QATAR UNIVERSITY

COLLEGE OF ENGINEERING

GENERATIVE ADVERSARIAL NETWORKS AS A METHOD TO PREDICT STRESSES IN STRUCTURES

BY

STEPHEN DOUGLAS TRENT

A Thesis Submitted to

the College of Engineering

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Mechanical Engineering

January 2021

COMMITTEE PAGE

The members of the Committee approve the Thesis of

Stephen Douglas Trent defended on Wednesday, December 2nd,

2020.

Dr. Jamil Renno
Thesis Supervisor

Dr. Mohamed M. Shadi
Committee Member

Dr. Sadok Sassi
Committee Member

Dr. John-John Cabibihan
Committee Member

Approved:

Dr. Khalid Kamal Naji, Dean, College of Engineering

ABSTRACT

TRENT,STEPHEN,D., Masters: January: 2021,

Masters of Science in Mechanical Engineering

Title: Generative Adversarial Networks as a Method to Predict Stresses in Structures

Supervisor of Thesis: Dr Jamil Renno

There have been continuous advances in the field of Finite Element Analysis (FEA) allowing designers, architects, engineers and the public at large increasing ease and access. The methods of implementation however, have remained largely unchanged since their inception in the 1960s relying predominantly on costly computational software and hardware to carry out time and computing resource intensive calculations. furthermore, alterations to any simulation inputs, constraints or design parameters potentially nullify the previous results and require subsequent additional simulations. While there have been strides made towards adaptive FEA software [such as Ansys Discovery live®for instance] these too tend to be prohibitively more costly or resource intensive than their contemporary counterparts. As an additional consequence, the analysis of a component, structure or system is almost always done remotely, and ideally well before manufacture. Similarly, in situations which require active monitoring, the telemetry is required to be passed to remote systems capable of carrying out the FEA computations.

With the advent and rapid development of Artificial Intelligence (AI), more specifically advancements in Artificial Neural Networks (ANNs) as a new toolset for the solution of complex problems, the question arises, "Can Neural Networks be trained to emulate Finite Element Analysis?". This is the basis on which this work centres:

Utilising a conventionally generated FEA dataset, a conditional Generative Adversarial Network (cGAN) is "taught" the physical behaviour of platework of varying geometry and material properties and subjected to randomly placed varying loadings. The subsequent "trained" model is hence capable of generating predictions for arbitrary inputs which correspond to the domain of input on which it was trained. Three experiments resulted in separate cGAN generator models trained to infer deflections from forces, stresses from deflections and stresses from forces respectively. After a moderate training regime of 200 Epochs each, the outputs of the models are shown to be in reasonable agreement to the ground truth with mean errors in the range of 5-10 %. Whilst not perfect FEA replacements, the trained models show potential for improvement and in their existing implementation allow for near real-time iterations or testing of hypothetical force additions via a purpose built application. Furthermore, this adds credence to deploying systems which implement purpose trained models for the ablity to self monitor structures in situ in realtime.

DEDICATION

To my wife Karla. For putting up with me at the best of times and pushing me on through the worst.

ACKNOWLEDGMENTS

The works described in this Thesis were carried out at the Department of Mechanical Engineering of Qatar University between August 2019 and May 2020. Whilst not based on the initial outline we developed at the onset of the Thesis, I am greatly appreciative for the guidance, support and advice provided by my Thesis advisor, Dr. Jamil Renno and owe him an immense debt of gratitude for his willingness to entertain and redirect my propensity to drift off on a tangent in order to keep both me and this work on track. As a result this has been an immensely gratifying and rewarding learning experience.

I would further like to aknowledge and express gratitude to my family and friends who remained encouraging and ever steadfast with their support from the sidelines especially as we all had to deal with the unprecedented hardships brought about by the COVID-19 pandemic.

I also gratefully acknowledge the partial financial support provided by the Qatar National Research Fund through the National Priorities Research Program under grant number NPRP 11S-1220-170112 and Qatar University Internal Grant QUCG-CENG-19/20-6.

Thank you all!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AI	Artificial Intelligence
ANN	Artificial Neural Network
AR	Augmented Reality
cGAN	conditional Generative Adversarial Network
CMS	Component Mode Synthesis
CNN	Convolutional Neural Network
DAE	Direct Absolute Error
DL	Deep Learning
DOF	Degree(s) of Freedom
EVP	Eigen Value Problem
FEA	Finite Element Analysis
FEM	Finite Element Method
GPU	Graphics Processing Unit
GUI	Graphical User Interface
ML	Machine Learning
MLP	Multi-layer Perceptron
MOR	Model Order Reduction
NASA	National Aeronautic and Space Administration
PDE	Partial Differential Equation
PCA	Principal Component Analysis
NASTRAN	NASA Structural Analysis
RDMN	Response Dependant Nonlinear Mode
RPD	Relative Percent Difference
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
VR	Virtual Reality

1 Introduction

1.1 Background

Real world systems, processes and mechanisms are fairly complex or sophisticated often requiring the usage of approximation based experimental studies or numerical methods when a straightforward analytical equation or solution is unavailable. Furthermore, many physical laws and phenomena are described by the use of partial differential equations (PDEs). Since most PDEs cannot be solved analytically, their solutions can be obtained through discretization followed by numerical methods. The *Finite Element Method* (FEM) is one such numerical technique which relies on the discretization of a problem into smaller discrete constituent parts i.e. 'finite elements'. Various assumptions and idealisations may be utilized to simplify the problem but ultimately equations describing these finite elements are incorporated into a larger arrangement of equations which collectively model the entire problem. The subsequent solution of these sets of equations accurately approximates the desired solution, within the constraints of the aforementioned assumptions and idealisations. *Finite Element Analysis* (FEA), the practical application of FEM, allows for the accurate study of systems and for estimations to be made about them under varying scenarios. Today FEM and by extension FEA are amongst the most widespread approaches employed in the prediction of material responses to real-world forces, vibration, heat, fluid flow and other physical effects. The earliest roots of which can be traced back to a 1956 paper by M.J. Turner et al and in which the term "finite elements" is coined [1]. Following shortly thereafter NASA developed the first structural analysis software "*NASA Structural Analysis*" (NASTRAN) which employed the FEA theoretical methodologies available at the time towards the development of vehicles in their aerospace programmes [2]. Although rudimentary by modern standards, the success of such software drove continual advances in FEA research and development. Whilst not yet fully eliminating the need, the growing sophistication and trust in contemporary commercial applications such as ABAQUS, ANSYS, COMSOL, LS-DYNA, and NASTRAN, among others, has led to a significant reduction in the number of physical experiments and prototypes required during design and optimization of processes, devices and products. Today the usage of FEA in the fields of civil, structural, and mechanical engineering is ubiquitous with further applications in the fields of biology and medicine [3; 4] to geology [5] and oceanography [6; 7] becoming increasingly common.

1.2 Fundamental Concepts of the FEM

The following subsections, whilst by no means comprehensive, aim to provide a concise understanding of the fundamental underlying concepts and principles behind FEM and FEA. For additional information about FEM, the interested reader is advised to consult [8; 9; 10]. For those familiar with the topic, feel free to advance ahead to Section 1.3.

1.2.1 Stress and Strain

Continuum mechanics is a branch of physics which assumes that a substance, body, or object entirely fills the space it occupies. It ignores discontinuities (i.e. interatomic distances) as length scales are usually orders of magnitude larger. Many physics laws such as conservation of energy, mass, and momentum are based on this postulation with a continuum of known boundary referred to as a domain. To determine the reaction of a body under an applied force (internal or external), it is required to know the force intensity and the body's cross sectional area. In a continuous material, *stress* (σ) is defined as the average intensity of the force(s) divided by the body's cross sectional area. *Strain* (ε) corresponds to the amount of elongation or shortening per unit length caused due to the stress. These basic definitions refer to average values, as stress and strain can be irregularly distributed over the cross section of the body. Strain is a dimensionless parameter, whilst stress is usually referenced in Pascals ($1Pa = \frac{1N}{m^2}$) and psi ($1psi = \frac{1lbf}{in^2}$). Figure 1.1 and Figure 1.2 illustrate both concepts.

$$Strain(\varepsilon) = \frac{Elongation}{Original\,length} = \frac{\delta}{L_0} \tag{1.1}$$

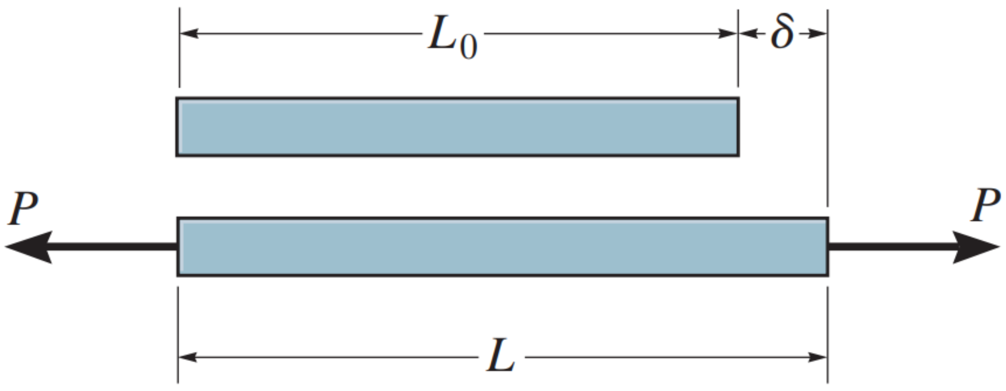


Figure 1.1: Elongated solid bar under axial load. Average strain is defined by elongation, δ, divided by the bar's original length, L_o. [11]

$$Stress(\sigma) = \frac{Force}{Cross-sectional\,Area} = \frac{F}{A} \tag{1.2}$$

Stresses are broadly classified as either *compressive*, *tensile*, or *shear stress*. Compressive and tensile stresses act perpendicular to the cross-sectional area of an el-

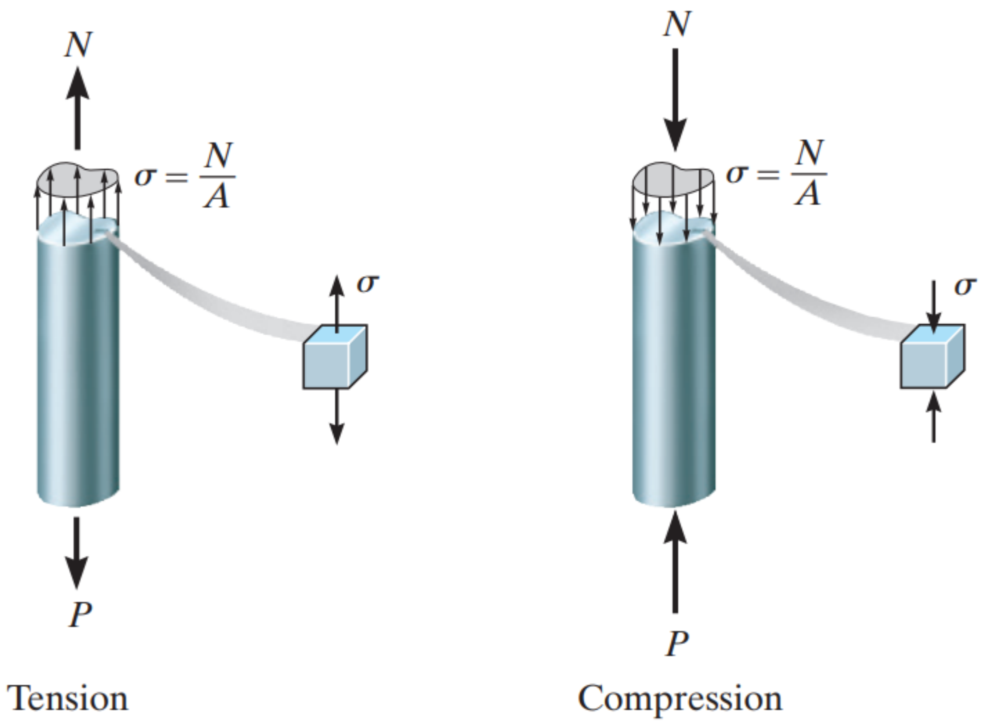


Figure 1.2: Solid rod under axial loading. Average normal stress can be defined by force divided by the bar's cross-sectional area [11]

ement, inducing shortening or elongation of a material when applied. Shear stress is coplanar with a material's cross section. Stress acting at an angle can be expressed in terms of its normal components. Sign conventions in literature establish that the magnitudes for normal tensile stresses are positive whereas for compressive stresses are negative. Stress elements are a useful representation of stress acting in a determined point on a body. Figure 1.3 (a) shows the general stress state of an infinitesimal body element. For Cartesian coordinates, normal stress components are identified by subscripts x, y and z which relate to each Cartesian axis. Hence σ_x, σ_y and σ_z act on surfaces perpendicular to the x, y and z axes respectively. Shear stresses require two subscripts where the first indicates the perpendicular surface upon which the shear stress acts, and the second denotes its direction. For instance Figure 1.3 illustrates how τ_{xy} acts on a surface perpendicular to the x-axis, with direction along the y-axis. For equilibrium, shear stresses with crossed subscripts generally have the same magnitude (i.e. $\tau_{AB} = \tau_{BA}$). Therefore, τ_{xy} equals τ_{yx}, τ_{zx} equals τ_{xz}, and τ_{zy} equals τ_{yz}.

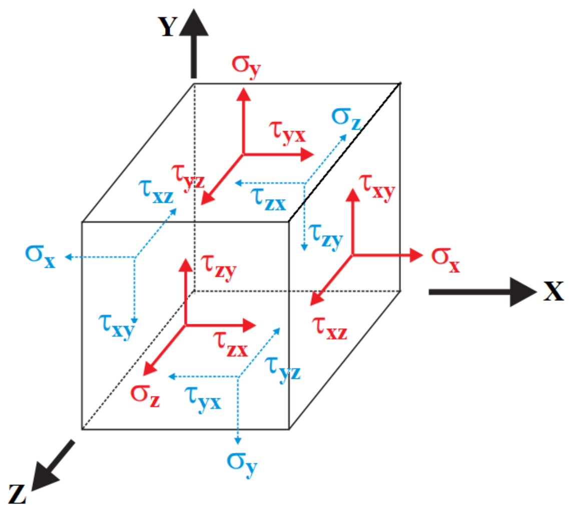


Figure 1.3: Component stresses on an infinitesimal element. Adapted from [9]

1.2.2 Stress-Strain curve and Young's Modulus

An important engineering consideration when analysing materials under stress and strain is the relationship exhibited between these two parameters themselves. This is frequently accomplished by inspecting the stress-strain curve produced by measuring the amount of strain a material undergoes within tensile or compressive stress ranges. The curve generated is unique for individual materials and allows for the inference of resultant mechanical properties. In Figure 1.4 a generic stress-strain curve with two primary stages is identified. The first, an "elastic" region, extends from the zero-load condition to the point referred to as the *yield stress*, σ_{yield}. At any point within this region the application and subsequent removal of loads would result in the material deforming yet returning to its original dimensions. If stresses are applied beyond the yield strength into the second "plastic" region, the material "yields" and permanent deformations occur. The *ultimate stress*, $\sigma_{ultimate}$, is the point of maximum stress along the stress-strain curve beyond which further stress eventually leads to material failure/rupture. Brittle materials such as ceramics have little to no plastic regions and rupture just beyond ultimate stress. Conversely ductile materials such as aluminium, steel or copper, exhibit considerable plastic regions and experience far larger amounts of strain before rupture.

The slope of the stress-strain curve within the elastic region is referred to as *Young's* (*or Elastic*) *modulus* and is a measure of the resistivity to deformation of a material when forces are applied. Soft materials such as rubber would thus have a flatter slope and be relatively easily deformed whilst tougher materials such a steels require significant more force to induce deformation and hence have steeper slopes. Figure 1.4 in conjunction with Equation 1.3 and Equation 1.4 provide the means to calculate E.

$$E = \frac{Stress}{Strain} = \frac{\sigma_{pl}}{\varepsilon} \tag{1.3}$$

substituting from Equation 1.2 and Equation 1.1 yields:

$$E = \frac{F/A}{\delta/L_0} = \frac{FL_0}{A\delta} \tag{1.4}$$

1.2.3 Material Isotropy and Poisson's ratio

Materials which exhibit uniform mechanical properties at all locations and along all directions within the material are referred to as *'isotropic'*. Typically this is common of fluids and gasses as well as commercial metal grades of copper, steel and aluminium. Corollary anisotropic materials are characterised by mechanical properties that vary directionally. This can be found naturally within wood or in materials such as layered

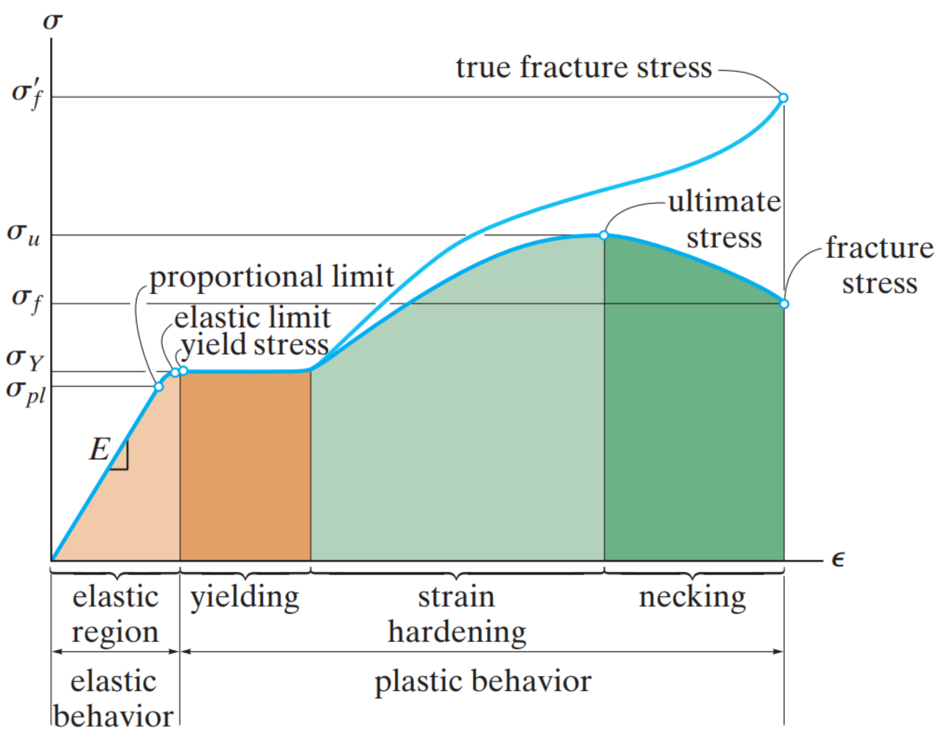


Figure 1.4: Generic Stress-Strain curve for a ductile material indicating mechanical properits such as elastic and plastic regions, elastic modulus and fracture stresses [11].

composites like carbon fibre which can be tailored to meet directional tensile needs. Furthermore manufacturing processes such as cold rolling may also induce anisotropic behaviour due to the alignment of the crystalline cells within metals [11].

When an isotropic material undergoes uniform compression (or axial elongation) its behaviour can be quantified by its *Bulk modulus*, B. This is a measure of how compressible a material is with relation to its volume, V (or density, ρ) and applied loading, P as given in Equation 1.5 and Equation 1.6:

$$B = -V\frac{dP}{dV} \tag{1.5}$$

or

$$B = \rho\frac{dP}{d\rho} \tag{1.6}$$

When in elongation, the material experiences transversal compression opposite to the axis of elongation (see Figure 1.5). Within the elastic range the ratio of transverse to longitudinal strains is referred to as *Poisson's ratio*, v. A Poisson's ratio of 0 would thus imply a material is perfectly compressible whilst a value of 0.5 implies perfectly incompressible. Metals have Poisson's ratio values around 0.3.

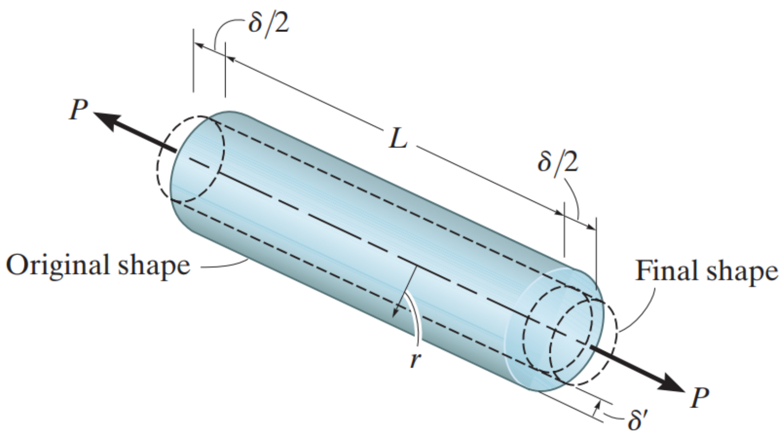


Figure 1.5: Illustration of Poisson effect due to material under axial tensile load. The applied load both elongates the bars material whilst resulting in a transverse compression as a result of the Poisson effect [11] .

The relationship between Bulk modulus, Elastic modulus and Poisson's ratio for an isotropic material within the elastic region is expressed in Equation 1.7.

$$B = -\frac{E}{3(1-2v)} \tag{1.7}$$

1.2.4 Plane, Principal and Maximum Shear Stresses

The magnitude of normal and shear stresses varies with rotation angles along the planes in which they are measured. If we seek to obtain values at specific angles we can do so through transformation along these planes as illustrated in Figure 1.6. The transformed planar stresses are calculated using Equations (1.8) to (1.10):

$$\sigma_{x'} = \frac{\sigma_x + \sigma_y}{2} + \frac{\sigma_x - \sigma_y}{2} cos(2\theta) + \tau_{xy} sin(2\theta) \tag{1.8}$$

$$\sigma_{y'} = \frac{\sigma_x + \sigma_y}{2} + \frac{\sigma_x - \sigma_y}{2} cos(2\theta) - \tau_{xy} sin(2\theta) \tag{1.9}$$

$$\tau_{x'y'} = -\frac{\sigma_x - \sigma_y}{2} sin(2\theta) + \tau_{xy} cos(2\theta) \tag{1.10}$$

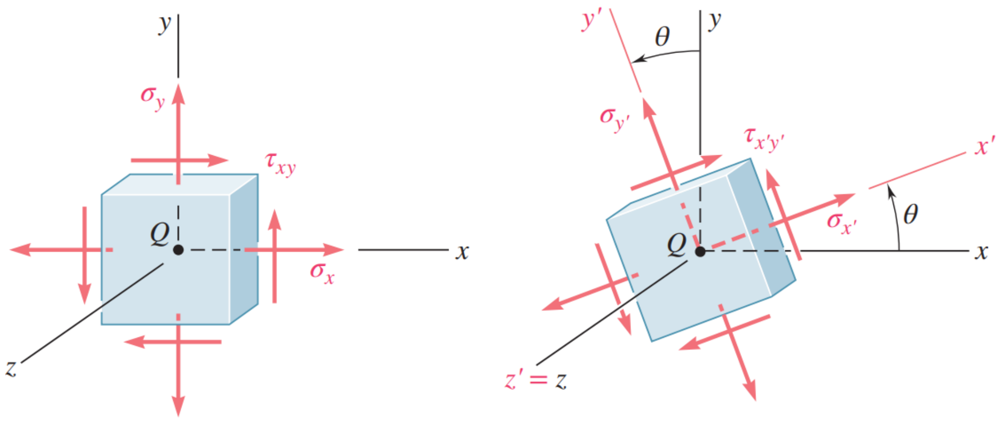


Figure 1.6: State of plane stress on an element.(a) referenced to axes $\{xyz\}$, referenced to rotated axes $\{x^{'}y^{'}z^{'}\}$ [12].

The normal and shear stresses vary during axes rotation, reaching minimum and maximum magnitudes at 90^o intervals. The *maximum* and *minimum normal* stresses, σ_{max} and σ_{min}, are called *principle stresses* with the *principle angle*, θ_p defining their positioning. These values can be located through the derivative of Equation 1.8 with respect to θ which yields Equation 1.11.

$$tan(2\theta_p) = \frac{2\tau_{xy}}{\sigma_x - \sigma_y} \tag{1.11}$$

1.2.5 Degrees of Freedom

Objects which inhabit our physical world are beholden to the laws of physics which govern it and as such are capable of moving, vibrating or deforming in complex ways. The complexity of this behaviour is assessed by identifying the number of independent motions which are required to represent all important motions of the system. These motions are referred to as *"degrees of freedom"* (DOFs) and are independent if they can still occur when all other DOFs are deliberately restrained. Within physics DOFs can refer to any number of independent parameters that define a systems configuration or state however in mechanics we tend to limit to the 6 DOFs of translation and rotation (see Table 1.1 and Figure 1.7) [13]:

1. Heaving - translation up and down
2. Strafing - translation left and right
3. Surging - translation forward and backward
4. Yawing - swivelling left and right
5. Pitching - tilting forwards and backward
6. Rolling - pivoting side to side

1.2.6 Meshing and Element Types

In order to translate from the problem statement/system definition to the FEM of the system it is necessary to discretize domains and/or geometries into constituent finite elements. Several geometric arrangements are employed as FEA elements for determined cases or circumstances and which are collectively referred to as the *element*

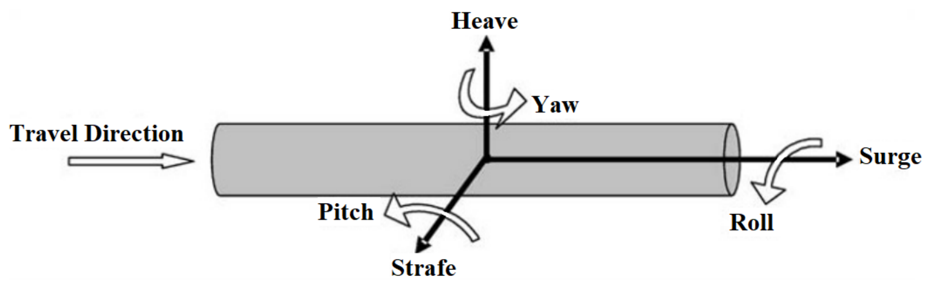


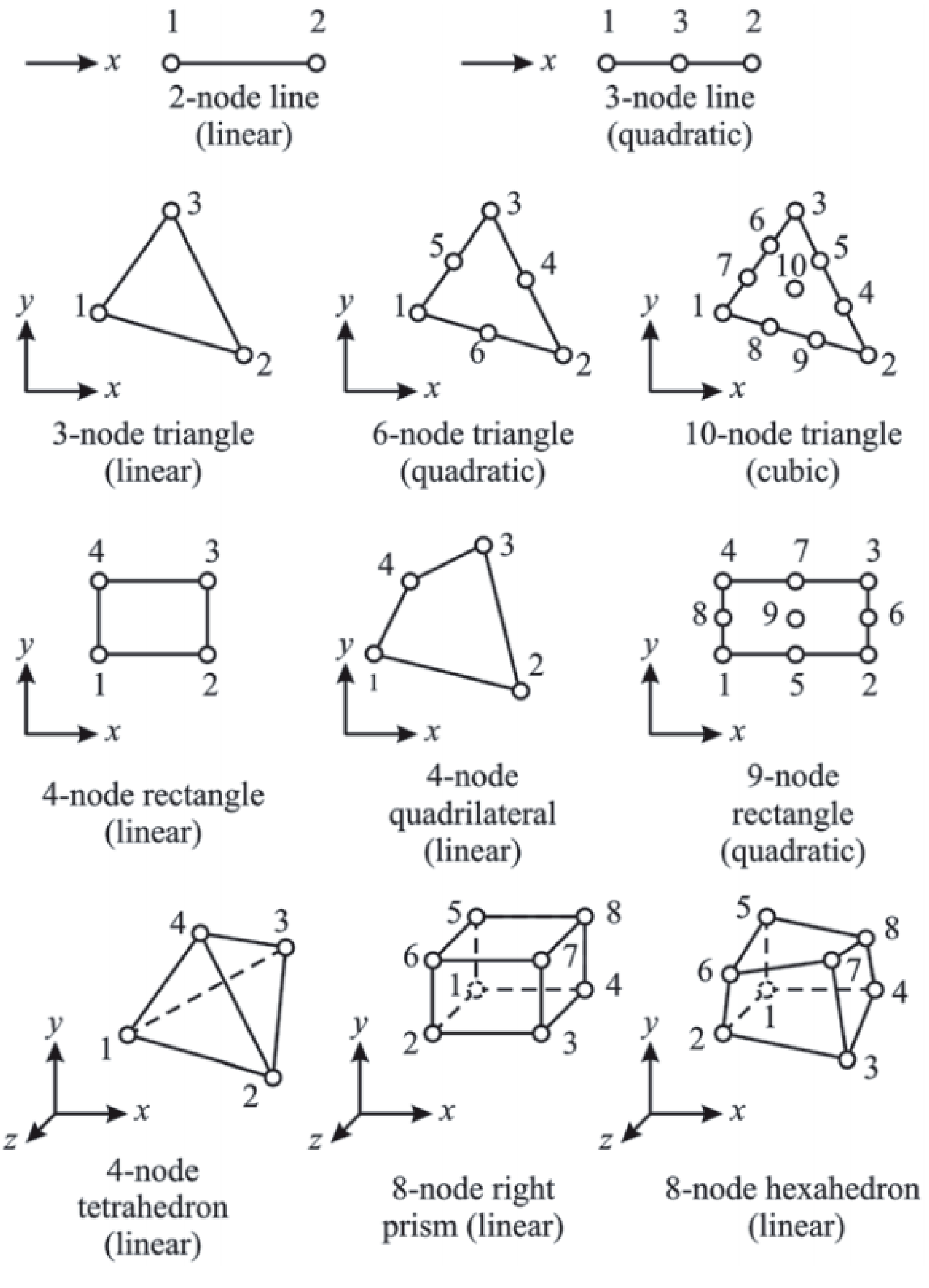
Figure 1.7: Illustration of the 6 DOFs of a unrestrained body (shown here as a cylinder) in Cartesian space [13].

