacencies
from models import Models
from utils import *
from global_vars import *

case = "circle"
type = 'magnet'

# Load train/test arrays: [X_train, Y_train, X_test, Y_test]
data = get_data(case, type)

# Get total dof and spatial dim from globals
dof, dim = dofdim[type][case]
```

With the data in memory, we build the graph hierarchy that underpins the encoder–decoder. The fine-level adjacency is computed once from element connectivity. Static pooling partitions (read from `global_vars`) define coarse graphs; `pooled_adjacencies` returns the coarsened adjacencies together with the subgraph mappings used by the pool and unpool operators. Channel widths are chosen per case, and a small factory constructs the correct MAgNET head.

```
if type == 'magnet':
    A = get_adjacency(case)
    no_poolings, poolseed = magnet_pool_details[case]
```

```
    pooled_adjs, subgraphs = pooled_adjacencies(A,
    ↪ no_poolings=no_poolings, poolseed=poolseed)

    channels = magnet_channels[case]
    input_shape = (dof,)
    network = Models(input_shape, channels)

    # Pick the right graph U-Net head for the data dimensionality
    if case in
    ↪ ['2dhole','2dlshape','2dretang','2L','forma2','forma_ap','circle']:
        model = network.magnet_2d(pooled_adjs, subgraphs)
    elif case == '3dbeam':
        model = network.magnet_3dbeam(pooled_adjs, subgraphs)
    else:
        model = network.magnet_3dbreast(pooled_adjs, subgraphs)
```

Training can proceed from scratch or from previously saved weights. A simple learning-rate scheduler keeps the rate high early in training and tapers it later. The wrapper `network.train` encapsulates the fit loop and writes weights to disk so experiments are repeatable.

```
training = True

if training == False:
    model.load_weights(weights_path+case+'_'+type+'.h5')
    print("=== Pretrained weights are assigned to the model ===")
else:
    no_epochs = epochs[type][case]
    new_weights_path = "/.../saved_models/circle.weights.h5"
    lr_scheduler = lrate_scheduler(case,type)
    network.train(model, data, new_weights_path, lr_scheduler,
    ↪ no_epochs)
    model.load_weights(new_weights_path)
```

After training, inference is vectorized over the test set. For CNN variants, tensors are reshaped; for MAgNET the feature rows already match the expected DoF layout. Predicted DoFs are then reordered into the per-node interleaved form $[1x, 1y, (1z), 2x, 2y, (2z), \dots]$, optional padding is removed for the padded 2D case, and both CSV and NPY artifacts are written for downstream evaluation.

```
test_features = data[-2]
```

```
n_test = len(test_features)
predictions = model.predict(test_features, batch_size=4)

# For CNN only: flatten back to (n_test, dof)
if type == 'cnn':
    predictions = predictions.reshape((n_test, dof))

# Convert to per-node [1x,1y,(1z), 2x,2y,(2z), ...]
predictions = reorder_dof(predictions, dof, dim)

# Optional (CNN L-shape): drop padded DOFs
if case == '2dlshape' and type == 'cnn':
    predictions = remove_pad(predictions)

# Save predictions
csv_filename = prediction_path + case + '_' + type + '_predicts.csv'
pd.DataFrame(predictions).to_csv(csv_filename, index=False,
    ↪ header=False)
np.save(prediction_path+case+'_'+type+'_predicts.npy', predictions)
print("=== Predictions are saved in : {}".format(prediction_path))

# Print a brief model summary if needed
network.print_summary(model, print=False)
```

The 2D head follows a U-Net pattern on graphs. Each resolution applies two graph aggregation (MAg) layers, then pools to a coarser graph defined by the subgraph map. At the bottleneck, two more MAg layers act on the coarsest adjacency. Unpooling lifts features back to finer graphs and concatenation restores skip connections from the down path. A final MAg layer with linear activation outputs the two displacement components per node.

```
def magnet_2d(self, pooled_adjacencies, subgraphs):
    c0, c1, c2, c3 = self.channels
    A, Ap1, Ap2, Ap3 = pooled_adjacencies
    subgraph1, subgraph2, subgraph3 = subgraphs

    inputs = Input(shape=self.input_shape)

    # Level 0 (fine graph)
    mag1 = self.mag(c0, A, 2)(inputs)
```

```
    mag1 = self.mag(c0, A, 2)(mag1)
    pool1 = G_Pool(subgraph1, nodes=len(A))(mag1)

    # Level 1
    mag2 = self.mag(c1, Ap1, 2)(pool1)
    mag2 = self.mag(c1, Ap1, 2)(mag2)
    pool2 = G_Pool(subgraph2, nodes=len(Ap1))(mag2)