Table 1.1: DOFs and force vectors in FEA for different Engineering disciplines [14]

Discipline	**DOF**	**Force Vector**
Structural/solids	Displacement	Mechanical forces
Heat conduction	Temperature	Heat flux
Acoustic fluid	Displacement potential	Particle velocity
Potential flow	Pressure	Particle velocity
General flows	Velocity	Fluxes
Electrostatics	Electric potential	Charge density
Magnetostatics	Magnetic potential	Magnetic intensity

library within software packages (See Table 1.2 for an example for common element types employed in ANSYS). Within an element each node corresponds to a coordinate in the model where DOFs are defined. For problems of structural analysis, these DOFs describe the displacement of a node as a result of loads imposed on the system. The resultant translational and rotational DOFs can be associated with the forces and moments transmitted through the nodes respectively. Correspondingly, strains can be determined from the relative motion of nodes, while the stresses are calculated from the strains and material properties [14]. Elements may have one or more discrete integration points (which may or may not correspond to nodal locations) at which the stress and strain values are calculated and monitored. Should values at nodes be sought, the results are copied or interpolated/extrapolated from the nearest integration points.

Elements can generally be classified into one-dimensional *'line'*, two-dimensional *'surface'* and three-dimensional *'solid' / 'volume'* elements. See Figure 1.8 and Table 1.2 for the elements shapes as well an excerpt of element types from the ANSYS element library.



2-node line (linear)
3-node line (quadratic)
3-node triangle (linear)
6-node triangle (quadratic)
10-node triangle (cubic)
4-node rectangle (linear)
4-node quadrilateral (linear)
9-node rectangle (quadratic)
4-node tetrahedron (linear)
8-node right prism (linear)
8-node hexahedron (linear)

Figure 1.8: The common element shapes utilized in ANSYS. From Top: Line elements; triangular elements; rectangular and quadrilateral elements; tetrahedrons, prisms and hexahedrons (for volume elements) [15].

Table 1.2: Excerpt of common ANSYS element types for structural analysis [15]

Element	**Order**	**Shape**	**Nodes**	**Physics**	**DOFs**
PLANE182	2D	Quad	4	Structural	UX, UY
PLANE183	2D	Quad	8	Structural	UX, UY
SOLID185	3D	Brick	8	Structural	UX, UY, UZ
SOLID186	3D	Brick	20	Structural	UX, UY, UZ
SOLID187	3D	Tet	10	Structural	UX, UY, UZ

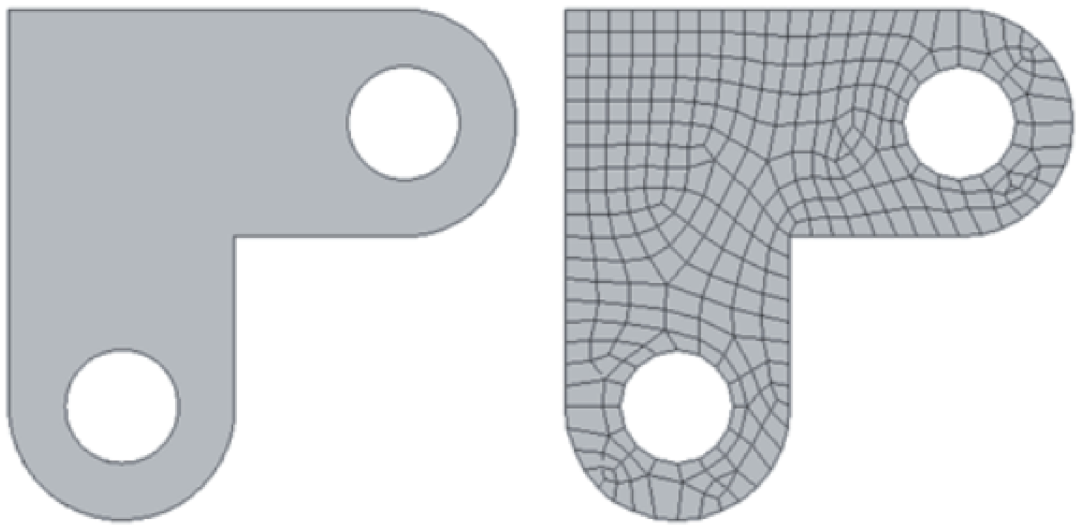


Figure 1.9: Example of meshed FE geometry. Left: Original unmeshed geometry. Right: Meshed geometry. Note the inclusion of triangular, rectangular and quadrilateral element shapes [14]

1.2.7 Mass, Damping and Stiffness matrices

A meshed FE model (as shown in Figure 1.9) includes 3 translation DOFs at each node point in the model. The FE node points are locations where the motions of the elements are calculated and reported from. The DOFs of a model can be arranged as members of an $n \times 1$ array:

$$U(t) = [u_1(t)\ u_2(t)\ u_3(t)\ u_4(t) \cdots u_n(t)]^T \qquad (n \times 1) \tag{1.12}$$

where u_i typically represent physical translations or rotations. A 'linear model' expresses forces as either explicit functions of time or linear functions of the motion variables i.e. displacements, velocities, and accelerations. Thus, equilibrium equations for the linear model have the general form of Equation 1.13

$$\begin{aligned}
&\begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{bmatrix} \begin{Bmatrix} \ddot{u}_1(t) \\ \ddot{u}_2(t) \\ \vdots \\ \ddot{u}_n(t) \end{Bmatrix} + \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} \begin{Bmatrix} \dot{u}_1(t) \\ \dot{u}_2(t) \\ \vdots \\ \dot{u}_n(t) \end{Bmatrix} \\
&+ \begin{bmatrix} k_{11} & k_{12} & \cdots & k_{1n} \\ k_{21} & k_{22} & \cdots & k_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & \cdots & k_{nn} \end{bmatrix} \begin{Bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_n(t) \end{Bmatrix} = \begin{Bmatrix} f_1(t) \\ f_2(t) \\ \vdots \\ f_n(t) \end{Bmatrix}
\end{aligned} \tag{1.13}$$

where m_i, c_i and k_i are the nodal masses, damping coefficients and stiffness ratios respectively, whilst $\ddot{u}_i$, $\dot{u}_i$ and u_i are the nodal accelerations, velocities and displacements, respectively. Equation 1.13 can be written in abbreviated form as:

$$\underline{M\ddot{U}} + \underline{C\dot{U}} + \underline{KU} = \underline{f}(t) \tag{1.14}$$

An objective of the FEM is to define the mass, damping, stiffness and force matrices $\underline{M}$, $\underline{C}$, $\underline{K}$, $\underline{f}(t)$ given an FE mesh and load description.

1.2.8 Example FEM of Linear Springs

As an illustrative example we look at the behaviour of a linear elastic spring capable of supporting only axial loading, where the elongation (or contraction) is directly proportional to the axial load applied and given by its spring stiffness k. Given that the springs supports only axial loading we select a local coordinate system along elements lengthwise x-axis as shown in Figure 1.10.

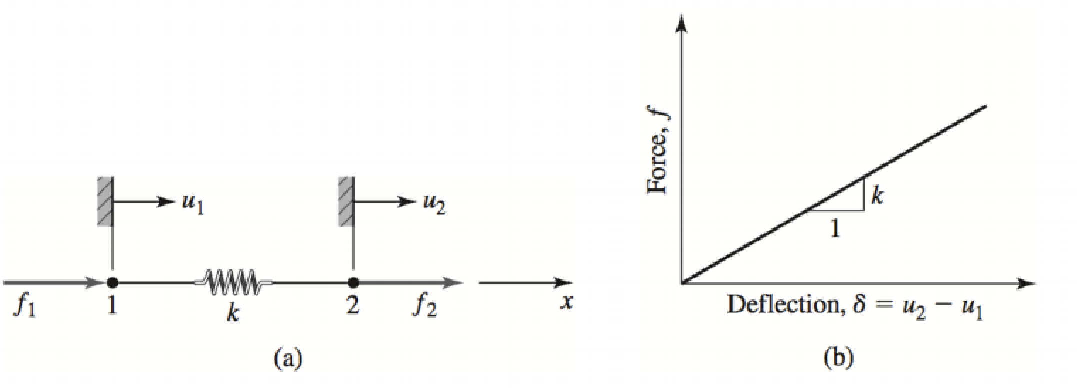


Figure 1.10: Linear spring element. (a) Linear spring element with nodes 1& 2, nodal displacements u_i, and nodal forces f_i. (b) The springs load-deflection curve representative of its spring constant, k [13].

If it is assumed that nodal displacements are both zero when the spring is undeformed, the net spring deformation is given by

$$\delta = u_2 - u_1 \tag{1.15}$$

and the resultant axial force in the spring is hence

$$f = k\delta = k(u_2 - u_1) \tag{1.16}$$

For equilibrium

$$f_1 + f_2 = 0 \text{ or } f_1 = -f_2 \tag{1.17}$$

which can be expressed in terms of applied forces using Equation 1.16 as

$$\begin{aligned} f_1 &= -k(u_2 - u_1) \\ f_2 &= k(u_2 - u_1) \end{aligned} \tag{1.18}$$

In matrix form this is expressed as

$$\begin{bmatrix} k & -k \\ -k & k \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \end{Bmatrix} \text{ or } [k_e]\{u\} = \{f\} \tag{1.19}$$

where $[k_e]$ is defined as the element stiffness matrix in the element coordinate system (or local system), $\{u\}$ is the column vector of nodal displacements and $\{f\}$ is the vector of element nodal forces [13].

Equation 1.19 shows that the element stiffness matrix for the linear spring element is a 2×2 matrix owing to the fact that the element exhibits two nodal displacements (or DOFs) which are not independent since the body (spring) is continuous and elastic. Additionally, the matrix is symmetric as a result of the symmetry of the forces (equal and opposite to ensure equilibrium).

To illustrate the influence of additional elements we look at the inclusion of a second linear elastic spring as shown in Figure 1.11.

Repeating the process of Equation 1.15 through (1.19) for both elements yields

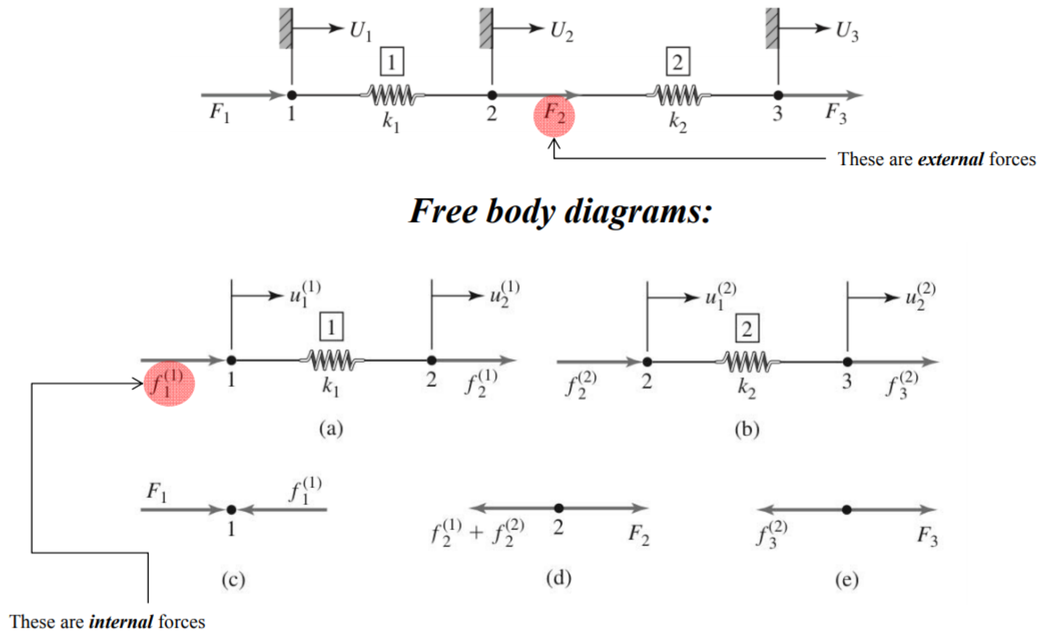


Figure 1.11: Illustration of two Linear spring element system. Free Body Diagrams: (a) spring element 1 (b) spring element 2 (c) node 1 (d) node 2 (e) node 3 [13]

Equation 1.20

$$\begin{bmatrix} k_1 & -k_1 \\ -k_1 & k_1 \end{bmatrix} \begin{Bmatrix} u_1{}^{(1)} \\ u_2{}^{(1)} \end{Bmatrix} = \begin{Bmatrix} f_1{}^{(1)} \\ f_2{}^{(1)} \end{Bmatrix}$$
$$\begin{bmatrix} k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix} \begin{Bmatrix} u_1{}^{(2)} \\ u_2{}^{(2)} \end{Bmatrix} = \begin{Bmatrix} f_2{}^{(2)} \\ f_3{}^{(2)} \end{Bmatrix} \tag{1.20}$$

We can relate the element displacements to the system displacements through compatibility conditions as in Equation 1.21

$$u_1{}^{(1)} = U_1 \qquad u_2{}^{(1)} = U_2 \qquad u_1{}^{(2)} = U_2 \qquad u_2{}^{(2)} = U_3 \tag{1.21}$$

Therefore Equation 1.20 becomes:

$$\begin{aligned}
\begin{bmatrix} k_1 & -k_1 \\ -k_1 & k_1 \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \end{Bmatrix} &= \begin{Bmatrix} f_1{}^{(1)} \\ f_2{}^{(1)} \end{Bmatrix} \\
\begin{bmatrix} k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix} \begin{Bmatrix} U_2 \\ U_3 \end{Bmatrix} &= \begin{Bmatrix} f_2{}^{(2)} \\ f_3{}^{(2)} \end{Bmatrix}
\end{aligned} \tag{1.22}$$

Here, we use the notation $f_i{}^{(j)}$ to represent the force exerted on element j at node i. Expanding each equation in Equation 1.22:

$$\begin{aligned}
\begin{bmatrix} k_1 & -k_1 & 0 \\ -k_1 & k_1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ 0 \end{Bmatrix} &= \begin{Bmatrix} f_1{}^{(1)} \\ f_2{}^{(1)} \\ 0 \end{Bmatrix} \\
\begin{bmatrix} k_2 & -k_2 & 0 \\ -k_2 & k_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} U_2 \\ U_3 \\ 0 \end{Bmatrix} &= \begin{Bmatrix} f_2{}^{(2)} \\ f_3{}^{(2)} \\ 0 \end{Bmatrix}
\end{aligned} \tag{1.23}$$

Summing member by member:

$$\begin{bmatrix} k_1 & -k_1 & 0 \\ -k_1 & k_1 & -k_2 \\ 0-k_2 & k_2 & \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} f_1{}^{(1)} \\ f_2{}^{(1)} + f_2{}^{(2)} \\ f_3{}^{(2)} \end{Bmatrix} \tag{1.24}$$

Referring to the free-body diagrams in Figure 1.11 of each of the nodes:

$$f_1{}^{(1)} = F_1 \qquad f_2{}^{(1)} + f_2{}^{(2)} = F_2 \qquad f_3{}^{(2)} = F_3 \tag{1.25}$$

gives the final form:

$$\begin{bmatrix} k_1 & -k_1 & 0 \\ -k_1 & k_1 & -k_2 \\ 0 & -k_2 & k_2 \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \\ F_3 \end{Bmatrix} \tag{1.26}$$

As with Equation 1.19, Equation 1.26 can be written in abbreviated form as

$$[K]\{U\} = \{F\} \tag{1.27}$$

where,

- $[K]$ - Global stiffness matrix
- $\{U\}$ - Vector of displacements
- $\{F\}$ - Vector of applied forces

The solution of Equation 1.27 may be found mathematically using a matrix inversion method to obtain:

$$[K]\{U\} = \{F\} \quad \Rightarrow \quad \{U\} = [K]^{-1}\{F\} \tag{1.28}$$

Here $[K]^{-1}$ is the *inverse stiffness matrix.* Whilst this could certainly be achieved manually for the simplistic example here, real world problems may have upwards of thousands or even millions of more sophisticated elements. Hence the reliance on computerized solutions today.

1.3 Advances in Real-time Structural Analysis

With FEA being a numerical method of problem solving it relies extensively on the usage of modern computing hardware to carry out the calculations and as such is typically performed “offline” which is to say at stages preceding usage; be it for analysing or simulating the iterative changes to a design or process; or post implementation such as investigating the root causes of a problem or failure. Typically, FEA is carried out in Three stages [8]:

1. A model is prepared in the *pre-processor* during which inputs such as geometries, material properties, loads and boundary conditions are defined.
2. A *solver* calculates a solution to the problem with respect to the aforementioned inputs.
3. Finally, the *post-processor* offers a means for the analyst to observe and assess the solution output.

The complexity of the problem may lead to the disconnect between parsing inputs to the solver and receiving outputs being a substantial amount of time. Whilst this is the generally accepted norm, there are situations which arise in which it would be preferable to be able to receive results in a timelier manner. These situations broadly include categorical time critical problems such as:

- Predictive control
- Interactive virtual environments
- Health monitoring or medical procedures

- Design / Process optimization

As demand has increased, several strategies have been employed in attempts to move towards establishing "online" real-time (or near real-time) approaches and methodologies capable of performing FEA tasks and models in situ. These strategies can be grouped into 3 primary distinctions [16]:

1. Use of powerful computational hardware, including altering the methods in how the components are used.
2. Code optimization, faster algorithms and parallelization of computation.
3. Development of alternate formalisms and formulations for structural analysis.

The first can be attributed to the increases in raw processing power which follows a trend commonly referred to as "Moore's Law" which states that the number of transistors on an integrated circuit doubles roughly every two years [17] (See Figure 1.12). Furthermore, purpose built discrete components capable of performing computationally intensive operations are increasingly commonplace with the most widely adopted being Graphic Processing Units (GPUs) used as co-processors alongside conventional CPUs [18].

The second, is closely tied to the first in that developments on the levels of software and programming languages as well as the ability to distribute calculations to networked computing devices allow for computational loads to be shared and executed in an increasingly rapid and efficient manner.

The third, and most relevant to our investigation, pertains to the advances in modelling methodologies and procedures. A non-exhaustive summary of these methodologies is discussed:

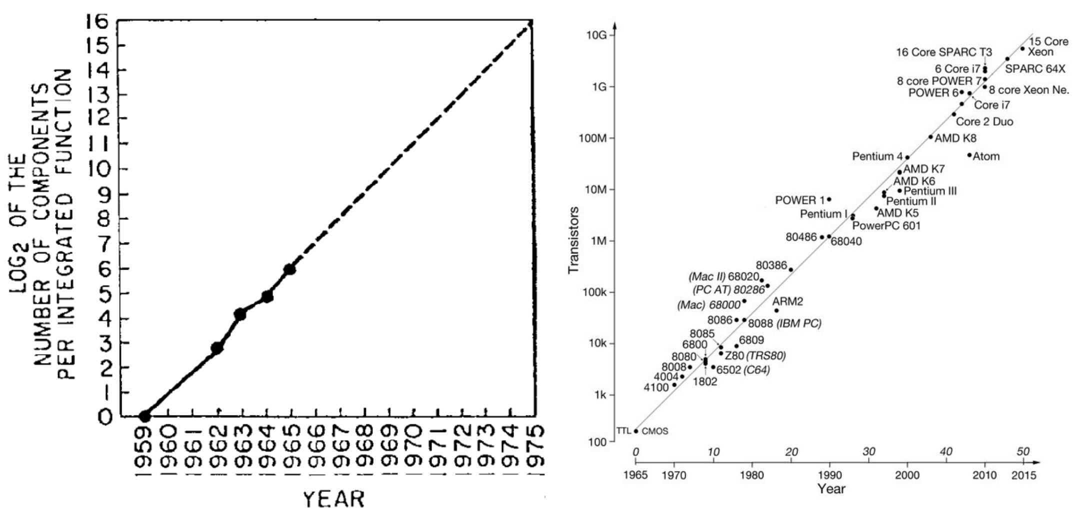


Figure 1.12: Extrapolated projection for number of components per integrated function vs year commonly referred to as "Moore's Law". Left: Moore's 1965 original [17]. Right: Updated plot as of 2015 for consumer CPUs [19]

1.3.1 Full Extent Finite Element Models

If there were no limits on computational power, we would undeniably choose to execute as comprehensive and theoretically rigorous a model as possible so as to arrive at a solution which most accurately describes the real-world counterpart. However, even systems of moderate complexity and number of DOFs result in systems of equations which are too prohibitively large to execute in real-time. Hence several attempts have been made to alter the way the problems are formulated and solved with an acceptable loss of accuracy in order to meet the objective of "real-time" simulation. It is not uncommon to limit models to the linear regimes in situations in which this assumption not only significantly reduces computational load but also falls within a level of accuracy which adequately approximates the ground truth. Furthermore since a large portion of the computation time is dedicated to solving the inverse of the stiffness matrix, $[K]^{-1}$ (as discussed in subsection 1.2.7), it is advantageous to precalculate the inverse matrices where possible. This was the approach carried out by Nikitin et al [20] for their Virtual Reality (VR) simulation of elastic bodies. Similarly Huang et al [21] coupled wireless sensors distributed within an environment to relay loads for which solutions were generated by an FEA in or order to overlay stress predictions using Augmented Reality (AR) in real-time (see Figure 1.13(a)). Their assumption of linear and semi-static conditions, with non-variable geometry, allowed their approach to make use of precalculated inverse stiffness matrices. This precomputation step greatly reduces the computational loads during the real-time phase of execution allowing for frames rates of between 30 down to 5 frames per second (fps) for models comprising between 500 to 4000 nodes (see Figure 1.13(b)) .

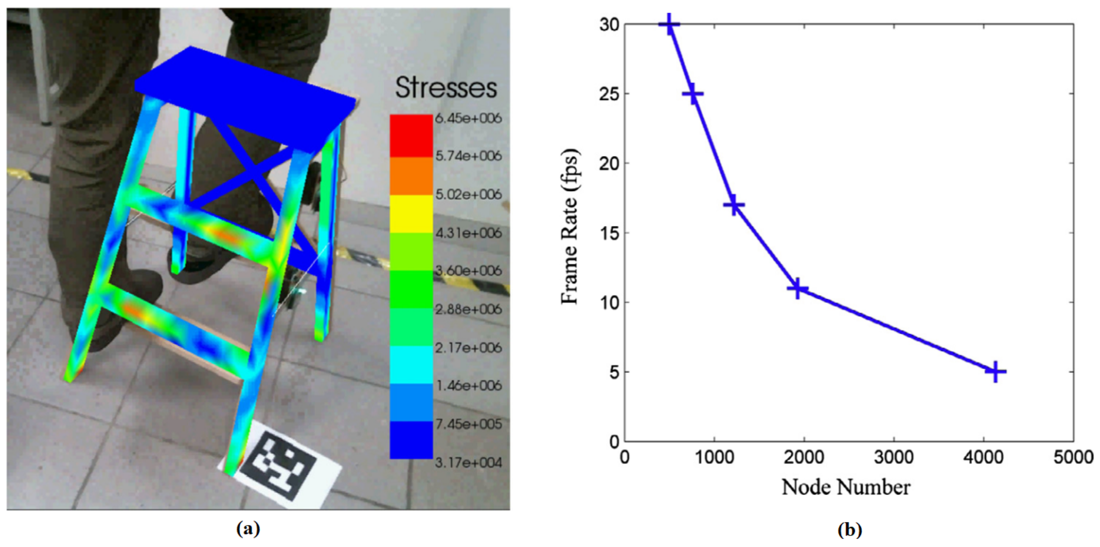


Figure 1.13: Real-time AR projection of a step ladder's stresses. (a) Real-time AR output. (b) Frame rate vs Number of Nodes "Node Number" [21]

In a more advanced approach Cerracchio et al [22] used an inverse Finite Element Method (iFEM) to reconstruct the deformed structural shape of a composite stiffened panel from in situ strain measurements. The reconstruction of the deformed shape from strain data enables the full-field reconstruction of structural strains and stresses and is known as shape sensing. It allows for the application of failure criteria for structural health assessment. To achieve this, a high-fidelity NASTRAN model in the linear elastic regimes was first modelled as the basis for the strain results which were then utilized against the strain measurements to inversely obtain the deformed shape. The iFEM methodology has the additional benefit of not requiring applied loading, material constants, or inertial and damping characteristics to arrive at a solution.

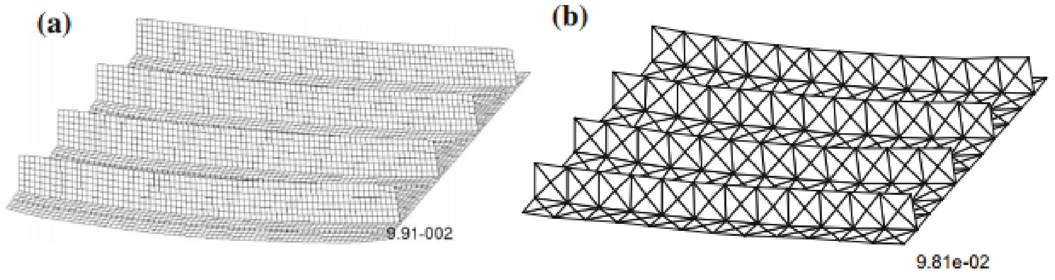


Figure 1.14: Deformed shape of a composite panel subjected to Thermal load: (a) Nastran evaluated deflection and (b) iFEM prediction [22]

There have been numerous studies which look to emulate the characteristics of soft biological tissues such as skin [23; 24], muscle [25; 26], internal organs [27; 28] and even brain tissue [29]. Due to the complexities unique to each of the biological materials' structures and behaviours however, most models are required to be non-linear or include hyper-elasticity. This unfortunately prohibits the models from being run in time frames nearing anything that can be considered real-time. For the purposes of virtual operations and training procedures where real-time execution is a requirement, the linear elastic assumption once again dominates but is augmented in several instances to improve accuracy in the regions of primary interest. One such example by Picinbono et al [30] incorporates a linear-elastic FE model alongside transversely isotropic properties with an additional external membrane to better represent the tougher outer surface skin of internal organs in order to provide realistic haptic feedback to the surgeon.

Another approach denoted as a 'hybrid condensed FE model' by Wu et al [31] proposes partitioning the model into two separate regions, namely a region being interacted with and the remainder of the model. A complex non-linear FE model able to deal with topological change is used to model a small-scale operational region, whilst a linear and topology-fixed FE model is used to model the large-scale non-operational region. Doing so whilst also allowing for the extent of the regions to be changed dynamically allows for the resultant virtual surgery training system to be employed on contemporary hardware.

1.3.2 Mass-Spring Systems

In instances where the predominant physical behaviour is dependent on the membrane stiffness as opposed to the bending or shear stiffness, mass-spring approximations may serve as suitable simplifications of the system. This methodology has been used in order to achieve levels of accuracy which simulate plausible physical behaviour in softer materials such as cloth simulations in entertainment [32] or soft tissue deformations in the medical and biological fields and in which exact numerical solutions are not a requirement. For instance, Wang et al [33] developed a mass-spring model for use in VR vascular surgery which closely mimics the physical behaviour of thin walled vascular tissue. The approach amounts to discretizing the component(s) into point masses, representing the internal forces between points as massless elastic springs and computing the positions and velocities at discrete timesteps. Additional springs may be incorporated to account for shear and bending stresses should it be desired. The relative simplicity of the resultant stiffness matrix allows for higher computational speeds at the loss of accuracy.

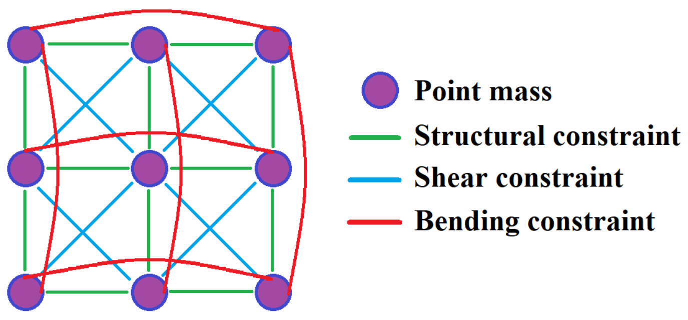


Figure 1.15: Illustrative representation of a Mass-spring structure. [33]

1.3.3 Model Order Reduction Techniques

FE models by both name and nature are characterized by the discretization of the system/geometry under investigation. This discretization leads to a system of equations of high-dimensionality, the solution of which constitutes most of the time and resource consuming processes of the FEA i.e. execution of the *Solver*. The ability to reduce the dimensions of the system and in doing so the ultimate processing cost whilst still retaining a desired level of accuracy is the underlying motivation to seek methods of *Model Order Reduction* (MOR).

One such technique, Mode Shape Extraction is based on modal vectors and coordinates. At its basis, several prior computations are performed during which the orthogonal mode shapes are determined. These mode shapes are thereafter used as the DOF in determining the resultant deformational component of the analysis. The advantage of this approach is that it is possible to reduce a system with an immense number of DOFs (in the order of millions) down to a few dozen orthogonal mode shapes. Furthermore, the use of orthogonal mode shapes also assists in the reduction process as they are decoupled from one another. This approach as a general idea is commonly utilized within FEA to determine structural transient responses within the linear domain. Recent research by Ferhatoglu et al [34] have developed a promising technique for superposition in the non-linear domain they refer to as Response Dependent Nonlinear Mode (RDNM) which appears to produce results with accuracy comparable to a non-reduced Full Solution method. The RDNM method makes use of a calculated complex stiffness matrix called the 'nonlinearity matrix' which for given displacement amplitude levels represents the corresponding nonlinearity. The RDNMs of the system are derived through solution of the Eigen Value Problem (EVP) which corresponds to the pattern and displacement level for which the matrix was determined. As changes are made to the excitation frequency, the pattern and correspondingly the steady state response level changes, in turn resulting in new and different equivalent stiffness matrices and therefore different EVPs. Essentially the method uses variable linear stiffness matrices which can be superimposed.

Another method is the usage of Component Mode Synthesis (CMS) techniques wherein the DOFs for an elastic structure are divided between interior or boundary DOFs. The boundary DOFs correspond to the nodes retained in the simulation model for the purpose of kinematic and dynamic boundary conditions. Thereafter the modes for the constraint and the normal modes for the fixed boundary are determined. For the constraint, the modes are static with their number corresponding to the number of DOFs of boundary. To obtain the constraint modes, a unit displacement is applied to one of the boundaries DOFs while all remaining DOFs at the boundary are fixed. To obtain the fixed-boundary normal modes conventional modal analysis is conducted

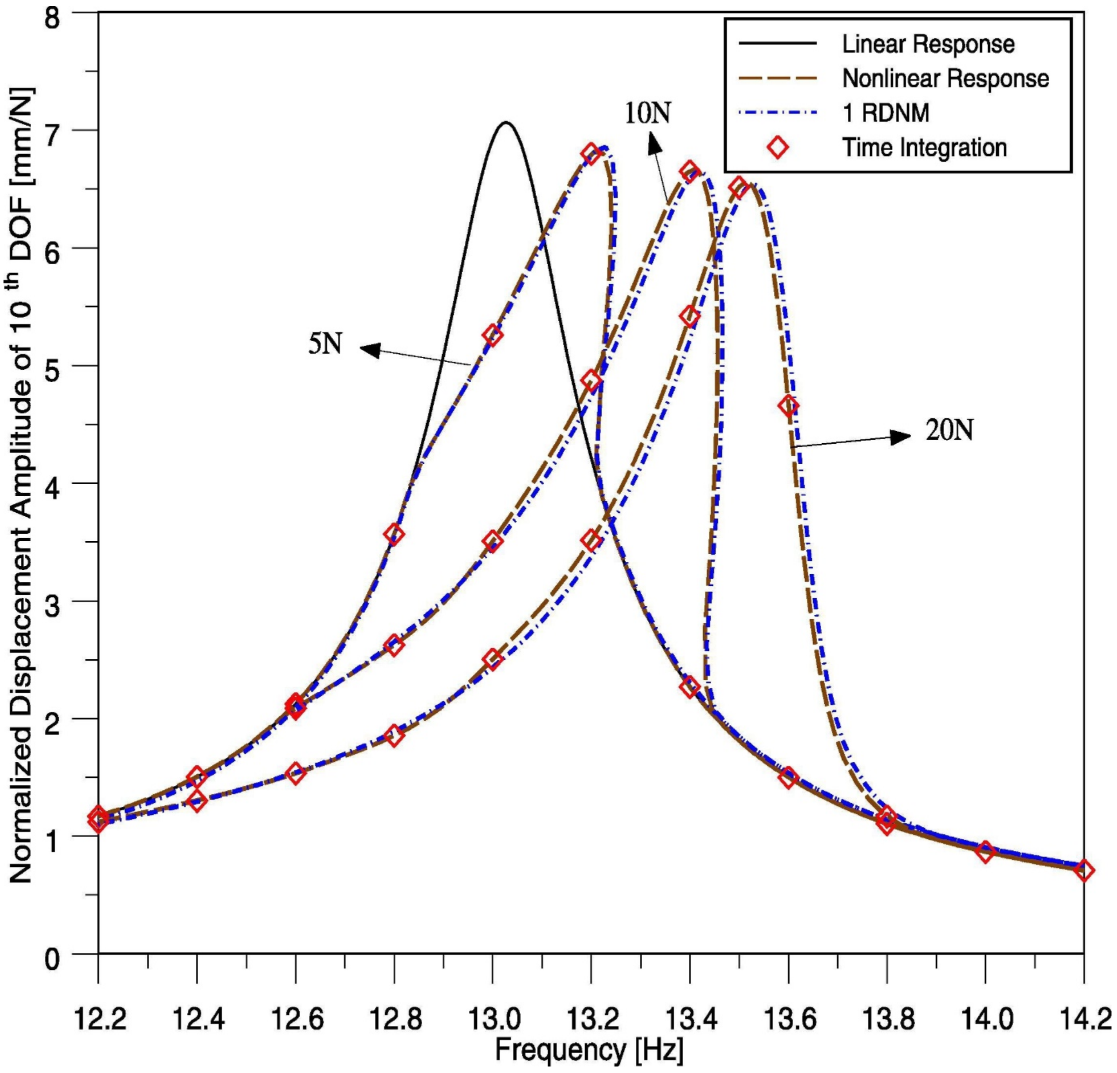


Figure 1.16: Comparative results for Displacement of the tenth DOF for Linear Response vs Nonlinear and RDNM models [34]

where all boundary DOFs are fixed. This combination of modes does not make an orthogonal set of modes. The obtained set of modes is normalized prior to simulation in order to keep the numerical benefits of a decoupled system, doing so however causes the modes to lose their interpretability much like the creation of orthogonal axes in the statistical technique Principal Component Analysis (PCA)[1]. With either of these reduction techniques the modes are determined from the undeformed initial conditions

[1]Principal component analysis is a procedure that convert a set of possibly correlated variables into a set of linearly uncorrelated variables and often employed in statistical methods for the purposes of dimensionality reduction. Technique developed by Karl Pearson as early as 1901 [35]. A tutorial on its usage is provided in [36]

and are thus intended for linear analysis and are correspondingly applicable for small deformations.

1.3.4 Training of Neural Networks based on FEA data

A relatively new and exciting field of research with respect to application in the physical sciences is the rapidly advancing field of *Artificial Intelligence* (AI) and its usage of *Artificial Neural Networks* (ANN) within *Machine Learning* (ML) and D*eep Learning* (DL). This is in no small part due to their seemingly endless application and growing usage case potentials.

Within the realm of FEA itself, neural networks have been used as a form of indirect Model Order Reduction technique looking to generalise volumes of data into condensed representations of the data domain of all potential data configurations (for the problem or system under investigation that is). This is achieved through purpose-built architectures which are trained on said data with the goal of producing a model which ideally inherently "understands" the system on which it was trained. Should this be achieved, the trained model is expected to be capable of inferring predictions at rates considerably quicker and using less computational resources than through conventional means. A caveat of the training/learning process however is that the dataset required typically still needs to be initially obtained through conventional means. When pertaining to structural models this necessitates conventional FEA means.

Hambli et al [37] utilized an architecturally simple neural network in order to achieve the real-time prediction of tennis ball and racket deformations for usage in Virtual Reality (VR) simulations. A dataset of randomly generated parameters such as velocity, angle of impact and impact location were used as input whilst deformations (of ball and strings), impact forces and deflection angles formed the output dataset (see Figure 1.17). For the factors identified, only discrete values were retained for generating the datasets. five values for ball velocity, four values for impact angle and thirteen impact zones. The combination of all factors produced a two hundred and sixty ($13\times5\times4$) design of experiments (DoE) combinations for the FE simulation to study the effect of each variable and the interaction between them. The trained model can predict the resulting (1 of 260 discrete) dynamic interactions with haptic feedback to the user in real-time.

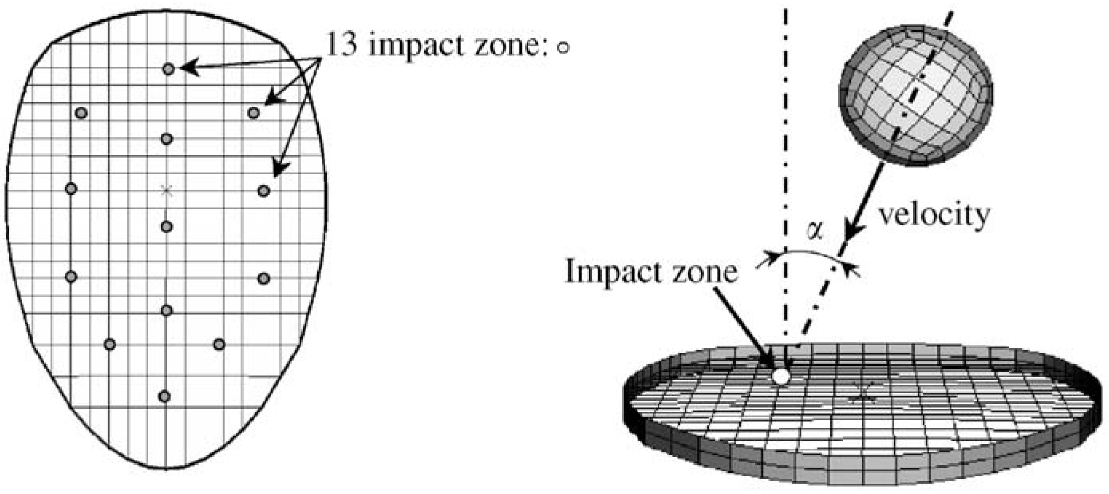


Figure 1.17: VR Tennis Racket parameters. Left: Tennis racket impact zones and Right: output responses for real-time model developed by Hambli et al [37]

Ordaz-Hernandez et al [38] developed a model capable of predicting the non-linear responses of a cantilever beam via a multi-layer feed-forward neural network referred to as a ‘universal approximator’ multi-layer perceptron (see Figure 1.18). Comparing their results to two FE models; one linear and fast yet less accurate, one non-linear but slower and more accurate; they found that their neural network performed with the speed of the linear FEA but accuracy comparable to the non-linear model within the domain of training inputs.

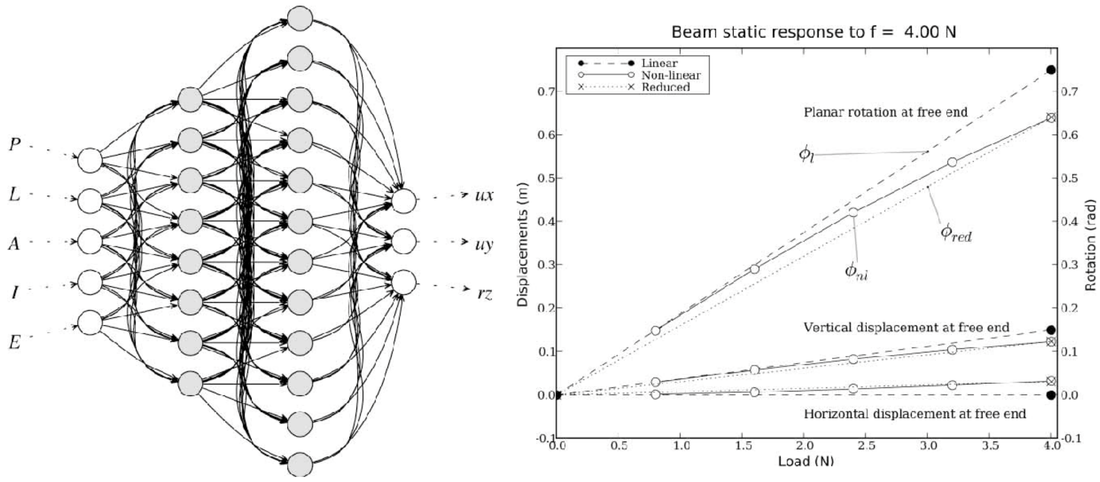


Figure 1.18: Left: Architecture of "universal approximator" multilayer perceptron wherein variables *P*, *L*, *A*, *I*, *E* fed in and the predicted responses of *ux*, *uy*, *uz* for the cantilever beam free end are returned. Right: Great displacements domain case. The nonlinear model requires five steps to complete its response. The linear and the reduced model require only one step. Ordaz-Hernandez et al [38]

Morooka et al [39] developed a system reliant on neural networks to determine the deformations of a liver model consisting of 729 vertices and 512 cubic elements. Their 'neuroFEM' model was able to return results which were in close agreement with those provided by conventional non-linear FEA methods (see Figure 1.19). Using a modest computer (Pentium4 2.8 GHz with 1Gb RAM) the computational time of the neuroFEM is also stated to be roughly 1,000 times faster than that of the non-linear FEM.

1.4 Problem Statement

With the advances and increasing number of techniques in AI, more specifically ML and DL, it is likely that there exist several novel, as of yet untested, methodologies which could be strong candidates for application in FEA with the potential ability

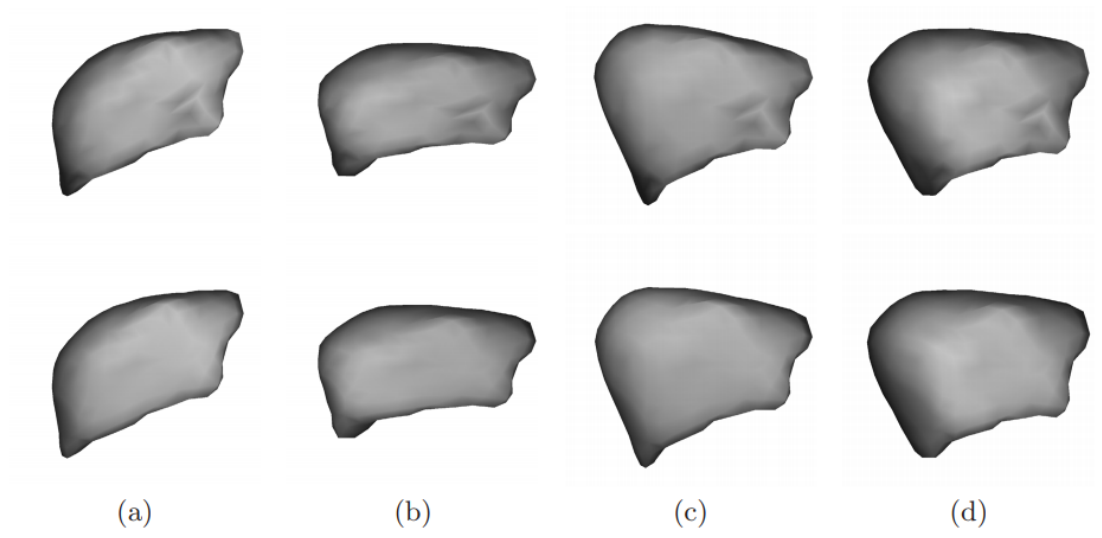


Figure 1.19: Comparison of methods between Morooka et al. [39] (top row) and non-linear FEM (bottom row). (a) and (b) generated from two forces in training dataset whilst (c) and (d) are from arbitrary forces applied to original model.

to produce real or near-real-time results. One such candidate is the Image-to-Image translation capable Generative Adversarial Network (GAN). Whilst GANs have been predominantly used for abstract or even artistic use cases, I however propose that their powerful image generating capabilities have the potential ability to infer the physical force-deflection-stress relationships found in static FEM. Furthermore, they may be capable of doing so at resolutions and rates at or in excess of those discussed in subsection 1.3.4.

1.5 Objectives

The aim of this thesis is to **develop and utilize purpose-built ML models to learn the inherent physical behaviour of a domain of commonplace FEA problems in order to produce user requested predictions in real-time**. We will use the case of isotropic rectangular platework of varying geometric and material properties subjected to several varying loads.

To fulfil this aim, several objectives need to be achieved:

1. Create a dataset(s) containing the inputs and responses of the FEA "problem set"
2. Implement a working ANN capable of training on the Dataset(s)
3. Generate trained model(s) using the ANN
4. Verify the validity of the model(s)
5. Implement a Graphic User Interface (GUI) for prediction of user inputs.

1.6 Structure of the Thesis

chapter 1 provides a brief background of FEA, the fundamentals of FEA, the significance of the issue, the problem statement, objectives and structure of the study.
chapter 2 provides information regarding contemporary developments in AI, The working principles of AI and its constituent sub sections as well as breakdown of the conditional GAN “Pix2Pix” on which this work is primarily based.
chapter 3 covers the methodology and process of implementation of dataset creation, neural network creation, training, result verification and GUI implementation.
chapter 4 discusses the results of the model training. The comparison between a sequentially stepped stress prediction (ie force to deflection to stress) and directly from force to stress are analysed as well as a discussion on the GAN as a feasible alternative to conventional FEA.
chapter 5 presents the conclusions drawn as well as a discussion on potential future work.

Thereafter ancillary media such as the **Appedices** and **References** are provided.

2 Literature Review

This chapter serves to cover the theoretical and operating principles behind AI, Machine Learning and related sub-topics. Readers familiar with these topics may advance to section 2.7 for discussions of GANs, cGANs and their applicability to FEA for the purposes of this thesis.

2.1 Artificial Intelligence vs Machine Learning vs Deep Learning

Artificial Intelligence (AI), Machine Learning (ML) and Deep Learning (DL) have increasingly become more common and popular buzzwords, often used interchangeably. When speaking of AI the implication is that systems, be they physical machines or software, are capable of performing one or more of the following tasks [40]:

- understanding human behaviour or language,
- performing mechanical tasks involving complex manoeuvring,
- solving complex computer-based problems often involving large data in a very short time.

ML is a subset of the field of AI wherein machines, or more precisely algorithmic architectures acquire knowledge automatically from the data presented to them. In this way the internal algorithms are said to "learn" the required relationships, features, functions or actions required in order to fulfil the task for which it was created. This learning can be achieved in a number of ways but broadly speaking is carried out through the ML architecture systematically analysing data samples and adjusting its internal parameters until such time that a model of suitable accuracy is achieved.

DL in turn is a subset of ML which relies on networks that make extensive usage of hidden layers. This hidden aspect of the architecture is what gives DL its “depth”. Furthermore, some sources distinguish ML and DL by the level of feature extraction or data pre-processing required by the user, with ML generally requiring features to be defined up front whilst DL does not [41]. The relationship between these classes and subclasses is illustrated in Figure 2.1.

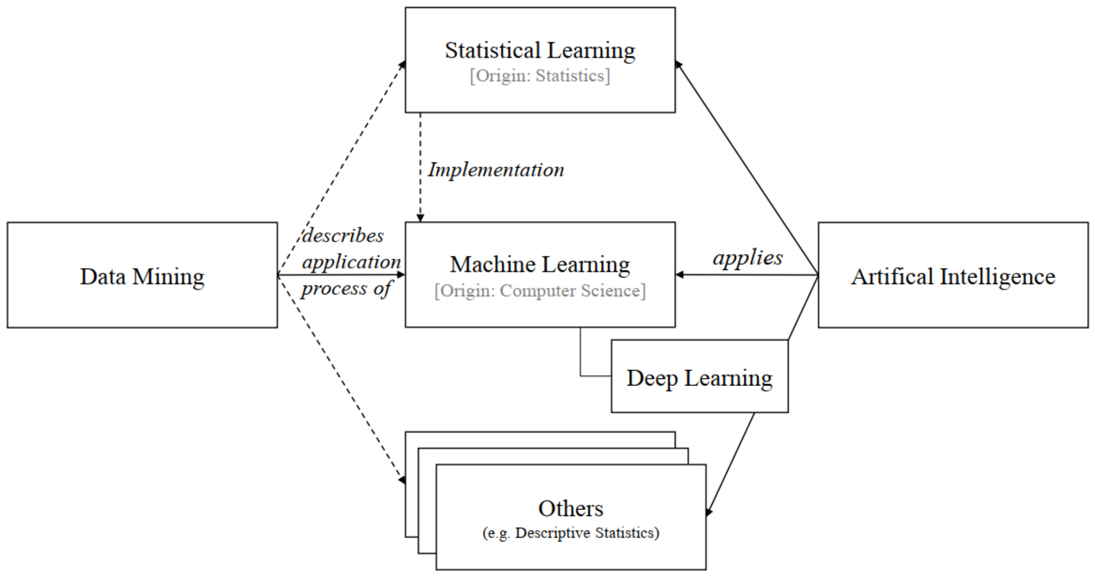


Figure 2.1: Relationship between Artificial Intelligence, Machine Learning and Deep Learning [41].

2.2 Types of Machine Learning

Machine Learning can be divided into several groupings according to the way in which their learning feedback is approached, namely Supervised, Semi-supervised, Unsupervised and Reinforcement Learning [42].

2.2.1 Supervised Learning

In Supervised learning the dataset is comprised of collections of labelled examples $\{(x_i, y_i)\}_{N_i=1}$ wherein each element x_i contains a vector of features $[x^j, \cdots, x^D]^T$ which describe it. These features could be of any type or class from descriptive text to discrete or continuous numeric values.

For example, a dataset of drug trial patients could have features such as age (in discrete years), weight (continuous kg's), gender (discrete options of M/F), dosage (in mg/number of pills) and "received placebo" (true/false). For each element of the dataset, their features will be structurally the same with each elements feature $x_i^{(j)}$ corresponding to the same feature and type of information as the other elements $x_N^{(j)}$.

The Label, y_i can be an element of a finite set of classes $\{1, 2, \cdots, C\}$ or a real number, a vector or a multidimensional matrix etc. This could for instance be whether the patients of the aforementioned dataset where (cured, had no effect, condition worsened, died).

The goal of supervised learning is to attempt to create a model capable of correlating and predicting the label of an input from its feature set. A commonly taught and cited example of which is the MNIST dataset which is used to train networks in recognizing numeric characters from handwritten inputs [43]. Figure 2.2 shows extract of handwrittern numbers from the MNIST dataset.