    # Level 2
    mag3 = self.mag(c2, Ap2, 2)(pool2)
    mag3 = self.mag(c2, Ap2, 2)(mag3)
    pool3 = G_Pool(subgraph3, nodes=len(Ap2))(mag3)

    # Bottleneck (coarsest graph)
    mag4 = self.mag(c3, Ap3, 2)(pool3)
    mag4 = self.mag(c3, Ap3, 2)(mag4)

    # Up 1
    up1 = G_Unpool(subgraph3, nodes=len(Ap2))(mag4)
    up1 = Concatenate(axis=1)([mag3, up1])
    mag5 = self.mag(c2, Ap2, 2)(up1)
    mag5 = self.mag(c2, Ap2, 2)(mag5)

    # Up 2
    up2 = G_Unpool(subgraph2, nodes=len(Ap1))(mag5)
    up2 = Concatenate(axis=1)([mag2, up2])
    mag6 = self.mag(c1, Ap1, 2)(up2)
    mag6 = self.mag(c1, Ap1, 2)(mag6)

    # Up 3 (back to fine graph)
    up3 = G_Unpool(subgraph1, nodes=len(A))(mag6)
    up3 = Concatenate(axis=1)([mag1, up3])
    mag7 = self.mag(c0, A, 2)(up3)
    mag7 = self.mag(c0, A, 2)(mag7)

    out = self.mag(2, A, 2, activation=None)(mag7)
    model = Model(inputs=inputs, outputs=out)
    return model
```

Data ingestion mirrors the grid pipeline but keeps arrays flat for MAgNET. The loader reads the four CSVs, reshaping only for CNN variants. A small special case exists for a padded L-shape so all samples share a common rectangular tensor.

```
import numpy as np
import pandas as pd
import itertools
from global_vars import *

srcpath = os.path.dirname(os.getcwd())
datapath = srcpath+'/MagNet'
print(datapath)

def get_data(case, type):
    print_magnet()

    if type == 'cnn' and case in ['2dhole','3dbreast']:
        raise ValueError('This case is not implemented')

    print("=== Loading dataset for the "+ type.upper() +" network and
    ↪ "+ case + " case === ")

    X_train =
    ↪ pd.read_csv(datapath+'/FEMData/features_train_'+str(case)+'.csv',
    ↪ header=None)
    Y_train =
    ↪ pd.read_csv(datapath+'/FEMData/labels_train_'+str(case)+'.csv',
    ↪ header=None)
    X_test  =
    ↪ pd.read_csv(datapath+'/FEMData/features_test_'+str(case)+'.csv',
    ↪ header=None)
    Y_test  =
    ↪ pd.read_csv(datapath+'/FEMData/labels_test_'+str(case)+'.csv',
    ↪ header=None)

    if case == '2dlshape' and type == 'cnn':
        case += '_padded'

    if type == 'cnn':
```

```
        n_train = len(X_train); n_test = len(X_test)
        if case == '3dbeam':
            dim, n_x, n_y, n_z = cnn_input_shapes[case]
            X_train = X_train.reshape(n_train, dim, n_x, n_y, n_z)
            Y_train = Y_train.reshape(n_train, dim, n_x, n_y, n_z)
            X_test  = X_test.reshape(n_test, dim, n_x, n_y, n_z)
            Y_test  = Y_test.reshape(n_test, dim, n_x, n_y, n_z)
        else:
            case = case.replace('_padded','')
            dim, n_x, n_y = cnn_input_shapes[case]
            X_train = X_train.reshape(n_train, dim, n_x, n_y)
            Y_train = Y_train.reshape(n_train, dim, n_x, n_y)
            X_test  = X_test.reshape(n_test, dim, n_x, n_y)
            Y_test  = Y_test.reshape(n_test, dim, n_x, n_y)

    return [X_train, Y_train, X_test, Y_test]
```

Adjacency construction follows the finite-element mesh: for each element, all its nodes are connected to one another, which forms an intra-element clique. This densifies the local neighborhood relative to edge-only graphs and improves aggregation. An optional n-hop expansion provides a quick way to enlarge the receptive field without stacking more layers.