2.2.2 Unsupervised Learning

In Unsupervised learning the dataset may possess similar feature vectors as those in Supervised learning. However, the primary difference is that the labels that categorise the elements of the dataset are not present $\{x_i\}_{N_i=1}$. As a result, Unsupervised learning

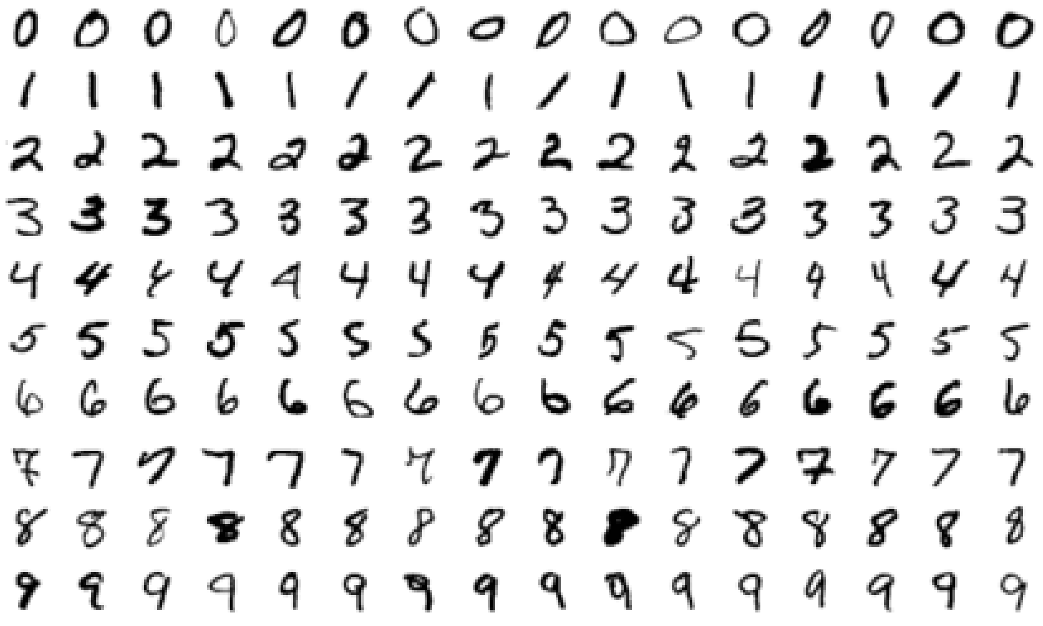


Figure 2.2: Excerpt from the 'MNIST' handwritten numerical character dataset [43]

typically aims to group or cluster datasets along the features or could be used for the purposes of dimensionality reduction much like is done within statistical methods such as PCA [44].

2.2.3 Semi-supervised Learning

In semi-supervised learning, the dataset is comprised of both labelled and unlabelled elements. The number of labelled elements is usually outweighed by the number of those unlabelled. The objective of the labelled elements is to assist in allowing the model to self-categorise/infer labels for the remainder from the feature vectors and in doing so, typically, arrive at a better model than that derived from a purely unlabelled dataset.

It should be noted however that having labels does not guarantee better models as it is possible for unsupervised models to infer abstract relationships (from the perspective of human interpretation) which might otherwise not be apparent and which drive the model to learn more efficiently than those of human labelled datasets [45].

2.2.4 Reinforcement Learning

In Reinforcement learning the model is developed as an ‘agent’ which inhabits an environment and is given sensors as a means to perceive it. Its perception is a vector of features. Furthermore, it is given the ability to perform actions within the framework of its environment. The outcome of its actions results in rewards (or punishments) which serve to drive change in the model. The end goal of the Reinforcement Learning model is to derive a set of actions or a “policy” which maximizes the expected rewards. Reinforcement learning is often utilized in situations where problems require sequential action and for this reason are frequently used for the artificial intelligences of a game-playing nature [46], stock trading and finance [47; 48] or robotics ([49] - see Figure 2.3).

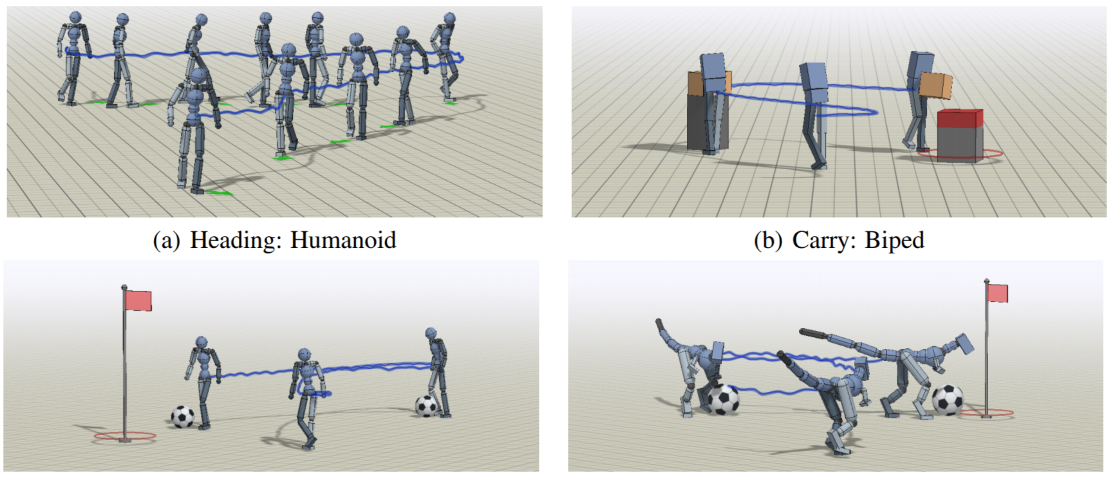

(a) Heading: Humanoid
(b) Carry: Biped
(c) Dribble: Humanoid
(d) Dribble: T-Rex

Figure 2.3: Reinforcement learnt model able to transfer the locomotion to mannequins of different configurations performing varying tasks such as (a) following a path/heading (b) carrying objects (c) dribbling a ball and (d) transferring leant skills between skeletal configurations [49]

Summarily, Machine Learning is a broad field which is continuously growing to include additional capablilities and tasks, many of which were considered beyond the capablilites of machines not too long ago. Figure 2.4 serves as a non exhaustive summation of contemporary ML capabilities [50].

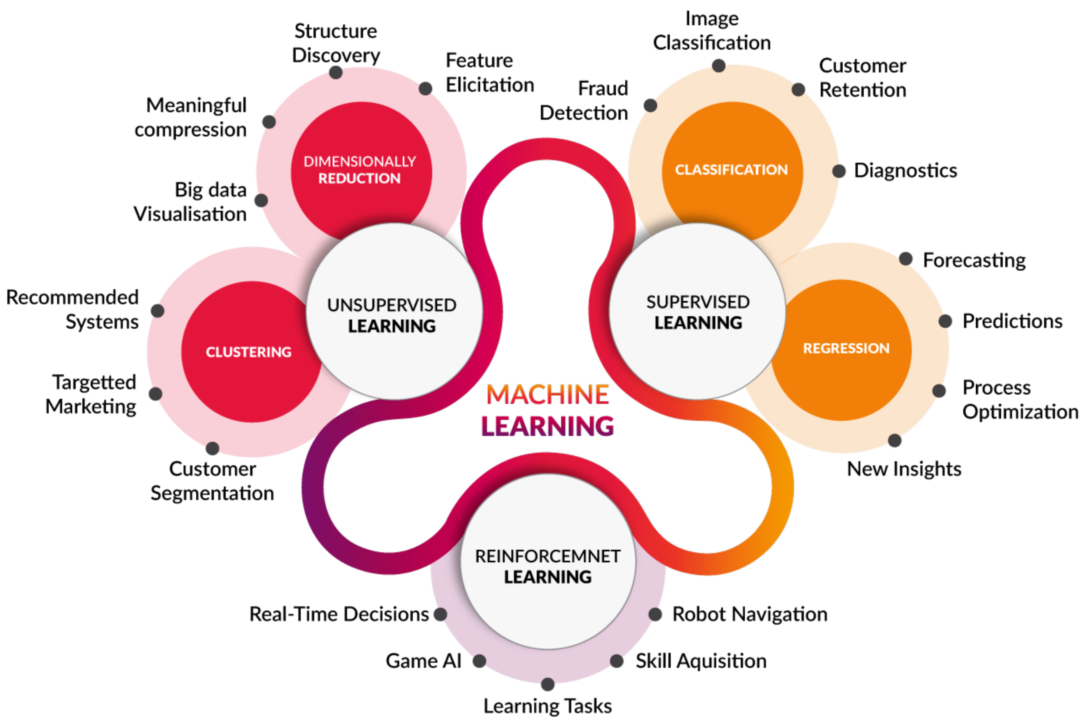


Figure 2.4: Commonly performed Machine learning tasks and their grouping [50]

2.3 Artificial Neural Networks

The working principle behind ML relies on a network of interconnected nodes, commonly referred to as *‘neurons’* or *‘perceptrons’*. These neurons are arranged progressively from input layer to output layer in what is referred to as an *Artificial Neural Network* (ANN). The specific layout and degree to which neurons and layers are interconnected is known as the *“connectivity pattern”* or *“architecture”* of the network. During each cycle of learning or ‘training’, each neuron in the input layer receives a single input (variable, feature etc) from the feature vector of a sample of the dataset being ‘taught’ and through connections to subsequent neurons in subsequent layers, relay information deeper through the network until arriving at the output layer. During transmission itself, the *weights*, w of the neural connections and *biases*, b of the internal neurons affect to what extent data is transmitted to the next connected neuron in the network. At the neurons themselves these weighted inputs are linearly aggregated along with that neurons bias. Thereafter various *‘activation functions’* are applied to the aggregate value of the neuron, z resulting in the *’activation’* value, a, of that neuron. The neurons activation value is transmitted as inputs to the next connected neuron. This occurs recursively for each subsequent neuron in each subsequent layer. This is represented diagrammatically in figure 2.5 and a summation of commonly used activation functions is presented in Table 2.1.

The output for each neuron can be represented mathematically as:

$$a_{out} = g(z) \tag{2.1}$$

where

$$z = \sum_{i=1}^{N} (a_i \times w_i) + b \tag{2.2}$$

and,

$a_1 ... a_n$	- neuron activation values.
$w_1 \ldots w_n$	- neural connection weights.
b	- neural bias.
z	- Aggregated sum of all inputs to neuron.
$g(z)$	- activation function.

Furthermore, should a loss function form part of the network such as in cases

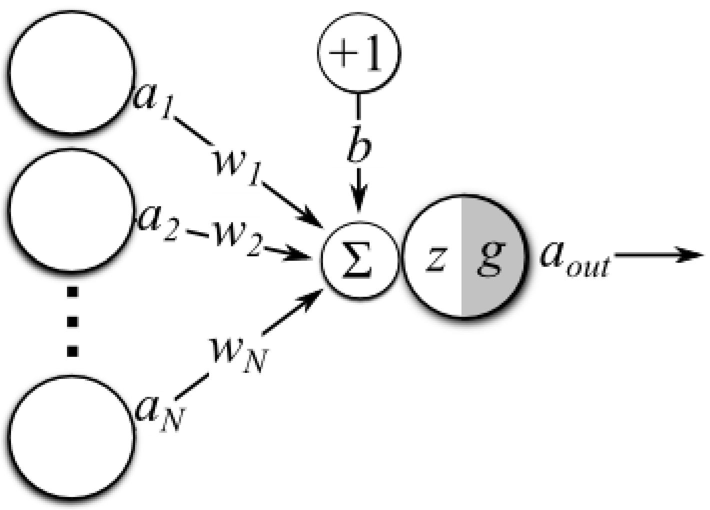


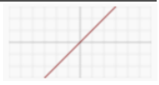
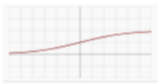
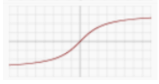
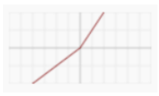
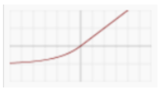
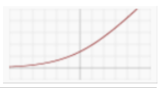
Figure 2.5: Working principle of a single artificial neuron. Arrows indicate direction of data flow where *a* are neuron activations, *w* connection weights, *b* neuron bias, *z* aggregated sum of the linear product of nodal activations and weights as well as the nodal bias, *g* activation function [51].

of supervised learning, it usually serves to calculate the degree to which the output response is inaccurate. To achieve an accurate output the error needs to be minimized. This is commonly achieved through *'gradient descent'*and *'back-propagation'* which are the processes by which adjustements are applied to the networks 'learnable parameters' i.e. network weights and biases. This repeated parameter self-adjustment ultimately results in the iterative improvement and learning of the model.

2.3.1 Loss Functions

For learning tasks such as regression learning, models are tasked with finding patterns in the data - a functional relationship between the inputs X and output Y components of the data. If our domain set of X is $\mathbb{R}^i$ (for i number of input variables) and the set of Y "labels" is the set of $\mathbb{R}$ Real numbers it may be more adequate to call Y the *"target"* set. The training data remains a finite sequence of $\{(x_i, y_i)\}_{N_i=1}$ and the

Table 2.1: Frequently used Activation Functions

Name	Equation	Derivative	Plot
Identity	$f(x) = x$	$f^{'}(x) = 1$	
Binary Step	$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$	$f^{'}(x) = \begin{cases} 0 \text{ for } x \neq 0 \\ ?^1 \text{ for } x = 0 \end{cases}$	
Logistic / Soft Step	$f(x) = \frac{1}{1+e^{-x}}$	$f^{'}(x) = f(x)(1-f(x))$	
TanH	$f(x) = \frac{2}{1+e^{-2x}} - 1$	$f^{'}(x) = 1 - f(x)^2$	
ArcTan	$f(x) = tan^{-1}(x)$	$f^{'}(x) = \frac{1}{(x)^2+1}$	
Rectified Linear Unit	$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	$f^{'}(x) = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$	
Parametric Rect. Linear Unit	$f(x) = \begin{cases} \alpha x \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$	
Exponential Linear Unit	$f(x) = \begin{cases} \alpha(e^x - 1) \ ; x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	$f^{'}(x) = \begin{cases} f(x) + \alpha \ ; x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$	
SoftPlus	$f(x) = log_e(1+e^x)$	$f^{'}(x) = \frac{1}{1+e^{-x}}$	

output a function model that maps X to Y. The measure of success can be evaluated as the quality of a *hypothesis function*, $h : X \to Y$, by a measure of difference between the true labels / targets and their predicted values. For example a *square difference* would be given by Equation 2.3

$$L_{\mathscr{D}}(h) \overset{\text{def}}{=} \underset{(x,y)\sim\mathscr{D}}{\mathbb{E}} (h(x) - y)^2 \tag{2.3}$$

Since the hypothesis function may be any number of formulae it is possible to generalize Equation 2.3 [52]; Given any set of Hypotheses $\mathscr{H}$ and some domain Z let ℓ be any function from $\mathscr{H} \times \mathscr{Z}$ to the set of non-negative real numbers, $\ell : \mathscr{H} \times \mathscr{Z} \to \mathbb{R}_+$. Such functions are called *loss functions*. We can define *risk functions* as the expected loss of a classifier, $h \in \mathscr{H}$ with respect to a probability distribution D over Z, namely,

$$L_{\mathscr{D}}(h) \overset{\text{def}}{=} \underset{z \sim \mathscr{D}}{\mathbb{E}} [\ell(h,z)] \tag{2.4}$$

This states that we consider the expectation of the loss of h over objects z picked randomly according to $\mathscr{D}$. Similarly we can define *empirical risk* as the expected loss over a given sample $\mathscr{S} = (z_1, \cdots, z_m) \in \mathscr{Z}^m$, namely,

$$L_{\mathscr{S}}(h) \overset{\text{def}}{=} \sum_{i=1}^{m} [\ell(h,z_i)] \tag{2.5}$$

With application to the ML tasks of classification or regression for example yields Equation 2.6 and Equation 2.7 respectively:

1. **0-1 loss:** Random variable z ranges over the set of pairs $X \times Y$ and the loss function is

$$L_{0-1}(h,(x,y)) \overset{\text{def}}{=} \begin{cases} 0 \text{ if } h(x) = y \\ 1 \text{ if } h(x) \neq y \end{cases} \tag{2.6}$$

2. **Square Loss:** Random variable z ranges over the set of pairs $X \times Y$ and the loss function is

$$L_{sq}(h,(x,y) \overset{\text{def}}{=} (h(x)-y)^2 \tag{2.7}$$

2.4 Gradient Descent

The goal of ML is to minimize the *risk function*, $L_{\mathscr{D}}(h)$ of Equation 2.4. Ordinarily we could not do this directly as it depends on the unknown distribution D and we would instead use the *empirical risk* based on a training sample set S and attempt to minimize the empirical risk of an hypothesis function, $L_{\mathscr{S}}(h)$. A method which attempts to minimise the risk function directly and which has found considerable favor and usage in ML is *Stochastic Gradient Descent* (SGD). SGD attempts this using a gradient descent procedure as its name implies. SGD is an optimization technique which, recursively improves the solution by iteratively taking steps along the negative of the gradient of the function to be minimized at the present location. Instead of a single hypothesis function however it looks to a vector $\mathbf{W}$ of hypotheses of convex hypothesis class, $\mathscr{H}$. Furthermore since D is an unknown, the gradient of $L_D(\mathbf{W})$ is also unknwon. SGD bypasses this by permitting the technique to take a step in a random direction provided the value of the direction is expected to be the negative of the gradient [52]. This procedure is conceptually illustrated in Figure 2.6.

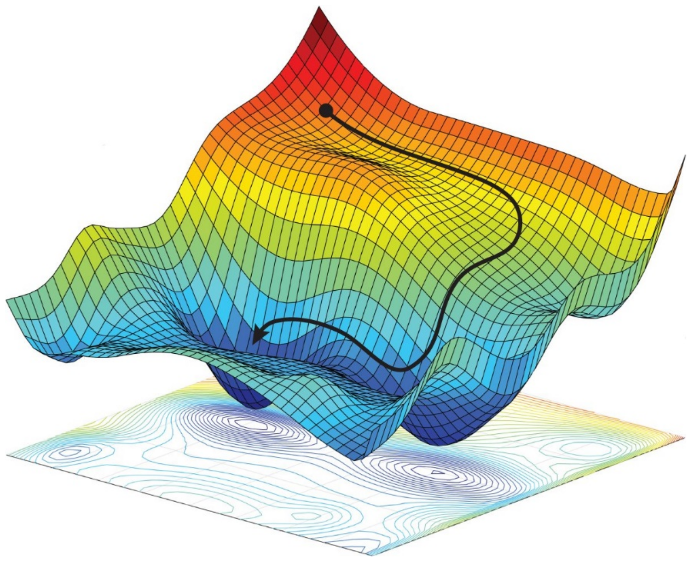


Figure 2.6: Illustrative representation of process of gradient descent which iteratively seeks to move towards a local/global minima along the path of steepest gradients [53]

2.4.1 Gradient Descent Algorithms

Gradient descent is a method to minimize an objective function, $J(\theta)$, parametrized by a model's *parameters* $\theta \in \mathbb{R}^d$. It does so through modifying the parameters in the direction opposite that of the *gradient of the objective function*, $\nabla_\theta J(\theta)$ with respect to the parameters.The magnitude of the steps taken to reach a (local) minimum is determined by the *learning rate* η. Simply put, gradient descent follows the slope of the objective function surface downhill until the base of a valley is reached [54]. Gradient descent can be separated into three groupings depending on the amount of data used to compute the gradients, namely

2.4.1.1 Batch gradient descent

Batch gradient descent is an idealised approach in that it is assured to converge to the global and local minimums for convex and non-convex error surfaces respectively. The caveat however is that to achieve this the gradient is calculated using the entire training dataset *per update*. Consequently this results in Batch gradient descent being prohibitively slow or even unachievable for large datasets which do not fit in the available memory. Furthermore, this eliminates it from 'online' usage as it is unable to update when presented with additional data. The Batch gradient descent update to the parameters is given by Equation 2.8:

$$\theta = \theta - \eta . \nabla_\theta J(\theta) \tag{2.8}$$

2.4.1.2 Stochastic gradient descent

Stochastic gradient descent (SGD), in contrast to Batch gradient descent, performs updates to parameters for every training example x^i and label y^i as expressed in Equation 2.9

$$\theta = \theta - \eta . \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{2.9}$$

Where Batch gradient descent may perform computations which are redundant for large datasets as it recalculates gradients for similar entries prior to updating each parameter, SGD removes this redundancy by consecutively performing updates. Consequently, SGD is usually much faster and has the added ability to learn 'online' as it can accept new data continuously. A caveat of SGD is that owing to its response to each data point in the training set it is likely to fluctuate significantly. In the end however, SGD exhibits the same convergence to minima when the learning rate is gradually decreased during training. [54]

2.4.1.3 mini-batch gradient descent

Mini-batch gradient descent falls between Batch and Stochastic gradient descent methods with respect to responsiveness and computational speed and load. Mini-batch gradient descent performs updates for each n "mini-batch" grouping of training examples.

$$\theta = \theta - \eta . \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.10}$$

The benefits of this are two-fold:

1. Reduction in the parameter updates variance, which may result in convergence that is more stable.

2. Allowance to incorporate common matrix optimizations of contemporary programming libraries which results in improved operational speeds and efficiencies.

Generally speaking ML and DL models usually opt for some form of mini-batching, so much so that the term SGD is used synonymously with mini-batched gradient descent [55]. Similarly, we adopt this naming convention in subsequent sections.

2.4.2 Gradient descent optimization algorithms

Over the years there have been several optimization techniques developed to improve the efficiency and rate of convergence of gradient descent methods implemented specifically within ML and DL systems. These optimizations are outlined here and summarised in Table 2.2 [55].

2.4.2.1 Momentum

SGD experiences difficulties navigating areas where the surface curved steeper along one axis than another which may occur around local optima. In such circumstances SGD may oscillate between adjacent slopes of topological 'ravines' whilst making only slight progress along the floor of the ravine towards the local optimum. In attempt to dampen this oscillation an additional fraction γ of the previous step's update vector is added to the current update vector as reflected in Equation 2.11:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{2.11}$$

The resultant effect is that the *"momentum"* term containing γ increases for dimensions whose gradients slope in the same direction and reduce updates for those which are counter directed. The net result is faster convergence and reduced oscillation.

2.4.2.2 *Nesterov accelerated gradient*

Nesterov accelerated gradients (NAG) works with momentum as its basis and attempts to give foresight with respect to expecting the slope to flatten before rising again and thus decrease the momentum preemptively:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \\ \theta &= \theta - v_t \end{aligned} \tag{2.12}$$

In Equation 2.12 we retain momentum through the momentum term γv_{t-1} however by computing $\theta - \gamma v_{t-1}$ we look ahead by calculating the gradient with respect to the approximate future position of the parameters.

2.4.2.3 *Adagrad*

Where the two aforementioned optimizations improve convergence specifically for slope of the objective (error) function, *Adagrad* is an algorithm for optimizations based on individual parameters. It adapts the learning rates which are applied on a *per parameter* basis, performing larger updates for infrequent parameters and smaller updates for frequent parameters. Consequently it is suitable for handling sparse data. As Adagrad uses a different learning rate for every parameter θ_i for every timestep t we set $g_{t,i}$ to be the gradient of the objective function with respect to the parameter θ_i at timestep t

$$\begin{aligned} g_{t,i} &= \nabla_{\theta t} J(\theta_{t,i}) \\ \theta_{t+1,i} &= \theta_{t,i} - \eta \cdot g_{t,i} \end{aligned} \tag{2.13}$$

Modifying Equation 2.13 to apply specific per parameter learning rates of Equation 2.13 yields Equation 2.14:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} \cdot g_{t,i} \tag{2.14}$$

$G_t \in \mathscr{R}^{d \times d}$ in Equation 2.14 is a diagonal matrix where each diagonal element i,i is the sum of squares of the gradients with respect to θ_i up to time-step t and ε is a small (usually of order $1e$-8) smoothing term that avoids division by zero. Since G_t contains the sum of squares of previous gradients relative to all parameters θ along its diagonal, the implementation Equation 2.14 can be vectorized by element-wise matrix-vector multiplication $\odot$ between G_t and g_t to arrive at Equation 2.15.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t \tag{2.15}$$

A primary benefit of the Adagrad technique is the lack of need to manually adjust the learning rate. The primary weakness however is its accumulation of the squared gradients in the denominator - Since each consecutive addition is positive the accumulated sum grows indefinitely which ultimately results in the learning rate becoming negligably small after which the algorithm ceases learning and is unable to update for additional knowledge.

2.4.2.4 *Adadelta*

A technique which aims to improve upon the aggressive decreasing learning rate of Adagrad is *Adadelta*. Instead of accumulating *all* past squared gradients, Adadelta limits the accumulation to a moving window of fixed size w. Furthermore, the sum of the gradients is recursively defined as a decaying average of past squared gradients instead of storing w previous gradients. The *running average*, $E\left[g^2\right]_t$, at time step t then depends only on the the current gradient and the previous average by a fraction γ (similar to *momentum* in Equation 2.11)

$$E\left[g^2\right]_t = \gamma E\left[g^2\right]_{t-1} + (1-\gamma){g_t}^2 \tag{2.16}$$

For clarity, standard SGD update of Equation 2.8 is rewritten in terms of the parameter update vector $\Delta\theta_t$:

$$\begin{aligned} \Delta\theta_t &= -\eta \cdot g_{t;i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{2.17}$$

The parameter update vector of Adagrad derived previously in Equation 2.15 subsequently takes the form:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t \tag{2.18}$$

We replace the diagonal matrix G_t with the decaying average over previous squared gradients $E\left[g^2\right]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t \tag{2.19}$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[g^2]_t} g_t \tag{2.20}$$

The units of Equation 2.20 do not match as they dont have the same units as the parameter. To account for this, another exponentially decaying average, this time of squared parameter updates is defined:

$$E\left[\Delta\theta^2\right]_t = \gamma E\left[\Delta\theta^2\right]_{t-1} + (1-\gamma)\Delta\theta_t{}^2 \tag{2.21}$$

The root mean squared error of parameter updates is thus:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \varepsilon} \tag{2.22}$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step $t-1$. Replacing the learning rate η in Equation 2.20 with $RMS[\Delta\theta]_{t-1}$ yields the final Adadelta update rule of Equation 2.23

$$\begin{aligned} \Delta\theta_t &= -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g^2]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{2.23}$$

The primary advantage of Adadelta is the removal of the need to set a default learning rate, as it is eliminated from the update rule.

2.4.2.5 Adam

Adaptive Moment Estimation (Adam) is a method that also computes adaptive learning rates for each parameter. Adam stores an exponentially decaying average of past gradients m_t in addition to the exponentially decaying past squared gradients v_t as done by Adadelta [56]:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) {g_t}^2 \end{aligned} \tag{2.24}$$

In Equation 2.24 m_t and v_t are estimates of the mean and uncentered variance of the gradients and are referred to as the first and second *moments* respectively. The moments would be biased towards zero as they are implemented as vectors of 0's during the initial steps, so to counter this, 'bias-corrected' first and second moments are defined:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-{\beta^t}_1} \\ \hat{v}_t &= \frac{v_t}{1-{\beta^t}_2} \end{aligned} \tag{2.25}$$

The bias-corrected moments of Equation 2.25 are used to update the parameters which produces the Adam update rule Equation 2.26

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \varepsilon}} \hat{m}_t \tag{2.26}$$

The authors of the paper [56] propose default values of $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$.

2.4.2.6 *AdaMax*

In the Adam update rule of Equation 2.26, the v_t factor scales the gradient inversely proportionally to the $L_2{}^2$ norm of the prior gradients through the v_{t-1} term and current gradient $|g_t|^2$:

As an extention of Adam by the same authors [56], they propose *AdaMax* which generalizes to the L_p norm[3]

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)|g_t|^2 \tag{2.27}$$

Furthermore β_2 is also parameterized as β_2^p:

$$v_t = \beta_2^p v_{t-1} + (1-\beta_2^p)|g_t|^p \tag{2.28}$$

Norms for large p values generally become unstable, however L_∞ generally exhibits stable behavior. Consequently, AdaMax proposes that v_t with L_∞ converges to the more stable value as follows

$$\begin{aligned} u_t &= \beta_2^\infty v_{t-1} + (1-\beta_2^\infty)|g_t|^\infty \\ &= \max(\beta_2 \cdot v_{t-1}, |g_t|) \end{aligned} \tag{2.29}$$

In Equation 2.29 u_t denotes the infinity norm-constrained v_t: Substituting Equation 2.27 through Equation 2.29 into the Adam update of Equation 2.26 by replacing $\sqrt{\hat{v}_t} + \varepsilon$ with

[2]L_2 norm is a standard method to compute the length of a vector in Euclidean space. Given $x = [x_1 x_2 \cdots x_n]^T$, L_2 norm of x is defined as the square root of the sum of the squares of the values in each dimension.[57]

[3]L_p norm is length of a vector in p space. For a real number $p \geq 1$, the p-norm or L_p-norm of x is defined by $\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p}$. [58]

u_t produces the AdaMax update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t}\hat{m}_t \tag{2.30}$$

Note that due to the liance on the max operation for u_t , it is not as likely to bias towards zero as m_t and v_t in Adam, which is why there is no need to derive a bias correction for u_t. The authors proposed default values for $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\eta = 0.002$.

Table 2.2: Summary of SGD optimization algorithms. Table created by author

Method	**Rule**	**Advantages**
Momentum	$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$ $\theta = \theta - v_t$	Faster convergence Reduced Oscillation
NAG	$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$ $\theta = \theta - v_t$	Anticipates slope flattening
AdaGrad	$g_{t,i} = \nabla_{\theta t} J(\theta_{t,i})$ $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t$	Adapts learning rate *per parameter* Suitable for sparse data
Adadelta	$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g^2]_t} g_t$ $\theta_{t+1} = \theta_t + \Delta\theta_t$	Removes need to set learning rate
Adam	$\hat{m}_t = \frac{m_t}{1-\beta^t{}_1}$, $\hat{v}_t = \frac{v_t}{1-\beta^t{}_2}$ $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \varepsilon}} \hat{m}_t$	Suited large data/parameters situations Appropriate for noisy/sparse gradients
AdaMax	$u_t = \max(\beta_2 \cdot v_{t-1}, \lvert g_t \rvert)$ $\theta_{t+1} = \theta_t - \frac{\eta}{u_t} \hat{m}_t$	Suitable for time variant data

2.5 Backpropogation

The preceding section outlined the goal of gradient descent and its optimization methods concerned with rapidly and efficiently arriving at a minima along the multi-dimensional surface which defines the objective (generally loss) function. In order to actually take a step along the surface, within the context of neural networks, an additional step is required to make alterations to the internal parameters which define the network - The process by which the network learnable parameters / weights and biases are changed is called *back-propogation*. The process was first proposed by Rumelhart, Hinton and Williams in 1986 [59]. Their described means of implementation has changed very little and is currently the defacto means by which the majority of ML and DL architectures are adjusted during training. To quote the paper directly;

"The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the [neural] net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units"

In implementation the back-propogation algorithm is essentially an application of the *chain rule* which runs recursively backwards from the last network layer to the first along every internal neural connection. We frequently encounter situations in which the quantities appearing in a function are themselves functions of another variable. Accordignly, the chain rule states that if you have three functions f, g and h with f being a function of g and g being a function of h then the derivative of f with respect to h is equal to the product of the derivative of f with respect to g and the derivative of g with respect to h [60]:

$$\frac{\partial f}{\partial h} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h} \tag{2.31}$$

Consider the hypothetical network in Figure 2.7. The network is composed of L layers (an input layer, an output layer, and $L-2$ hidden layers). Each layer consists of N_v neurons, where $v = 1,2,\cdots,L$ is the index of the layer. Figure 2.7 displays a closer

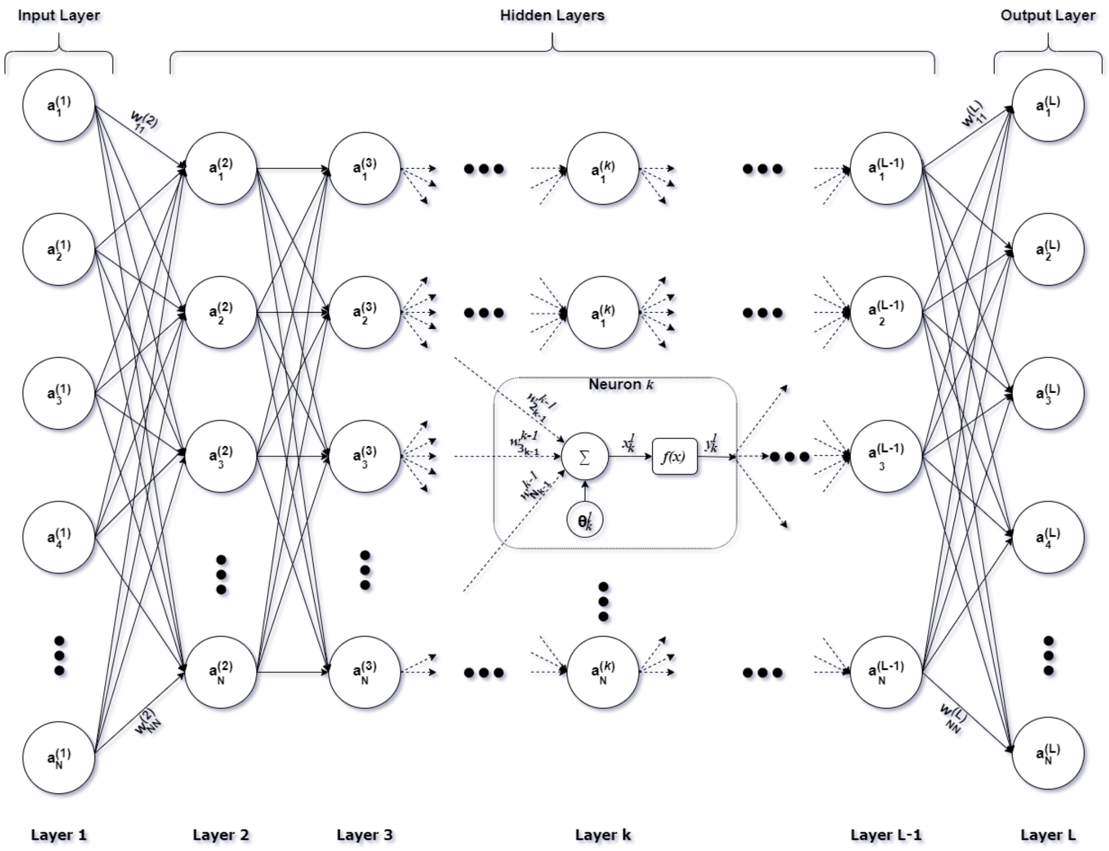


Figure 2.7: Architecture of a Hypothetical Neural Network. Adapted from [61].

look at an arbitrary neuron of layer *k*.

During training, the input sample is propagated in the forward direction until the output emerges from the output layer. This output can be viewed as a nonlinear function of the input in terms of the current weights, biases and thresholds of the activation functions applied (if any). A cost/loss function (or more generally an objective function) is then used to compute the error between the actual output of the network and the desired target output associated with that input sample. Thereafter, the error is back-propagated from the output layer through the hidden layers and finally to the input layer. During the back-propagation process, the sensitivity of error to each weight, bias and threshold in the network is obtained. These sensitivities are used to update the weights and thresholds of the network according to the gradient decent method. This process of forward-propagation, back-propagation, and weights updating is iterated until the network parameters (i.e. weights, biases and activation function thresholds) converge to optimal values that minimize the error between the actual output and desired targets.

In forward-propagation, the neuron collects the weighted outputs of layer $k-1$, sums them to a threshold $\theta^l{}_k$ and then applies an activation function $f(x)$ on the result of the summation to compute the output of the neuron $y^l{}_k$. This forward-propogation process can be expressed by Equation 2.32 and Equation 2.33 and was described in section 2.3

$$x_k^l = \theta_k^l + \sum_{i=1}^{N_{k-1}} w_{ik}^{k-1} y_i^{k-l} \tag{2.32}$$

$$y_k^l = f(x_k^l) \tag{2.33}$$

Note that the superscripts in the above equations denote the layer to which the quantities belong. Accordingly, the input propagates throughout the entire network (neuron by neuron) until an output vector $Y = \left[y_1^L, y_1^L, y_1^L, \cdots, y_{N_L}^L\right]^T$ emerges from the last layer L. Now suppose that the input-output example used in this training iteration consists of an input $P = \left[p_1^1, p_2^1, p_3^1, \cdots, p_{N_1}^1\right]^T$ associated with a desired target vector $T = [t_1, t_2, t_3, \cdots, t_{N_L}]^T$. The error E between the actual and desired outputs can be then computed according to the following cost function:

$$E = \frac{1}{2}\sum_{i=1}^{N_L} \left(y_i^L - t_i\right)^2 \tag{2.34}$$

You will note that this is equivalent to the square loss of Equation 2.7 with the addition of the factor $\frac{1}{2}$ introduced here for convenience to cancel out the exponent when the cost function is differentiated with respect to Y as will be shown. As noted previously, the objective of back-propagation algorithm is to optimize the network's weights and biases in order to minimize the cost function. To do so, the first step is to obtain the sensitivity factors. For the k^{th} neuron of layer l, the sensitivity factors to be computed are the partial derivatives of the error E with respect to the weights $w_1k, w_2k, w_3k, \cdots, w_{N_{l-1}k}$ along with the bias θ_k^l, which can be defined according to the chain rule as follows:

$$\begin{aligned} \frac{\partial E}{\partial w_{ik}^{l-1}} &= \frac{\partial E}{\partial x_k^l} \frac{\partial x_k^l}{\partial w_{ik}^{l-1}} \\ &= \frac{\partial E}{\partial x_k^l} y_i^{l-1} \end{aligned} \tag{2.35}$$

$$\begin{aligned} \frac{\partial E}{\partial \theta_k^l} &= \frac{\partial E}{\partial x_k^l} \frac{\partial x_k^l}{\partial \theta_k^l} \\ &= \frac{\partial E}{\partial x_k^l} \end{aligned} \tag{2.36}$$

From Equation 2.35 and Equation 2.36 we can infer that the sensitivity factors are dependent on the partial derivative of the error with respect to x_k^l. This derivative is often referred to as the delta error of the neuron Δ_k^l

$$\Delta_k^l = \frac{\partial E}{\partial x_k^l} \tag{2.37}$$

According to the chain rule, the delta error of any neuron can be expressed as:

$$\begin{aligned} \Delta_k^l &= \frac{\partial E}{\partial y_k^l} \frac{\partial y_k^l}{\partial x_k^l} \\ &= \frac{\partial E}{\partial y_k^l} f'(x_k^l) \end{aligned} \tag{2.38}$$

For a neuron k in the output layer L, Equation 2.38 be simplified to Equation 2.39

$$\Delta_k^L = (y_k^l - t_k).f^{'}(x_k^l) \tag{2.39}$$

However, for the neurons in layers 1 to $L-1$ it is necessary to derive a formula for computing the derivative $\frac{\partial E}{\partial y_k^l}$. As shown in Figure 2.7 the output of the k^{th} neuron of layer l (i.e. y_k^l) contributes to the input of neurons of the next layer $l+1$. Accordingly, it can be proved that the derivative of the error with respect to y_k^l is:

$$\begin{aligned} \frac{\partial E}{\partial y_k^l} &= \sum_{i=1}^{N_{l+1}} \frac{\partial E}{\partial x_i^{l+1}} \frac{\partial x_i^{l+1}}{\partial y_k^l} \\ &= \sum_{i=1}^{N_{l+1}} \Delta_i^{l+1} w_{ki}^l \end{aligned} \tag{2.40}$$

Substituting Equation 2.40 in Equation 2.38 gives the delta error corresponding to any neuron k in layers 1 to $L-1$:

$$\Delta_k^l = f^{'}(x_k^l) \sum_{i=1}^{N_{l+1}} \Delta_i^{l+1} w_{ki}^l \tag{2.41}$$

Equation 2.41 suggests that the delta errors, and hence the sensitivity factors, of a certain layer l depend on the delta errors of the next layer $l+1$. Therefore, in back-propagation algorithm, we start by calculating the delta errors of the output layer L, then we move in the backward direction and calculate the delta errors starting from the last hidden layer $L-1$ until arriving at the input layer. Once the delta errors and sensitivity factors are computed for all neurons, the weights and thresholds can be updated according to the gradient decent method as expressed by the methods described in section 2.4.

The standard backpropagation algorithm for training of NN networks can be summarized in the following steps [59]:

1. Select the hyperparameters that determine the architecture of the ANN (number of hidden layers and number of neurons in each hidden layer).
2. Initialize the weights and biases by uniformly distributed random numbers
3. Pick an input-output sample (or a batch of input-output samples) and forward-propagate its input through the network according to Equation 2.32 and Equation 2.33.
4. Compute the error between the actual output and the desired target associated with the forward-propagated input-output sample(s) as expressed by equation Equation 2.34.
5. Calculate the delta errors of the output layer according to equation Equation 2.39.
6. Calculate the delta errors of the other layers starting from the hidden layer $L-1$ until the input layer according to equation Equation 2.41.
7. Obtain the sensitivity factors as explained by... Equation 2.35 to Equation 2.37.
8. Apply the gradient decent method to update the weights and thresholds according to a chosen technique in section 2.4.
9. Repeat steps 3 to 8 for each sample (or batch of samples) in the training dataset.
10. Repeat steps 3 to 9 for a sufficient number of epochs[4] until a certain stopping criterion is satisfied.

[4]An epoch is defined as one complete pass of the entire training dataset.

2.6 Convolutional Neural Networks

A specific class of Neural Networks well suited to data distributed across areas or volumes is the *Convolutional Neural Network* (CNN). CNNs are primarily used to solve difficult image-based tasks such as pattern recognition, classification and machine vision problems such as those in medical imaging [62; 63] or traffic prediction and autonomous vehicles [64; 65]. A primary identifying characteristic of the CNN is that its input is designed to accept input structured into 3 dimensions; height, width and depth, with depth referring to the 3rd dimesion of the input and not depth of layers of the CNN. Architecturally CNNs are composed of convolutional layers, fully-connected layers and pooling layers. Successive stacking of the layers with additional activation functions at key points between results in the formation of a complete CNN (see Figure 2.8).

Functionally the CNN can be broken down into four primary operations [66]:

1. An input layer stores the values for the input images' constituent pixels.
2. The convolutional layer with rectified linear unit (ReLu) activation function determines the output to the next (and often dimensionally smaller) layer of neurons through the calculation of the scalar product between their weights and the region of input layer each neuron is connected to.
3. The pooling layers are tasked with downsampling along a chosen spatial dimension of the input fed to it and in doing so further reduce the number of parameters.
4. Fully-connected layers in conjunction with activation functions attempt to produce scores to be used for classification.

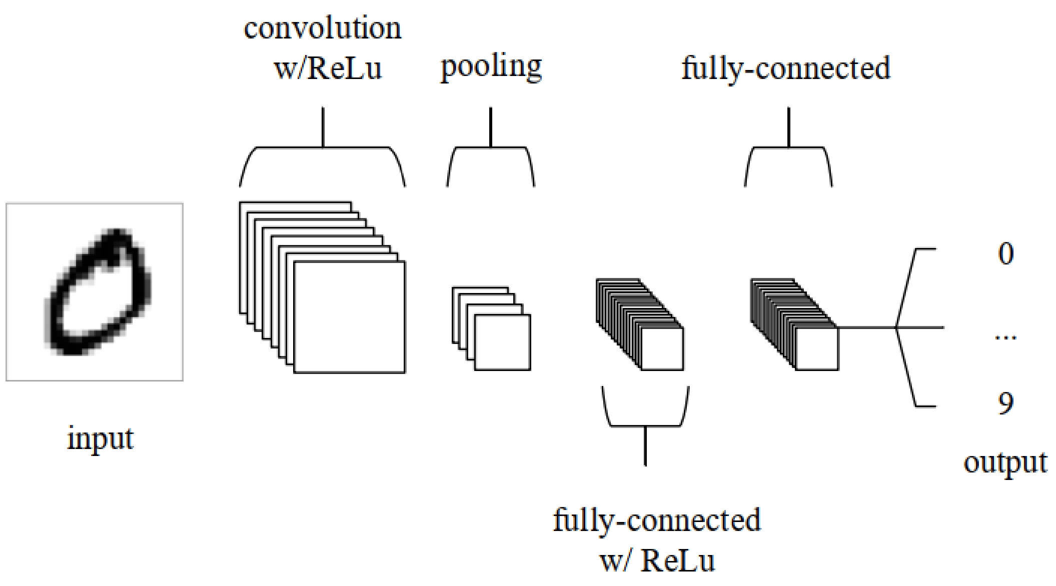


Figure 2.8: Example of a simple CNN architecture used in the classification of handwritten digits [66]

2.6.1 Convolutional Layers

The primary layer, and basis from which the CNN derives its namesake, is the *convolutional layer*. The principle operation of the convolutional layer is focused around the use of learnable kernels. These kernels are, in general, spatially much smaller than the input to their layer but extend along the full depth of the input. Operationally, the kernel, which is an $N \times N$ array, has an input feature map in which each element has a value. This map operates as the values of the linear transformation for each input value it encounters as the kernel moves across the totality of the input:-i.e. At each location as the kernel moves across the input, the product between each element of the kernel feature map and the input it overlaps is computed. These values are summed to obtain the value for each location. The final output of the procedure is the output feature map. Factors which affect the output feature map of the convolutional layer are:

- Kernel size

- Stride
- Padding

The kernel size dictates the $N \times N$ region which is summed at each pass, the stride relates to the step size (in number of elements) of the kernel across adjacent elements after each operation, whilst padding refers to the use (or lack thereof) of additional elements along the periphery of the data to be read by the convolutional layer.

Furthermore the alteration of these convolutional layer parameters defines the spatial dimensionality of the convolutional layers output. The resultant output size can be calculated using Equation 2.42:

$$\frac{(V-R)+2Z}{S+1} \tag{2.42}$$

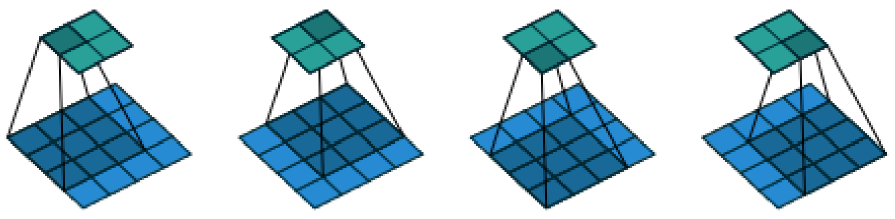
where V represents the input volume, R represents the kernel volume, Z is the amount of padding and S is stride.

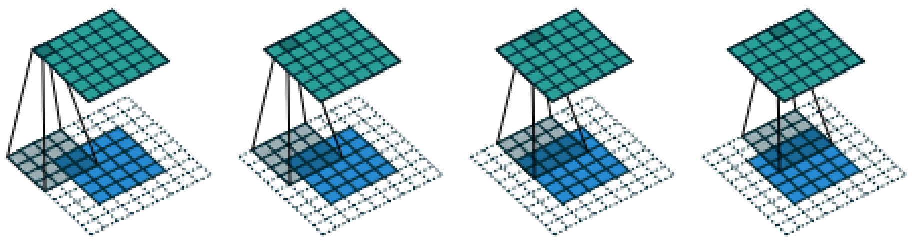
Several examples which vary these paramaters are given in Figure 2.9. For a more in-depth and comprehensive discussion on convolutional layers the reader is directed to [53].

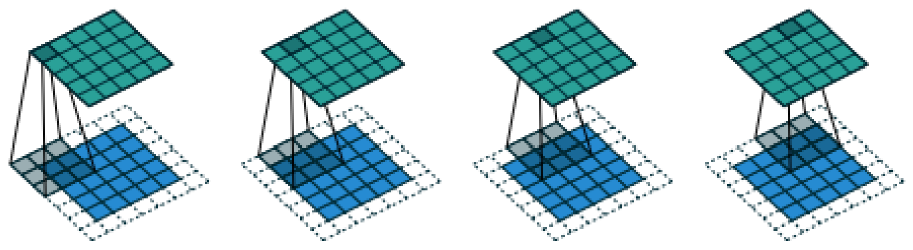
2.7 Generative Adversarial Networks

The generative Adversarial Network (GAN) is a specialised form of the CNN which is primarily utilized in the creation of images. It was first proposed and demonstrated by Ian Goodfellow et al as recently as 2015 [67] and since then it has grown in popularity with numerous derivative variations being constantly developed [68].

The core principle relies on the competition between two rival Neural Networks;


(No padding, unit strides) Convolving a 3×3 kernel over a 4×4 input using unit strides (i.e., $i = 4$, $k = 3$, $s = 1$ and $p = 0$).


(Arbitrary padding, unit strides) Convolving a 4×4 kernel over a 5×5 input padded with a 2×2 border of zeros using unit strides (i.e., $i = 5$, $k = 4$, $s = 1$ and $p = 2$).


(Half padding, unit strides) Convolving a 3×3 kernel over a 5×5 input using half padding and unit strides (i.e., $i = 5$, $k = 3$, $s = 1$ and $p = 1$).

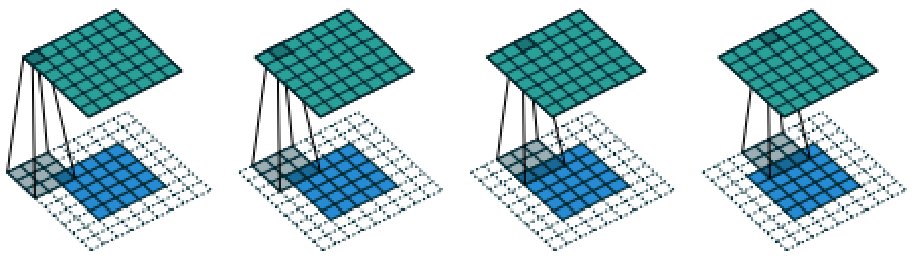

(Full padding, unit strides) Convolving a 3×3 kernel over a 5×5 input using full padding and unit strides (i.e., $i = 5$, $k = 3$, $s = 1$ and $p = 2$).

Figure 2.9: Examples of the working process of a 2-D Convolutional layer with variations to Input size i, Kernel size k, Stride s and Padding p [53]

a *Generator*, *G* and a *Discriminator*, *D*; hence their "adversarial" nature. The Generator is tasked with producing an image which it aims to pass off as genuine to the Discriminator. The Discriminator in turn is tasked with identifying whether the images it receives from the Generator are in fact “real”, or “fake”. For the Discriminator to have a reference it is supplied two images during the training phase – one sample from the dataset of genuine ‘ground truth’ images as well as the image from the Generator. After assessing the level to which it believes the generator is real (or fake) via loss functions, the results are back propagated for each network (*G* and *D*) and the cycle is repeated. The Generator’s aims to minimize the likelihood of creating a fake (or maximizing the likelihood of passing off its images as real) whilst the Discriminator aims to minimize the opposite i.e. minimize the likelihood of accepting a fake (or maximizing the likelihood of identifying a forgery). This competitive loop results in a simultaneous improvement of both networks which ultimately results in the Generator being capable of producing near-ground-truth images.

2.7.1 Image-to-Image Translation using conditional GANs

A subset of GANs that are not only interested in generating images but tailoring the images generated to meet certain imposed criteria are deemed "conditional GANs" (cGANs). Amongst the first to explore and develop such a cGAN was that of Phillip Isola et al [69] with their seminal "Pix2Pix" model.

Foundationally, its working principle is almost identical to most GANs that operate on the competitive Generator vs Disciminator game-theory approach as discussed, however instead of simply generating an image from random input noise and discerning whether or not it is passable as genuine, it learns the mapping from an input image

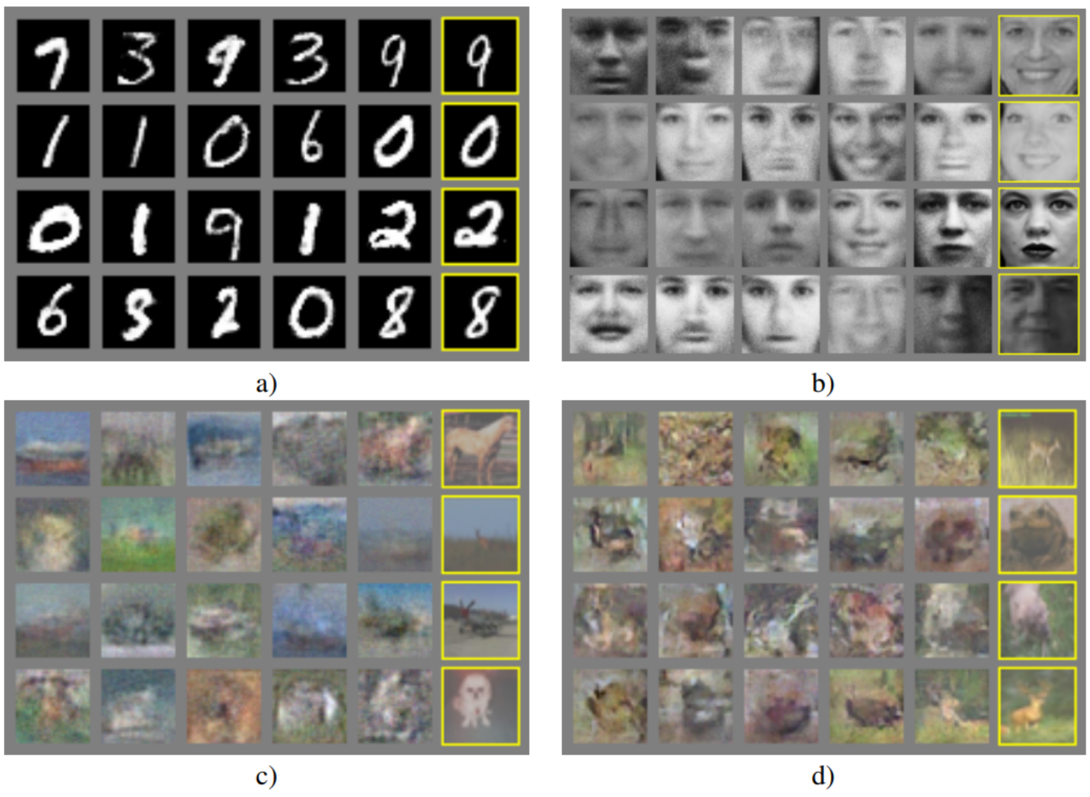


Figure 2.10: Examples of outputs from the GAN developed by Goodfellow et al. [67] at work: Generating (a) "handwritten" numbers, (b) faces, (c) and (d) animals

to an output image. Furthermore, additional novelty is achieved through a self-taught loss function: unlike conventional CNNs which aim to minimize a loss function which it explicitly needs to be instructed to be minimized, Pix2Pix learns the mapping via a loss function that is itself taught from the dataset with little to no user input- essentially a "universal" loss function. What this allows is for the same approach to be applied to various problems which would traditionally require unique loss functions for each category of input.