```
def get_adjacency(case: str):
    file_path = os.path.join(datapath, 'connectivity',
    ↪ f'connect_{case}.csv')
    df_adj = pd.read_csv(file_path, header=None, sep=';')

    # Split single-column strings on commas -> ints
    element_connectivity = df_adj[0].str.split(',',
    ↪ expand=True).astype(int).to_numpy()
    n_elements = element_connectivity.shape[0]

    n_nodes = element_connectivity.max() + 1
    A = np.zeros((n_nodes, n_nodes))

    for i in range(n_elements):
        element = list(element_connectivity[i, :])
        for pair in itertools.product(element, repeat=2):
            A[pair] = 1
    return A
```

```
17
18 def nth_adjacency(A, n):
19     A_n = A
20     if n == 1:
21         return A
22     for _ in range(n - 1):
23         A_n = np.dot(A_n, A)
24     rows, cols = np.nonzero(A_n)
25     A_n[rows, cols] = 1
26     return A_n
```

A few utilities complete the training loop. The learning-rate scheduler returns a callable for use in callbacks; DoF helpers convert between blocked and interleaved layouts so outputs align with FEM post-processing; padding removal and "largest displacement" are small conveniences for plotting and selection.

```
1  def lrate_scheduler(case,type):
2      if case == '2dlshape' or case =='2dhole' or case == '2L' or case
       ↪ == "forma6" or "circle":
3          def scheduler(epoch, lr):
4              if epoch < 10:           return lr
5              elif 10 < epoch < 200:   return lr - (0.0001 -
       ↪ 0.00001)/190
6              elif 200 < epoch < 900:  return lr - (0.00001 -
       ↪ 0.000001)/700
7              else:                    return lr
8      elif case =='3dbeam' and type =='cnn':
9          def scheduler(epoch, lr):
10             if epoch < 85:           return lr * np.exp(-0.001 *
       ↪ epoch)
11             else:                    return lr
12     else:
13         def scheduler(epoch, lr):
14             if epoch < 10:           return lr
15             elif 10 <= epoch < 100:  return lr - (0.0001 -
       ↪ 0.000001)/90
16             else:                    return lr
17     return scheduler
18
19 def reorder_dof(predictions, dof, dim):
```

```
    reorder_predictions = np.copy(predictions)
    for i in range(len(predictions)):
        reorder_predictions[i] = orginal_order(predictions[i], dof,
    ↪ dim)
    return reorder_predictions

def orginal_order(array, dof, dim):
    original_dof = np.zeros((dof,))
    if dim == 2:
        n = dof//2
        dof_x = array[:n]; dof_y = array[n:]
        for i in range(n):
            original_dof[2*i] = dof_x[i]
            original_dof[2*i+1] = dof_y[i]
    elif dim == 3:
        n = dof//3
        dof_x = array[:n]; dof_y = array[n:2*n]; dof_z = array[2*n:]
        for i in range(n):
            original_dof[3*i]   = dof_x[i]
            original_dof[3*i+1] = dof_y[i]
            original_dof[3*i+2] = dof_z[i]
    return original_dof

def remove_pad(predictions):
    unpad_indices =
    ↪ np.load(srcpath+'/main/connectivity/unpad_map_2dlshape.npy')
    return predictions[:, unpad_indices]

def max_disp_index(predictions, dof, dim):
    n_test = len(predictions)
    max_disps = np.zeros(n_test)
    for i in range(n_test):
        pred = predictions[i].reshape(int(dof/dim), dim)
        pred_norm = np.linalg.norm(pred, axis=1)
        max_disps[i] = np.max(np.abs(pred_norm))
    return int(np.argmax(max_disps))
```

Global variables centralize paths and case-specific hyperparameters, keeping the main script declarative. They specify where datasets and weights live, the DoF and dimension per run, graph

U-Net widths, pooling levels and seeds, and (optionally) CNN shapes if the grid baseline is used.

```
# Paths
srcpath = os.path.dirname(os.getcwd())
datapath = srcpath + "/data/FEMData/"
weights_path = srcpath + "/data/saved_models/"
prediction_path = srcpath + "/data/"
visualisation_path = srcpath + "/postprocess/visualisation/"

# DOFs and spatial dim per run
dofdim = { 'magnet': {...}, 'cnn': {...} }

# Graph U-Net widths per level
magnet_channels = { ... }

# Pooling levels and seeds
magnet_pool_details = { ... }

# CNN grids and widths (only if using CNN)
cnn_input_shapes = { ... }
cnn_channels = { ... }

# Epoch counts
epochs = { 'magnet': {...}, 'cnn': {...} }
```

Putting the pieces together, the flow is: choose a case and model type, load arrays, build the pooled graph hierarchy, instantiate the graph U-Net, train or load, predict on the test set, reorder DoFs to the per-node layout, and save artifacts. The helper routines cover data I/O, adjacency, scheduling, DoF order, and padding; the globals act as a registry so new meshes can be added without code changes.