Isola demonstrates that the Pix2Pix method can be utilised on a broad range of datasets for a variety of tasks such as

- Architectural labels transformed into building images
- Maps transformed to aerial photos
- Colorization of black and white images
- Transforming user drawn sketches into photos etc

Whilst these examples may find real-word usage as suitable solutions to their given problems, their artistic and highly subjective nature has led to further developements focussing predominantly on non-technical investigation cases such as landscape [70] or face generation[71].

2.8 Summary

The highly diverse, almost categorically agnostic, range of image-to-image translation tasks that cGANS are capable of accepting and generating results for leads one

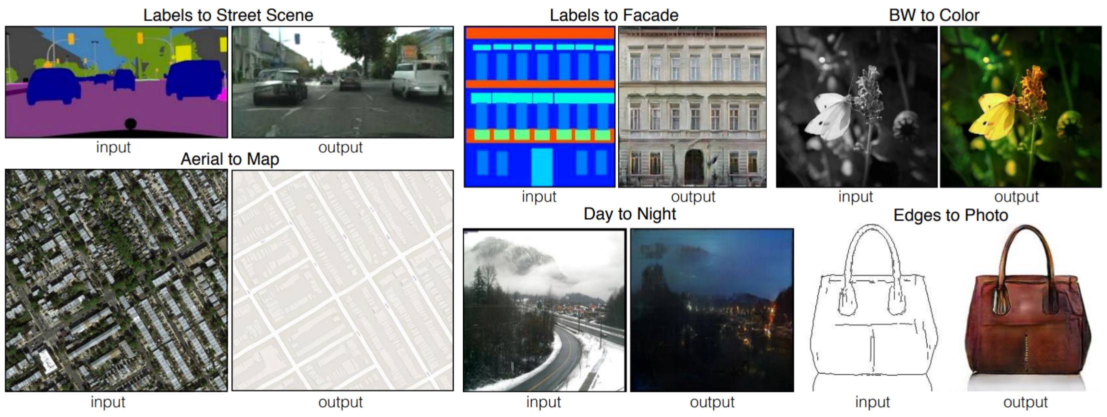


Figure 2.11: Example output of the diverse class of problems Pix2Pix image-to-image translation is able to attempt. In each Image Pair the left images represents the input to Generator whilst the right represent the generated results [69]

to speculate that they may be a promising approach for application to FEA tasks, especially since those tasks typically involve structured inputs and outputs. Furthermore, the autonomous nature by which the network learns both a loss function and subsequent mapping adapted to the task and data at hand, potentially makes them applicable to a wide variety of FEA problems and settings without the need to redesign the neural network for each problem. cGANs could hypothetically be tasked to emulate the input-output processes of FEAs if simply presented with data which has been engineered to meet with the structural (image-based) requirements of the network architecture and assessed on metrics which promote technical accuracy.

3 Methodology

The approach taken in attempt to train a ML model on the behavioural characteristics of platework relies on the creation of a well structured input dataset, the necessary neural architecture to train the models as well as the means to transform the data into valid inputs for the network itself. In order to achieve this the methodology of execution is broken down into six primary stages:

1. The FEA datasets on which the cGAN will be trained are generated.
2. The “Raw” datasets are pre-processed into a structure which can be read and interpreted by the cGAN.
3. The cGAN is formally defined and implemented in Matlab.
4. Generative models are trained in the cGAN.
5. The trained models are validated against ‘ground truth’.
6. A user interface allowing real-time predictions is implemented.

This sequence of stages is outlined in Figure 3.1

3.1 Generate FEA Dataset

The hypothesis that a GAN can be trained to represent the workings of the physical world relies strongly on the data presented to it. As mentioned, the thesis objective is to train a ML model to implicitly "understand" the physical properties and subsequently produce the displacement and stress responses of a common subset of FEA problems.

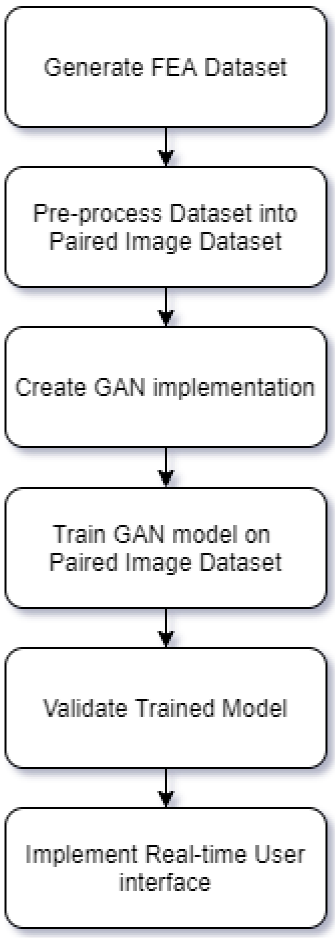

Generate FEA Dataset
Pre-process Dataset into Paired Image Dataset
Create GAN implementation
Train GAN model on Paired Image Dataset
Validate Trained Model
Implement Real-time User interface

Figure 3.1: Primary steps carried out in the Thesis methodology

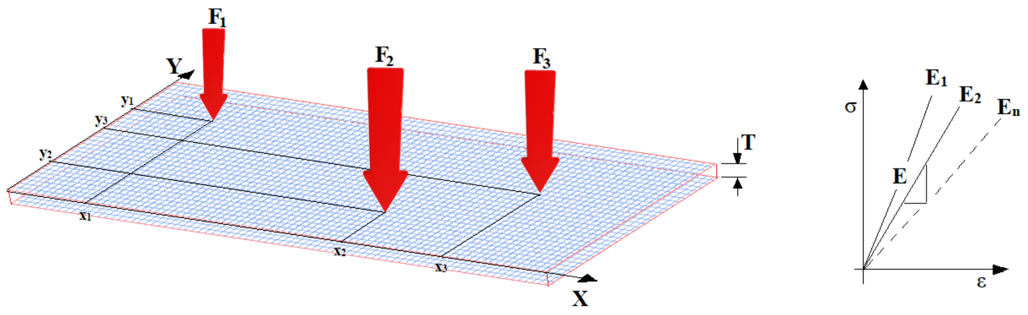


Figure 3.2: Illustrative experimental setup. A thin plate model with variable thickness T, modulus of elasticity E, three (of a maximum of five) representative variable perpindicular forces, F_1, F_2 and F_3 with their locations $[x_1,y_1]$, $[x_2,y_2]$ and $[x_3,y_3]$ respectively.

To this end it is essential that the generated dataset itself contains the relevant information capable of inferring the relationships we seek to simulate.

For the purposes of the investigation we will be attempting to model the response of thin platework[1] subjected to varying loads (in number of loads, positions, and range of magnitudes) for a range of moduli of elasticity and thicknesses. All edges are clamped fixing them in all 6 DOF. The choice of simplistic two-dimensional geometry is deliberate - A large area with numerous input-output coordinates can be rapidly produced. Furthermore should the outcomes of the thesis be successful the technique can potentially be built upon for more elaborate geometries and structures[2]. A representative illustration of the experimental setup is given in Figure 3.2.

3.1.1 APDL Pseudocode

To generate the dataset an ANSYS Mechanical APDL© batch script was written wherein nested loops sequentially increment the variables of:

[1]In plate theory plates with a thickness to length ratio of $\frac{1}{25}$ or less are considered thin [72]

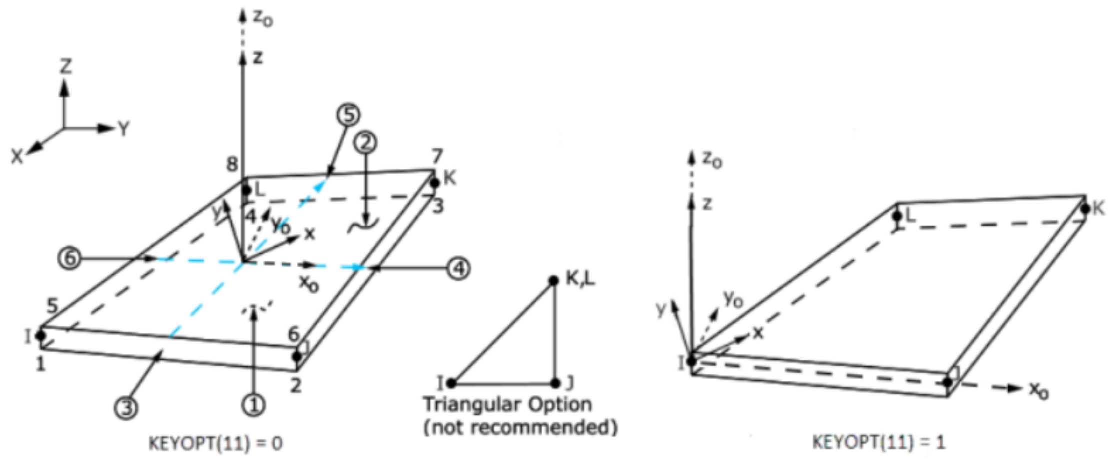
[2]As an additional note, whilst the study of thin plates is a field of research all its own, to the best of the authors knowledge there currently exist no known exact solutions for clamped rectangular plates under multiple varying loads [72; 73]

- Modulus of Elasticity
- Material thickness
- Aspect ratio,
- 5 sequential force additions applied to the platework mesh.

3.1.1.1 SHELL181

The plate is subdivided into a uniformly spaced grid mesh of 256x256 (65 536 total) nodes assigned element type Shell181. Shell elements such as SHELL181, are used as they provide both accuracy and efficiency while being suitable for the robust automatic meshing algorithms ANSYS provides. SHELL181 is suitable for the analysis of shell structures of thin to moderate thickness. Each element has four nodes with six degrees of freedom at each node (see Figure 3.3. SHELL181 is suitable for applications having both linear, large rotation, or large strain, nonlinear behaviour. Furthermore, SHELL181 allows for layered applications when modeling sandwich constructions or composite shells which allows further scope for potential future studies. The formulation of the elements is based on true stress and logarithmic strain measures. SHELL181 kinematics permits finite membrane stretching. Changes to the curvature within a time increment are however assumed to be minor. Further element properties are provided with the ANSYS release notes [74]

During each loop, the modulus of elasticity, thickness and aspect ratio for the plate is defined. The boundary conditions are set as fully fixed in all 6 DOFs for the perimeter nodes. Thereafter a perpindicular force is randomly applied (in both magnitude and planar location) within the inner domain of nodes which make up the mesh.



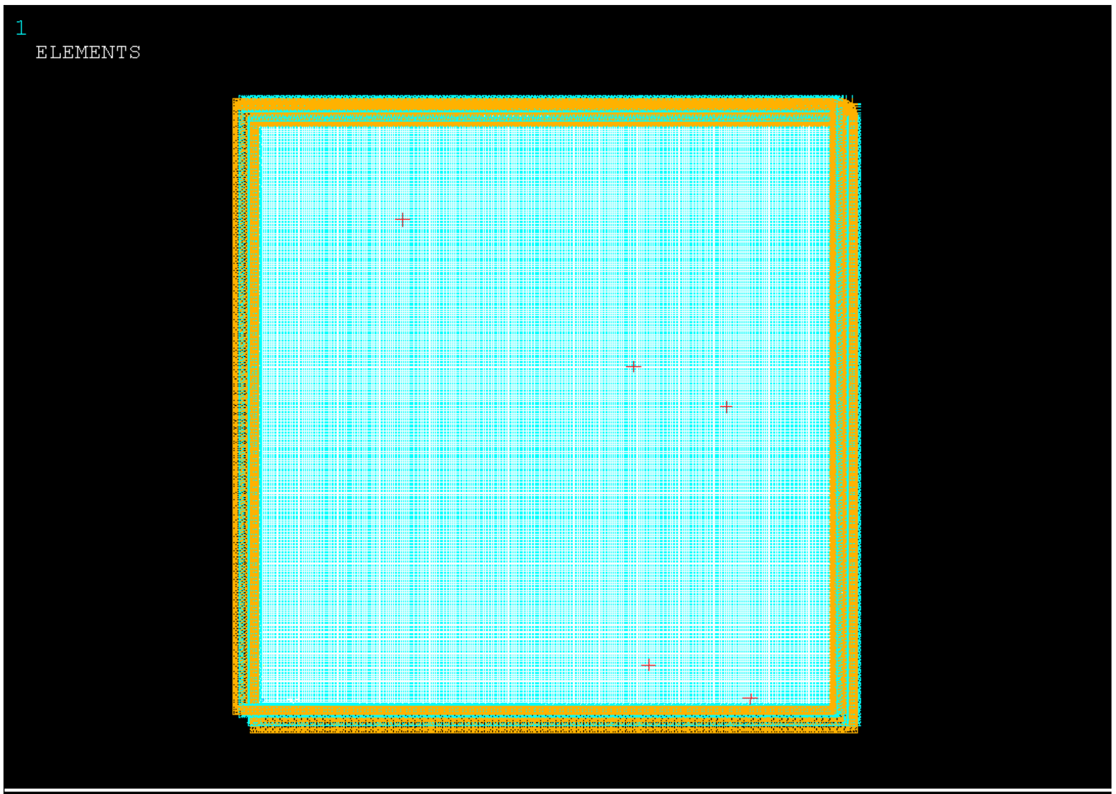
x_0 = Element x axis if element orientation (**ESYS**) is not specified.

x = Element x axis if element orientation is specified.

Figure 3.3: ANSYS SHELL181 Element geometry. Four-node element with six degrees of freedom at each node [74].

The simulation is then run and solves for the plate deflection as well as the 1^{st} principal stress of all the platework's nodes. The subsequent locations, deflections and stress values of each node are then written out to text file. Additionally, nodal locations and magnitudes of the force(s) are also written out to a secondary text file. Thereafter an additional load is applied to the mesh and the solution step rerun and recorded. Figure 3.4 presents a single example output of the APDL script.

This process is repeated for up to 5 loads in total per simulation as well as 25 repeated "trial cycles" to generate randomness and a wider statistical spread in the dataset. Thereafter, the parent loops of aspect ratio, thickness and modulus of elasticity are sequentially incremented as the process repeats. Using this methodology, a large dataset can be generated in a relatively short period of time: 6 modulus of elasticity, 8 plate thicknesses, 1 aspect ratio, 25 repeat trials with 5 consecutive force additions amount to a dataset of 6000 samples. Generation of this data set took a little over a week, 8.5 days in fact, running continuously on a modest workstation. The flowchart of



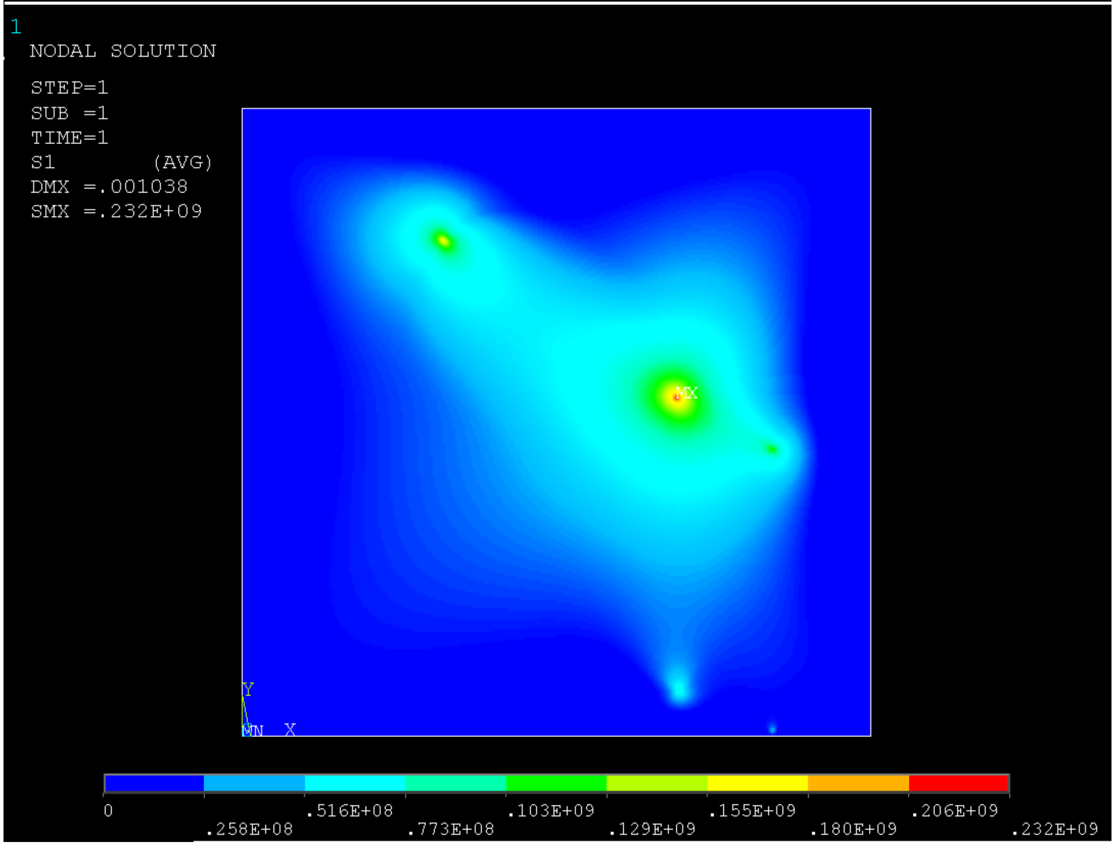


Figure 3.4: Output of the APDL batch script. Top: Meshgrid with boundary conditions and 5 loads randomly applied. Bottom: Solution stress results in false-color postprocess view

the script is outlined in figure 3.5 with full script code provided in Appendix A.

3.2 Pre-process FEA Dataset into Paired Image Dataset

The cGAN architecture is capable of processing input data in the form of structured pairs of images. One image as the Input and another as the Target we wish to replicate.

3.2.1 Images as a Data Structure

To utilize the FEA dataset it is necessary to convert each of the simulation results into structured image pairs. To achieve this, it is important to consider the implications and constraints of image data itself:

A rasterized image file whether it be a Bitmap [.bmp], Portable Network Graphics [.png], Joint Photographic Experts Group [.jpeg] or otherwise is composed of a matrix of pixels each with a minimum of 3 channels; traditionally red, green and blue; with an assigned bit depth of either 8bits, 16bits, or more, per channel. The bit depth corresponds to the magnitude of the color applied within each channel at each pixel and hence corresponds to 256 [0-255] or 65536 [0-65536] shades of color per channel for 8bit or 16 bit respectively [75].

What this infers is that the level of granularity for the data range which each pixel's channel can contain is limited by its bit depth. Additionally, the file and subsequent dataset size is also a function of the bit depth chosen. Furthermore, depending on the format, compression artefacts may occur which could result in data corruption. A comparison of high quality image formats is provided in Table 3.1. For the purposes of this initial investigation an 8bit depth is utilised with .png file format as it is a lossless data compression format [76].

Table 3.1: Comparison of common image file formats. Recreated from [76].

Comparison of Image File Formats				
Parameter	GIF89a	JPEG	TIFF	PNG
Max. Color Depth	8-bit	24-bit	48-bit	48-bit
Number of Colors	256 colors	16 million	281 trillion	281 trillion
Compression Technique	Lossless	Lossy	Lossy	Lossless
Gamma correction	No	No	Yes	Yes
Patent Issues	Yes	No	Yes	No

The variables which need to be transferred to image are:

- The modulus of elasticity
- The material thicknesses
- The locations and magnitudes of the forces
- The nodal deflections
- The nodal stresses.

In order to convey the maximum amount of information to each image file generated whilst also establishing a relationship between all the other images in the domain of the entire dataset, each of the 3 "Red", “Green" and "Blue" channels is assigned a corresponding input variable or response.

For the input image we assign the 1st "red" channel to force magnitude, 2nd "green" channel to the modulus of elasticity and the 3rd "blue" channel to material thickness. Similarly, for the output "Target" image, deflection is assigned to the 1st "red" channel whilst modulus of elasticity and material thickness are once again assigned to the 2nd and 3rd channels respectively.

3.2.2 Normalization of data across entire dataset

In order to facilitate that information in each dataset input and output image pair is correctly scaled with respect to one another and the dataset as a whole, each of the input variables and responses are required to be normalised. Hence the maximum and minimum values for each input and response is recorded over the entire domain of data derived from the APDL simulations. Thereafter we normalise each trial's variables and responses by dividing them by their respective maximums. This results in each value obtaining a value between [0,1]. To translate these normalised values to the domain of the images we then multiply each entry by the bit depth of each channel i.e. 255 (channel color range is 0-255). The resultant 3-dimensional colorspace of potential input values mapped to pixel colour is illustrated by Figure 3.6. Furthermore, the Paired Image creation process is schematically illustrated in Figure 3.8 with an example of the 2-dimensional image files demonstrated in Figure 3.7:

3.3 Define and Implement GAN in MATLAB

The cGAN which is implemented in the Thesis follows the working principles and network architecture of "Pix2Pix" by Phillip Isola et al. [69].

3.3.1 Network Architectures

3.3.1.1 Generator

A GAN generator, G, conventionally takes the form of an autoencoder-decoder, whereby the encoder portion is tasked with mapping the input space to another intermediate space (sometimes called a 'latent space'). Thereafter, the decoder by contrast has the complementary function of mapping from the latent space to another target space. The generator is segmented into sequential repeating sets of grouped "modules". Within each module of the encoder portion are 2D Convolution Function-Batch Normalisation function and Rectified Linear Unit (ReLU) activation functions whilst the decoder portions modules consist of Depth Concatenation, Transposed 2D Convolution function, Batch Normalization function and ReLU activation function. Unlike conventional GAN generator however, the Pix2Pix implementation has additional bridging connections between layers in the encoder and layers in the decoder to form a "U-Net". This ’crosslinking’ provides the ability to pass information across the network whilst avoiding the central bottleneck region inherent in the conventional encoder-decoder networks (Figure 3.9, left) and functionally imposes structural conditions between layers.

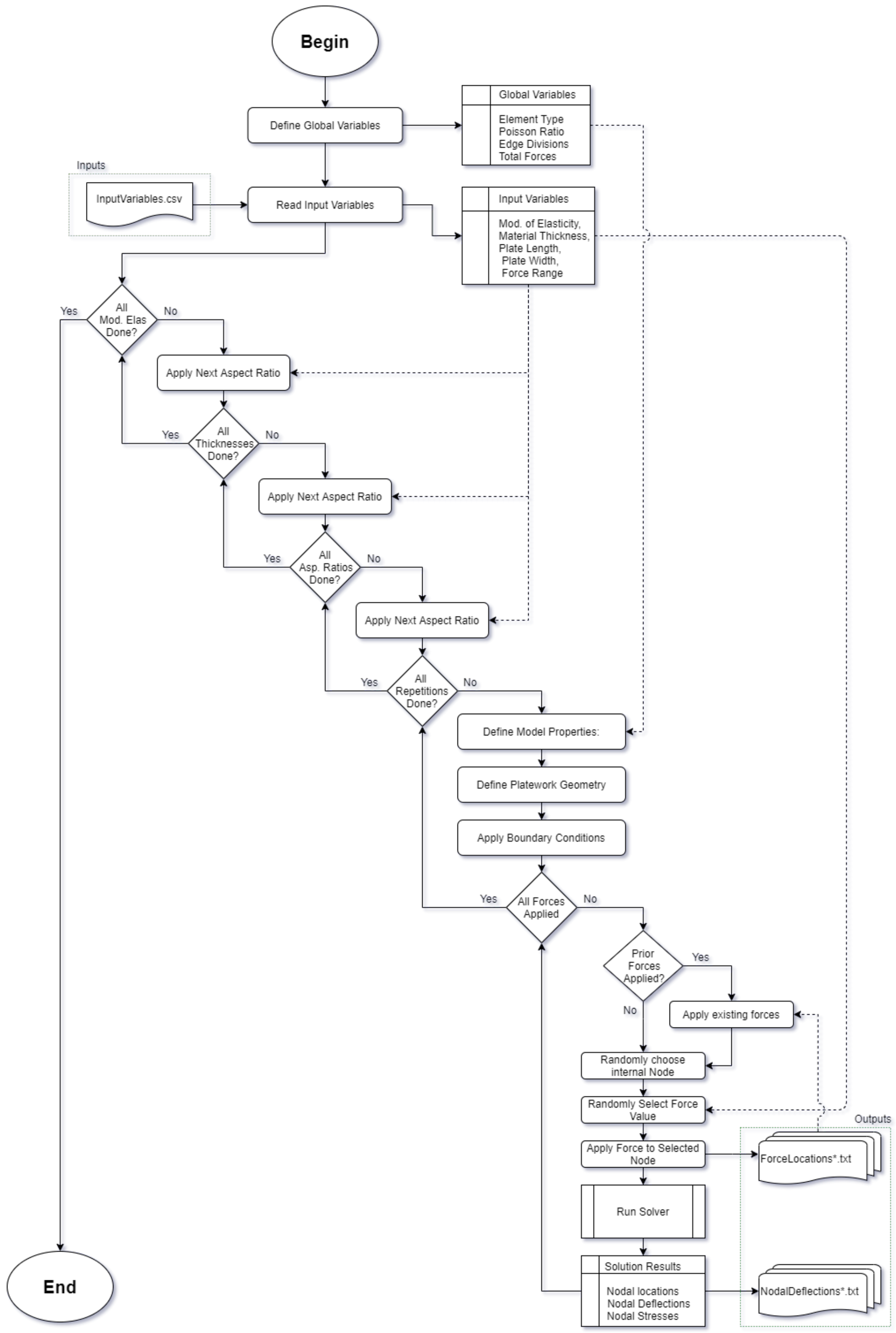


Figure 3.5: Flowchart of FEA Dataset generation process using ANSYS APDL. Created for this publication

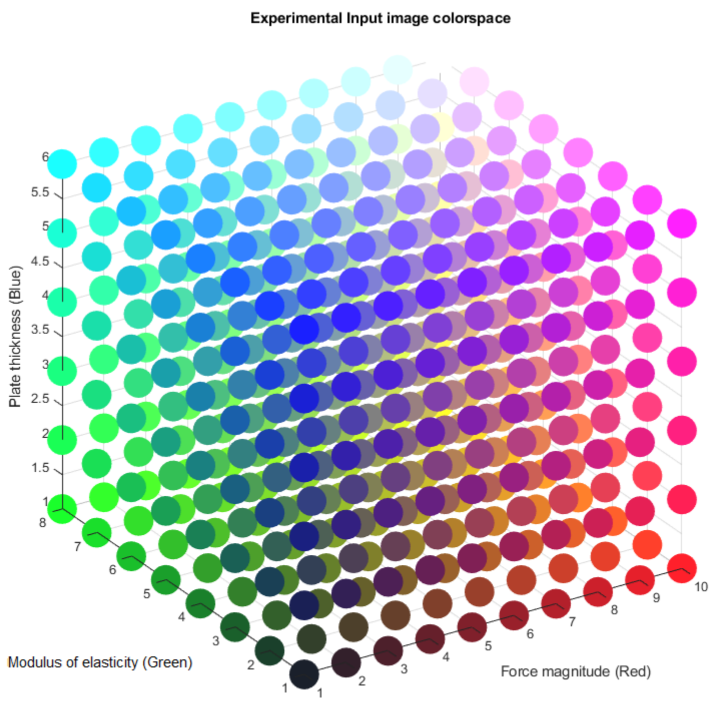


Figure 3.6: Experimental input colorspace. The colorspace which maps input values to RGB pixel channel intensities. For clartiy only the granularity is shown ie 10 force, 8 thickness and 6 modulus of elasticity values

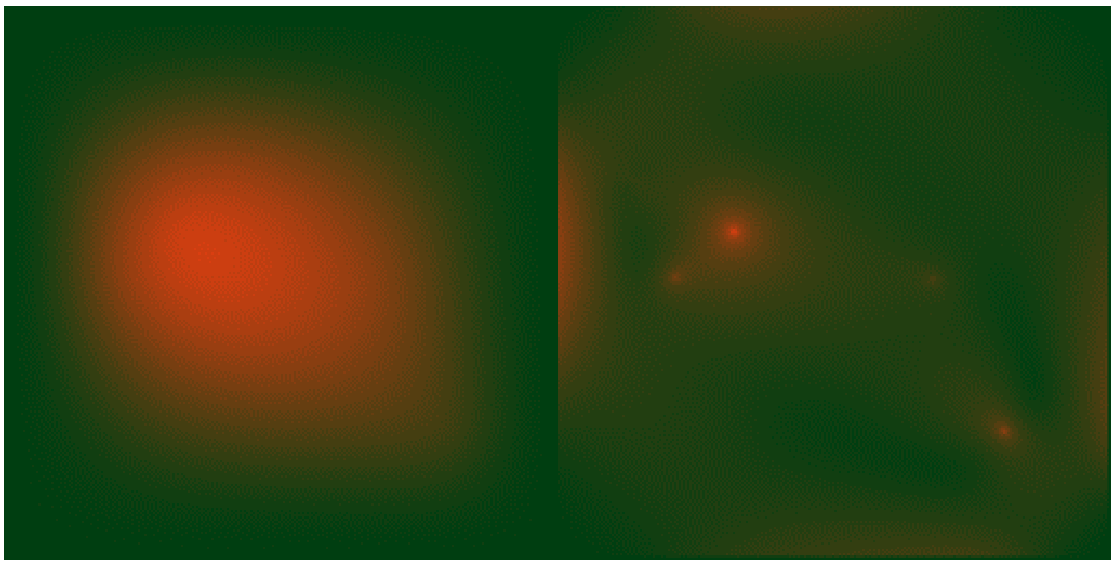


Figure 3.7: Example entry of the processed Paired Image Dataset. Left: Deflection as Input. Right: Corresponding Stresses

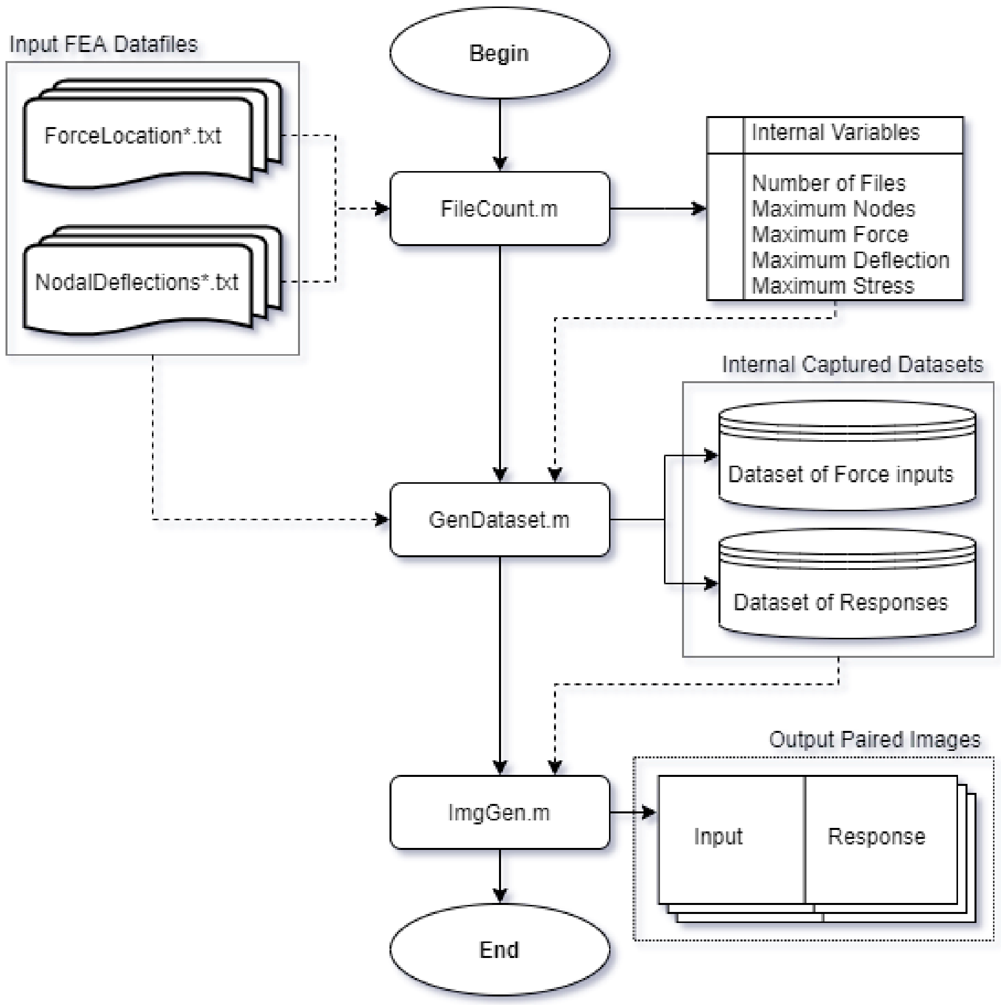


Figure 3.8: Flowchart of Paired Image Dataset creation process in MATLAB.

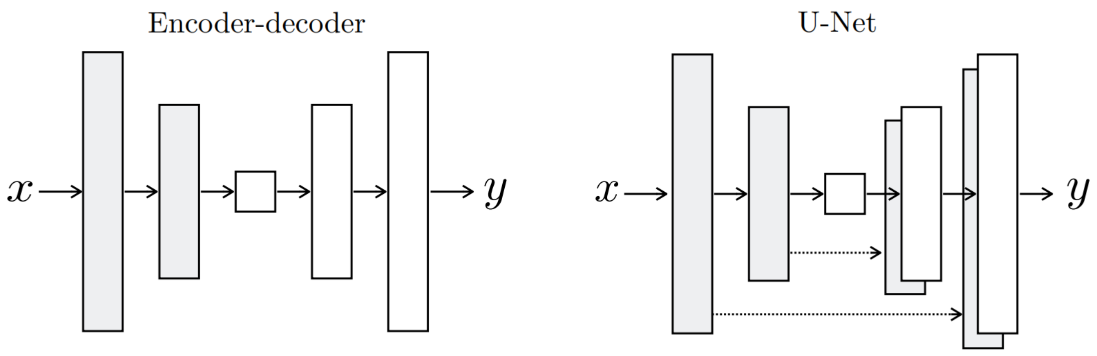


Figure 3.9: conventional GAN vs cGAN architectures. left: conventional encoder-decoder architecture and right: cGAN "U-Net" encoder-decoder with connections between mirrored layers in the encoder and decoder stacks [69]

Table 3.2: cGAN generator layer properties as defined in MATLAB.
'Learnables' refer to parameters which are updated during training.

	Layer Name	**Layer Type**	**Activations**	**Leamables**
1	inputlmage 258x256x3 images	Image Input	256*256*3	
2	Conv2d_1 64 4x4x3, stride[2 2], padding 'same'	Convolution	128*128*64	Weights 4*4*3*64 Bias 1*1*64
3	leaky_re_lu_1 Leaky ReLU with scale 0.3	Leaky ReLU	128*128*64	
4	conv2d_2 123 4x4x64, stride[2 2], padding 'same'	Convolution	64*64*128	Weights 4*4*64*128 Bias 1*1*128
5	batch_normalization_2 Batch normalization with 128 channels	Batch Normalization	64*64*128	Offset 1*1*128 Scale 1*1*128
6	leaky_re_lu_2 Leaky ReLU with scale 0.3	Leaky ReLU	64*64*128	
7	conv2d_3 258 4x4x128, stride[2 2], padding 'same'	Convolution	32*32*256	Weights 4*4*128*256 Bias 1*1*256
8	batch_normalization_3 Batch normalization with 256 channels	Batch Normalization	32*32*256	Offset 1*1*256 Scale 1*1*256
9	leaky_re_lu_3 Leaky ReLU with scale 0.3	Leaky ReLU	32*32*256	
10	conv2d_4 512 4x4x256, stride[2 2], padding 'same'	Convolution	16*16*512	Weights 4*4*256*512 Bias 1*1*512
11	batch_normalization_4 Batch normalization with 512 channels	Batch Normalization	16*16*512	Offset 1*1*512 Scale 1*1*512
12	leaky_re_lu_4 Leaky ReLU with scale 0.3	Leaky ReLU	16*16*512	
13	conv2d_5	Convolution	8*8*512	Weights 4*4*512*512

Table 3.2 continued from previous page

	Layer Name	Layer Type	Activations	Leamables
	512 4x4x512, stride[2 2], padding ’same’			Bias 1*1*512
14	batch_normalization_5 Batch normalization with 512 channels	Batch Normalization	8*8*512	Offset 1*1*512 Scale 1*1*512
15	leaky_re_lu_5 Leaky ReLU with scale 0.3	Leaky ReLU	8*8*512	
16	conv2d_6 512 4x4x512, stride[2 2], padding ’same’	Convolution	4*4*512	Weights 4*4*512*512 Bias 1*1*512
17	batch_normalization_6 Batch normalization with 512 channels	Batch Normalization	4*4*512	Offset 1*1*512 Scale 1*1*512
18	leaky_re_lu_6 Leaky ReLU with scale 0.3	Leaky ReLU	4*4*512	
19	conv2d_7 512 4x4x512, stride[2 2], padding ’same’	Convolution	2*2*512	Weights 4*4*512*512 Bias 1*1*512
20	batch_normalization_7 Batch normalization with 512 channels	Batch Normalization	2*2*512	Offset 1*1*512 Scale 1*1*512
21	leaky_re_lu_7 Leaky ReLU with scale 0.3	Leaky ReLU	2*2*512	
22	conv2d_8 512 4x4x512, stride[2 2], padding ’same’	Convolution	1*1*512	Weights 4*4*512*512 Bias 1*1*512
23	batch_normalization_8 Batch normalization with 512 channels	Batch Normalization	1*1*512	Offset 1*1*512 Scale 1*1*512
24	leaky_re_lu_8 Leaky ReLU with scale 0.3	Leaky ReLU	1*1*512	
25	conv2d_transpose_1 512 4x4x512, stride[2 2], cropping ’same’	Transposed Convolution	2*2*512	Weights 4*4*512*512 Bias 1*1*512
26	batch_normalization_9 Batch normalization with 512 channels	Batch Normalization	2*2*512	Offset 1*1*512 Scale 1*1*512

Table 3.2 continued from previous page

	Layer Name	Layer Type	Activations	Leamables
27	dropout 50% dropout	Dropout	2*2*512	
28	re_lu_1 ReLU	ReLU	2*2*512	
29	concatenate Depth concatenation of 2 inputs	Depth concatenation	2*2*1024	
30	conv2d_transpose_2 512 4x4x1024, stride[2 2], cropping 'same'	Transposed Convolution	4*4*512	Weights 4*4*512*1024 Bias 1*1*512
31	batch_normalization_10 Batch normalization with 512 channels	Batch Normalization	4*4*512	Offset 1*1*512 Scale 1*1*512
32	dropout_1 50% dropout	Dropout	4*4*512	
33	re_lu_2 ReLU	ReLU	4*4*512	
34	concatenate_1 Depth concatenation of 2 inputs	Depth concatenation	4*4*1024	
35	conv2d_transpose_3 512 4x4x1024, stride[2 2], cropping 'same'	Transposed Convolution	8*8*512	Weighs 4*4*512*1024 Bias 1*1*512
36	batch_normalization_11 Batch normalization with 512 channels	Batch Normalization	8*8*512	Offset 1*1*512 Scale 1*1*512
37	dropout_2 50% dropout	Dropout	8*8*512	
38	re_lu_3 ReLU	ReLU	8*8*512	
39	concatenate_2 Depth concatenation of 2 inputs	Depth concatenation	8*8*1024	
40	conv2d_transpose_4	Transposed Convolution	16*16*512	Weights 4*4*512*1024

Table 3.2 continued from previous page

	Layer Name	Layer Type	Activations	Leamables
	512 4x4x1024, stride[2 2], cropping 'same'			Bias 1*1*512
41	batch_normalization_12 Batch normalization with 512 channels	Batch Normalization	16*16*512	Offset 1*1*512 Scale 1*1*512
42	re_lu_4 ReLU	ReLU	16*16*512	
43	concatenate_3 Depth concatenation of 2 inputs	Depth concatenation	16*16*1024	
44	conv2d_transpose_5 258 4x4x1024, stride[2 2], cropping 'same'	Transposed Convolution	32*32*256	Weights 4*4*256*1024 Bias 1*1*256
45	batch_normalization_13 Batch normalization with 256 channels	Batch Normalization	32*32*256	Offset 1*1*256 Scale 1*1*256
46	re_lu_5 ReLU	ReLU	32*32*256	
47	concatenate_4 Depth concatenation of 2 inputs	Depth concatenation	32*32*512	
48	conv2d_transpose_6 128 4x4x512, stride[2 2], cropping 'same'	Transposed Convolution	64*64*128	Weights 4*4*128*512 Bias 1*1*128
49	batch_normalization_14 Batch normalization with 128 channels	Batch Normalization	64*64*128	Offset 1*1*128 Scale 1*1*128
50	re_lu_6 ReLU	ReLU	64*64*128	
51	concatenate_5 Depth concatenation of 2 inputs	Depth concatenation	64*64*256	
52	conv2d_transpose_7 64 4x4x256, stride [2 2], cropping 'same'	Transposed Convolution	128*128*64	Weights 4*4*64*256 Bias 1*1*64
53	batch_normalization_15 Batch normalization with 64 channels	Batch Normalization	128*128*64	Offset 1*1*64 Scale 1*1*64

Table 3.2 continued from previous page

	Layer Name	Layer Type	Activations	Leamables
54	re_lu_7 ReLU	ReLU	128*128*64	
55	concatenate_6 Depth concatenation of 2 inputs	Depth concatenation	128*128*128	
56	conv2d_transpose_8 3 4x4x128, stride[2 2], cropping 'same'	Transposed Convolution	256*256*3	Weights 4*4*3*128 Bias 1*1*3

3.3.1.2 Discriminator

The discriminator, D, is a CNN which Isola et al. refer to as a 'patchGAN' classifier. The difference between the patchGAN and a conventional CNN is that it produces an $N \times N$ array as opposed to a single scalar vector of classifications. This $N \times N$ array maps to patches from the input images. The discriminator sequentially assesses the 'real' or 'fake' state of the generated candidate images across these patches whilst also doing so convolutionally across the entire image. The responses are averaged as the ultimate output of network D. Isola et al also demonstrate that N can be significantly smaller than the full image dimension and still generate high quality results, consequently the discriminator layers implemented here are defined to achieve the recommended "patchsize" of 70×70 pixels. Architecturally the discriminator is comprised of an input layer which accepts the paired-image files overlayed on one another followed by sequential "modules" of 2D convolution, leaky ReLu and batch normalisation layers. The layer properties are provided in Table 3.3 and the network architectures as they are structured within Matlab are provided in Figure 3.10.

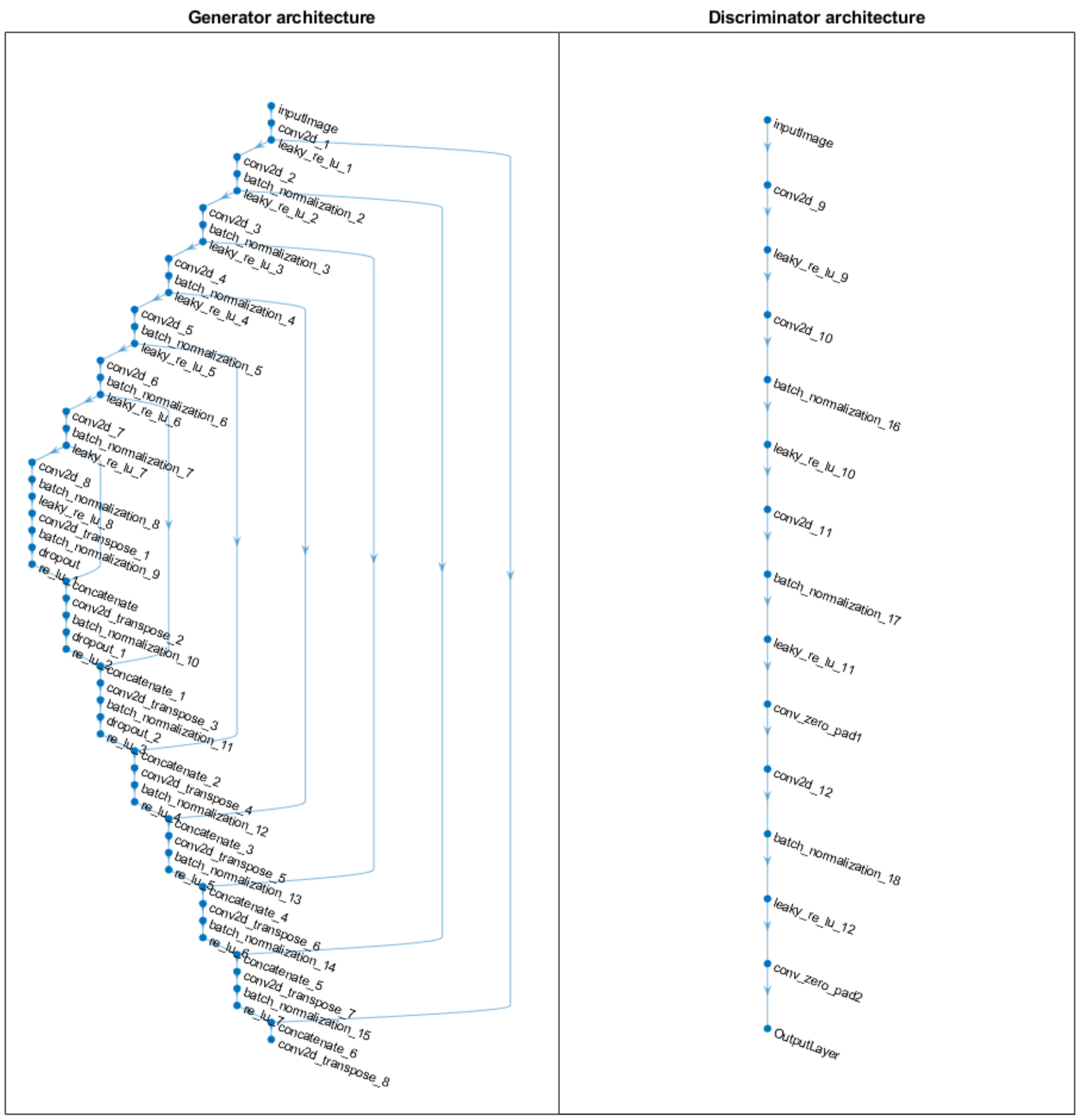


Figure 3.10: Layer structures of the cGANs generator and discriminator network architectures: Left: Generator network with bypass "U-net" bridge connections. Right: Discriminator network

Table 3.3: cGAN discriminator layer properties as defined in MATLAB.
'Learnables' refer to parameters which are updated during training.

	Name	Type	Activations	Leamables
1	inputlmage	Image Input	256*256*6	
	258x256x6[3] images			
2	conv2d_9	Convolution	128*128*64	Weights 4*4*6*64
	64 4x4x6, stride [2 2], padding 'same'			Bias 1*1*64
3	leaky_re_lu_9	Leaky ReLU	128*128*64	
	Leaky ReLU with scale 0.3			
4	conv2d_10	Convolution	64*64*128	Weights 4*4*64*128
	123 4x4x64, stride [2 2], padding 'same'			Bias 1*1*128
5	batch_normalization_16	Batch Normalization	64*64*128	Offset 1*1*128
	Batch normalization with 128 channels			Scale 1*1*128
6	leaky_re_lu_10	Leaky ReLU	64*64*128	
	Leaky ReLU with scale 0.3			
7	conv2d_11	Convolution	32*32*256	Weights 4*4*128*256

[3]The input is the combined "generated" and "target" images overlayed hence their common 256x256 dimension and combined 6 color channel depth

Table 3.3 continued from previous page

	Name	Type	Activations	Leamables
	258 4x4x128, stride [2 2], padding 'same'			Bias 1*1*256
8	batch_normalization_17	Batch Normalization	32*32*256	Offset 1*1*256
	Batch normalization with 256 channels			Scale 1*1*256
9	leaky_re_lu_11	Leaky ReLU	32*32*256	
	Leaky ReLU with scale 0.3			
10	conv_zero_pad1	Convolution	34*34*512	Weights 1*1*256*512
	512 1x1x256, stride [1 1], padding [1111]			Bias 1*1*512
11	conv2d_12	Convolution	31*31*512	Weights 4*4*512*512
	512 4x4x512, stride [1 1], padding [0000]			Bias 1*1*512
12	batch_normalization_18	Batch Normalization	31*31*512	Offset 1*1*512
	Batch normalization with 512 channels			Scale 1*1*512
13	leaky_re_lu_12	Leaky ReLU	31*31*512	
	Leaky ReLU with scale 0.3			
14	conv_zero_pad2	Convolution	33*33*31	Weights 1*1*512*31
	31 1x1x512, stride [1 1], padding [1111]			Bias 1*1*31

Table 3.3 continued from previous page

	Name	Type	Activations	Leamables
15	OutputLayer	Convolution	30*30*1	Weights 4*4*31
	1 4x4x31, stride [1 1], padding [0000]			Bias 1*1

3.3.2 Loss Functions and Gradients

The objective of the cGAN can be expressed as

$$\begin{aligned}\mathscr{L}_{cGAN}(G,D) =&\mathbb{E}_{x,y}[\log D(x,y)]+\\ &\mathbb{E}_{x,z}[\log(1-D(x,G(x,z))]\end{aligned} \tag{3.1}$$

where G tries to minimize this objective against an adversarial D that tries to maximize it, i.e.

$$G^{*} = \arg\min_{G}\max_{D}\mathscr{L}_{cGAN}(G,D) \tag{3.2}$$

Additionally, the Pix2Pix cGAN makes inclusion of the Mean Absolute Error loss, L_1, imposed on the generator so as to be near the ground truth output (in an L_1 sense) in addition to the conventional requirement of convincing the discriminator the output is real. This L_1 requirement is reflected in Equation 3.3

$$\mathscr{L}_{L1}(G) = \mathbb{E}_{x,y,z}\left[\|y-G(x,z)\|_1\right] \tag{3.3}$$

The final objective of the cGAN is summarised as:

$$G^{*} = \arg\min_{G}\max_{D}\mathscr{L}_{cGAN}(G,D)+\lambda\mathscr{L}_{L1}(G) \tag{3.4}$$

In practice, calculating the losses and utilizing them to change the network weighs and biases is done sequentially:

3.3.2.1 Generator Losses:

1. The generator loss, GAN_{loss} , is calculated as the sigmoid Cross-Entropy loss of the generated images and an array of ones. With Cross-Entropy loss given by Equation 3.5:

$$CE = -\sum_{i}^{C} t_i log(s_i) \tag{3.5}$$

Where t_i and s_i are the ground truth and the CNN scores for each class in C classes. For the cGAN with Binary classificaton ("Real" and "Fake") Equation 3.5 becomes:

$$CE = -\sum_{i=1}^{C'=2} t_i log(s_i) \tag{3.6}$$

For $C = 2$. t_1 [0,1] and s_1 are the groundtruth and the score for C_1, and $t_2 = 1 - t_1$ and $s_2 = 1 - s_1$ are the groundtruth and the score for C_2. Hence Equation 3.6 becomes

$$CE = -t_1 log(s_1) - (1 - t_1) log(1 - s_1) \tag{3.7}$$

Since the Sigmoid activation function is applied to the generator outputs Equation 3.7 becomes

$$GAN_{loss} = -t_1 log(s_1) - (1 - t_1) log(1 - s_1) \tag{3.8}$$

where $f(s_i)$ is the Sigmoid function

$$f(s_i) = \frac{1}{1 + e^{-s_i}} \tag{3.9}$$

2. The L$_1$, Mean Absolute Error loss is calculated between the generated image and the target image using Equation 3.10:

$$L_1 = \frac{1}{n}\sum_{i=1}^{n} \left| y_{\text{target}} - y_{\text{generated}} \right| \tag{3.10}$$

where *n* is the number of elements (pixels) per generated / target image and *y* the channel metric of interest (here deflection or stress).

3. The total generator loss is calculated in Equation 3.11:

$$Gen_{Tot} = GAN_{loss} + \lambda \times \mathrm{L1} \tag{3.11}$$

where λ = 100 as specified in the Pix2Pix paper [69].

.

3.3.2.2 Discriminator losses:

1. $Real_{loss}$ is the sigmoid cross entropy loss of the real images and an array of ones (since these are the real images)

2. $Generated_{loss}$ is the sigmoid cross entropy loss of the generated images and an array of zeros (since these are the fake images)

3. The $Total_{loss}$ is the sum of $Real_{loss}$ and the $Generated_{loss}$.

3.3.2.3 Gradients:

The gradients are applied through backpropogation using the minibatch SGD method and the Adam Solver optimization [56] with the recommended parameters:

- minibatch size of 32 samples
- learning rate of 0.0002
- $\beta 1 = 0.5$
- $\beta 2 = 0.999$

3.4 Training the GAN models

The MATLAB© implementation of the cGAN is carried out for 3 separate experiments:

1. Force $\rightarrow$ Deflection mapping
2. Deflection $\rightarrow$ Stress mapping
3. Force $\rightarrow$ Stress mapping

3.4.1 Matlab Pseudo-code

For the sake of brevity the implemented model training process is described via pseudo-code. The full MATLAB implementation and its component scripts can be found in their entirety in the Appendix B.

1. Read paired-image datasets into MATLAB Image datastore (Imds) file structures:

- 60% to Training Imds
- 30% to Testing Imds
- 10% to Validation Imds

2. Read minibatches of paired-images from the Training Imds
3. Process the paired-images:
    - Split into seperate "input" and "target / response" images
    - Apply random jitter - upscale and crop back down to required dimensions
    - Randomly flip along vertical / horizontal axes[4]
    - Normalise data values between [-1,1][5]
4. Create prediction from "input" image parsed to generator network
5. Compare "real" vs "fake" images using the discriminator:
    - Analyse input image against "ground truth" target image
    - Analyse input image against "prediction" image
    - Determine losses from the above using equations of section subsection 3.3.2
    - Determine gradients from above losses
6. Apply gradients to Generator and Discriminator networks
    - SGD performed by ADAM solver and backpropogated using MATLAB "dl-gradient.m" function call.

[4]The random jitter and random flip functions serve to introduce additional randomness to the dataset and in doing so discourage data overfitting [69]

[5]The normalization is required as the input channel ranges accept only [-1,1] within the architecture of the cGAN

7. Plot iterative results

8. Iterate loop

   - Each run though a mini-batch is an Iteration
   - Each run though all mini-batches from Training Imds is an Epoch
   - Save trained model to File at set "SaveFrequency" intervals.

3.5 Validate Trained Model

Typically GAN outputs are notoriesly difficult to validate due to the abstract or subjective nature of the subject matter they are called upon to generate, moreso when plausibilty to the observer is frequently the end goal [69]. For our objective purposes here however accuracy of prediction is paramount if we are to consider potential for deployment and useablility in a real-world setting. In a manner not too dissimilar to the "real"-"fake" judgement process of the disciminator during training we validate each generated result against ground truth by several metrics:

Direct Absolute Error, *DAE* - The Absolute value of the Difference between Target and Generated results.

$$DAE = \left| y_{\text{target}} - y_{\text{generated}} \right| \tag{3.12}$$

Relative error, E_{rel} - Ratio of the DAE magnitude to target value magnitude.

$$E_{rel} = \frac{DAE}{Target} \tag{3.13}$$

Relative Percent Difference, *RPD* - As there are regions wherein the target and generated results have zero values, instances of the previous metrics are indeterminate

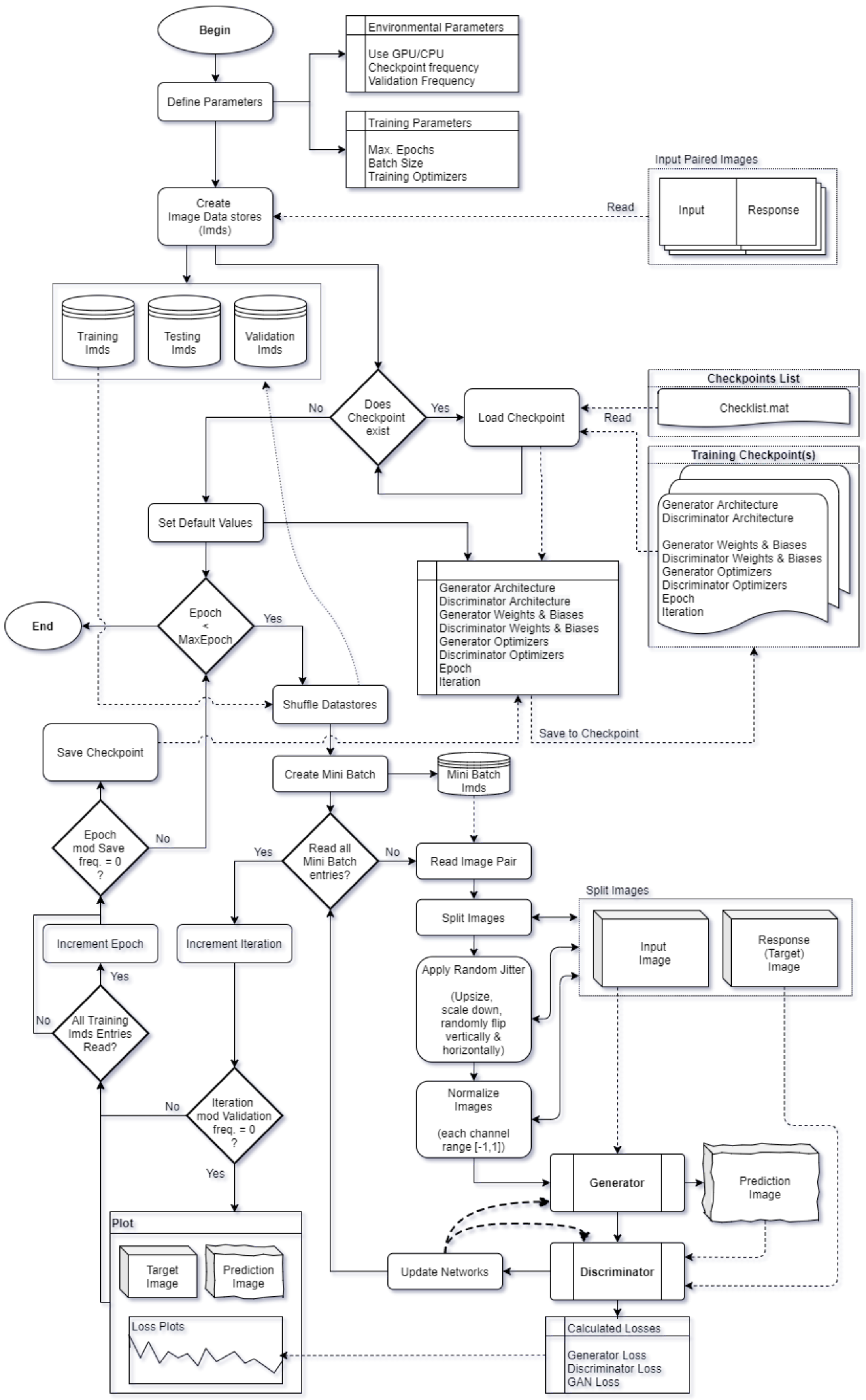


Figure 3.11: MATLAB cGAN implementation's training process flowchart. Chart created for this publication

or undefined, hence we also look at the percent variation between the target and generated results.

$$RPD = \frac{\left|y_{\text{target}} - y_{\text{generated}}\right|}{\left(y_{\text{target}} + y_{\text{generated}}\right) \div 2} = \frac{DAE}{\left(y_{\text{target}} + y_{\text{generated}}\right) \div 2} \tag{3.14}$$

These are calculated for the overall images (which results in 2-Dimensional plots), the averages over the entire result, and at the locations of the maximum input values. Furthermore to assess the changes the errors are calculated and tracked across every saved Epoch of the models in order to gauge improvement (if any).

3.6 Implement Real-time User interface

An GUI application was designed and implemented in MATLAB in order to allow for predictions to be generated for user inputs from trained models. Recall that the cGAN models are trained on discrete values of force, F, thickness, t and modulus of elasticity, E, with pixel values that can be represented by a colorspace as is reflected in Figure 3.6. The applications GUI however allows for input image pixel values to be created using any rational values within the upper and lower bounds of the training variables domains, namely:

- Forces, $F \in [0, 50000N]$
- Plate thickness, $t \in [2, 30mm]$
- Modulus of Elasticity, $E \in [235, 960GPa]$

This allows for far more granularity in input colourspace as shown by Figure 3.12. Ideally should the model have adequately "learnt" the workings of the training data domain we hope to be able to return reliable results for any input within this domain.

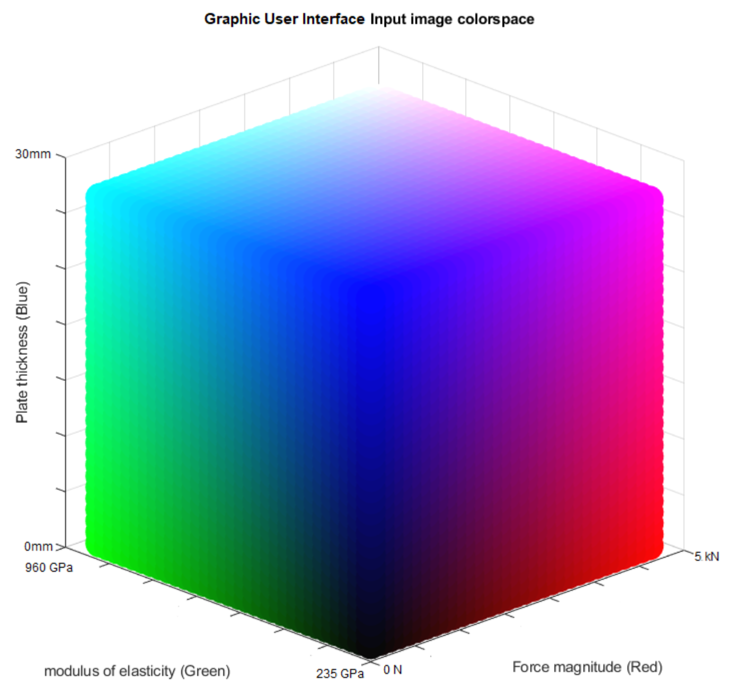
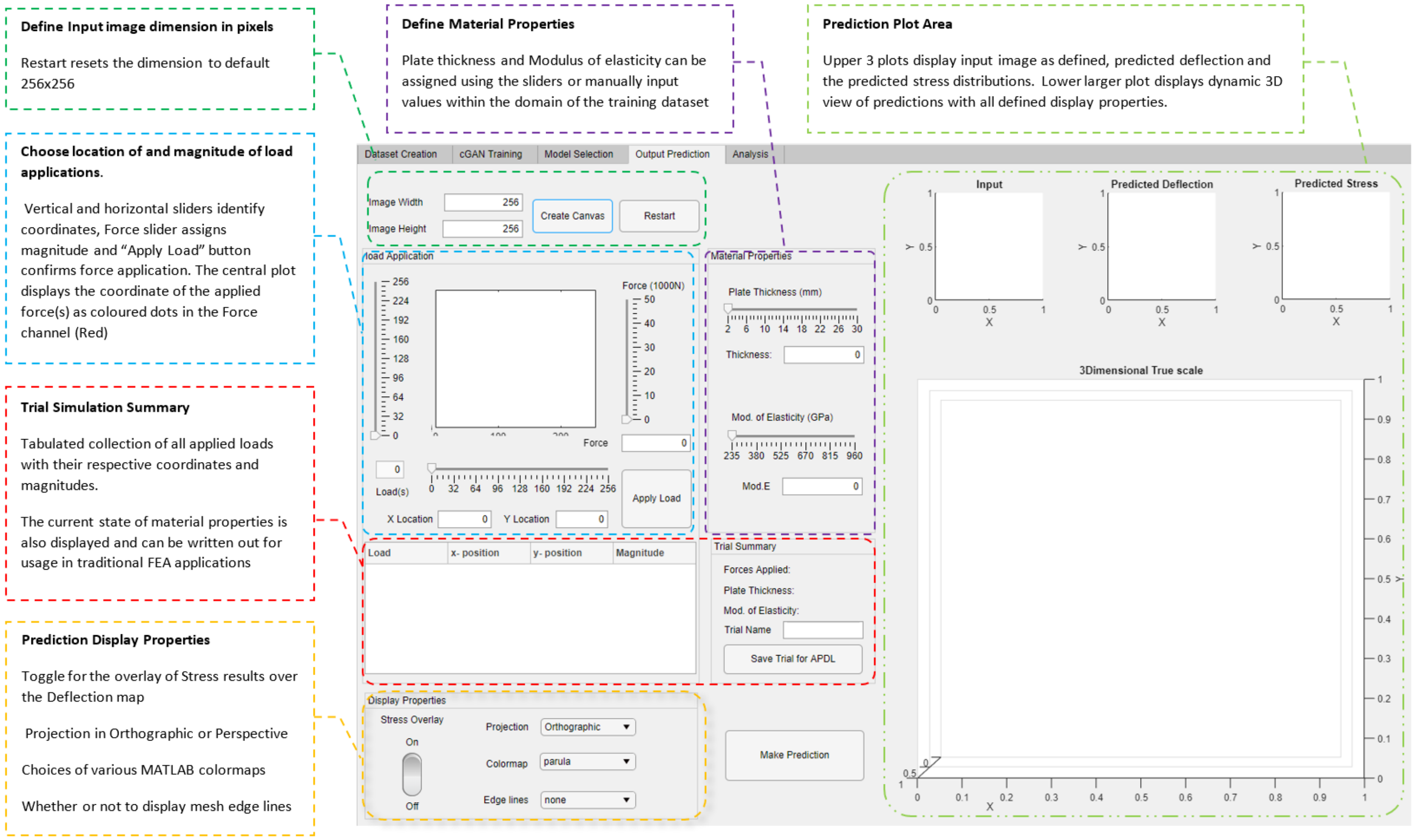


Figure 3.12: GUI application input colorspace. Any rational value within the bounds of each variables axis can be utilized allowing for finer input granularity.

Furthermore trained models can be selectively loaded from training checkpoints. The app parses the user input to the selected cGAN models and outputs the predictions. The input, predicted deflections and stresses as well as a three dimensional plot visualizes the true-scale deflections with the overlayed stress values (if desired). Additionally, it is also possible to write the input trials parameters to file for later evaluation against traditional FEM.


Define Input image dimension in pixels

Restart resets the dimension to default 256x256

Define Material Properties

Plate thickness and Modulus of elasticity can be assigned using the sliders or manually input values within the domain of the training dataset

Prediction Plot Area

Upper 3 plots display input image as defined, predicted deflection and the predicted stress distributions. Lower larger plot displays dynamic 3D view of predictions with all defined display properties.

Choose location of and magnitude of load applications.

Vertical and horizontal sliders identify coordinates, Force slider assigns magnitude and "Apply Load" button confirms force application. The central plot displays the coordinate of the applied force(s) as coloured dots in the Force channel (Red)

Trial Simulation Summary

Tabulated collection of all applied loads with their respective coordinates and magnitudes.

The current state of material properties is also displayed and can be written out for usage in traditional FEA applications

Prediction Display Properties

Toggle for the overlay of Stress results over the Deflection map

Projection in Orthographic or Perspective

Choices of various MATLAB colormaps

Whether or not to display mesh edge lines

Figure 3.13: Breakdown of the primary functions and usage of the GUI developed for end user input Predictions

4 Experimental Results

This chapter presents and discusses the results of the three primary experiments carried out with the cGAN architecture implemented as outlined in chapter 3.

4.1 Progressive Training Progress

With reference to Figure 4.1, we note that after a slow start roughly around the 45^{th} epoch or 600^{th} iteration, the discriminator score rises whilst conversely, the generator score decreases. This appears as an indication that the discriminator has more confidently been able to asses the generated result as 'fake' up to this point. At approximately the 1440^{th} iteration the cGAN has reached the point at which the overall losses are at their maximum. Both the generator and discriminator losses increase in step with one another which indicates that the discriminator is having increased difficulty in distinguishing 'fake' from 'real' in as much as the generator is producing mostly believable results. From here onward both networks advance in step with one another with learning changes reflected in finer details. These progressive improvements are reflected further in Figure 4.3,Figure 4.12 and Figure 4.19.

4.2 Experiment 1 - Force to Deflection mapping

The inputs to the Force$\rightarrow$Deflection mapping experiment are the 6000 paired images of input forces and target deflections. The input mappings are reflected in Figure 4.2. The training was undertaken for 200 epochs with the model being saved to checkpoint every 4 epochs.

4.2.1 Validation Results

Randomly selected inputs from the validation dataset are run through each of the saved checkpoints' trained models. As a preliminary visual inspection, an array of predictions is produced for each of these models. From the images (see Figure 4.3) it is possible to subjectively note the gradual improvement of the predictions as the model trains. Objectively, the generated results are measured against ground truth using the equations specified in Equation 3.12 through Equation 3.14. Figure 4.4 illustrates

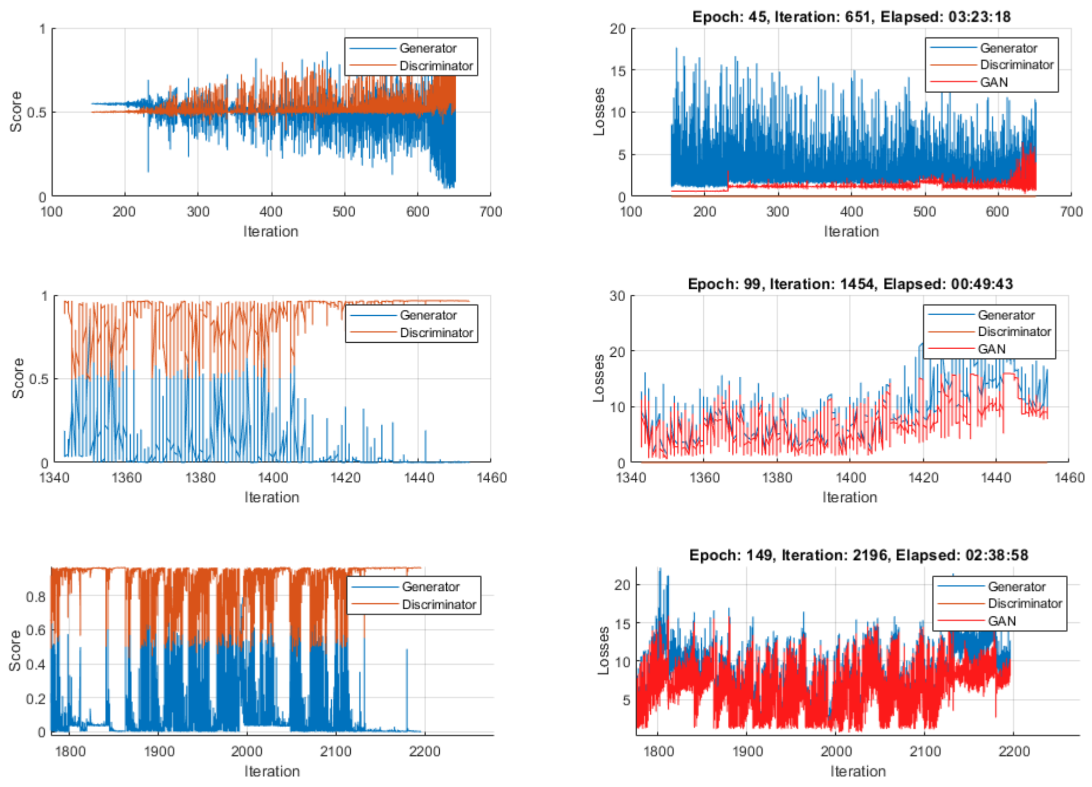


Figure 4.1: Model training progress showing Generator and Discriminator losses. Top: Scores, losses and random validation output for the first 650 iterations. Middle: snippet of training between 1340 and 1450 iterations. Bottom: Training progress between 1750 and 2200 iterations. The eventual alignment of Generator losses with that of the Discriminator indicates the improvement in fooling of the discriminator and corresponds with improved predictive accuracy.

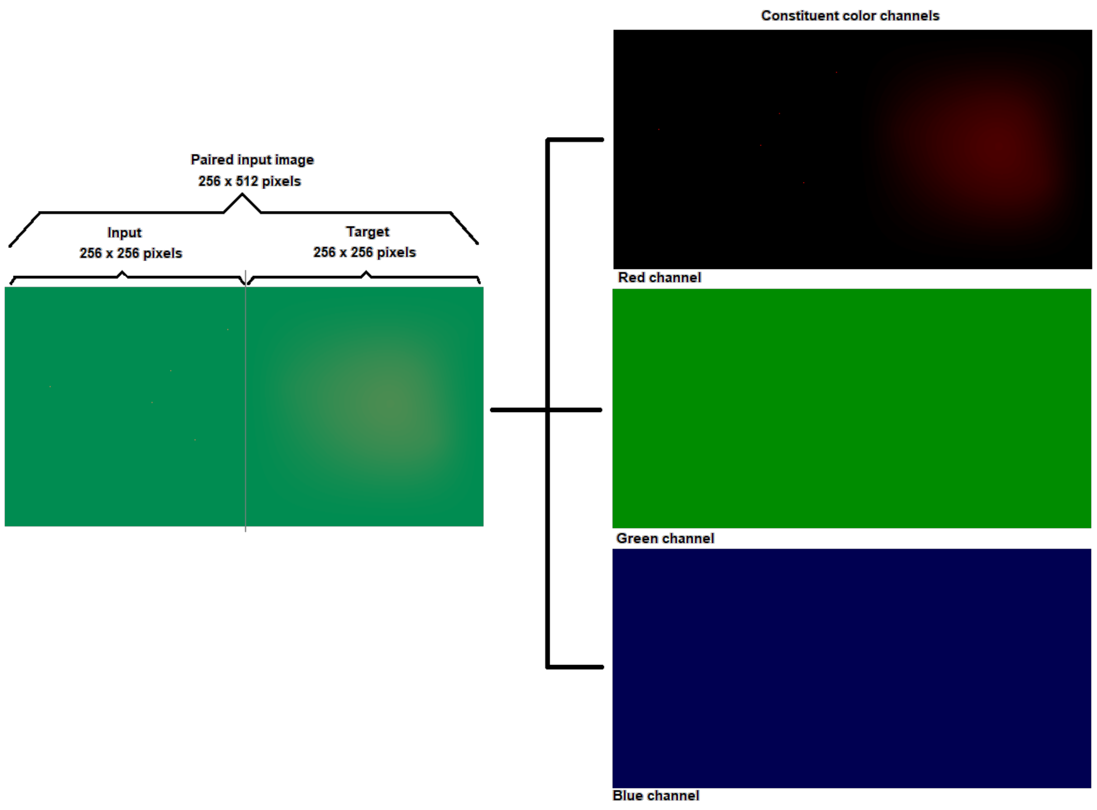


Figure 4.2: Color channel decomposition for Experiment 1. Single sample of 256 × 512 .png paired image dataset reflecting the color channel mapping: Red channel for input force(s) and response deflection magnitudes. Green and blue for thickness and modulus of elasticity respectively.

these outputs across the entirety of the prediction imagespace for a single example input whilst Figure 4.5 presents numerical plots for the error metrics for three random samples. As is evident from these plots the errors gradually converge towards the ground truth values - The results after 200 Epochs fall between a 10-20% error in terms of RPD and below 5% for DAE.

By gathering the error metrics across the *entire* validation dataset we plot the distribution of errors as they progress across all model epochs. This is reflected by Figure 4.10. Evidently, the the mean error for the maximum values (whichs strongly correlate to the location of force application) rapidly fall between 0 and 5% by the final

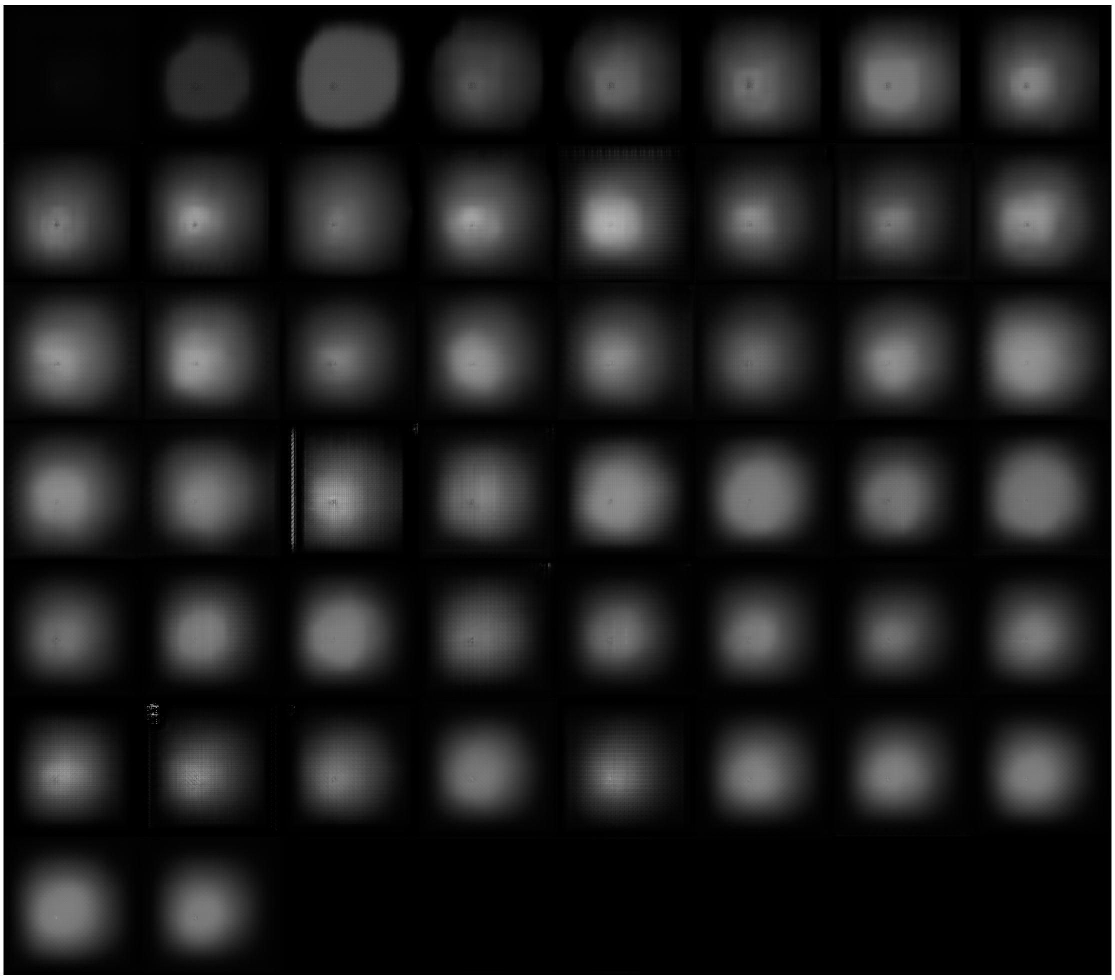


Figure 4.3: Generated predictions for sequential saved models for Experiment 1. 50 saved models which run from left to right and top to bottom. Note how the finer details emerge with continued training resulting in a smoother gradient.

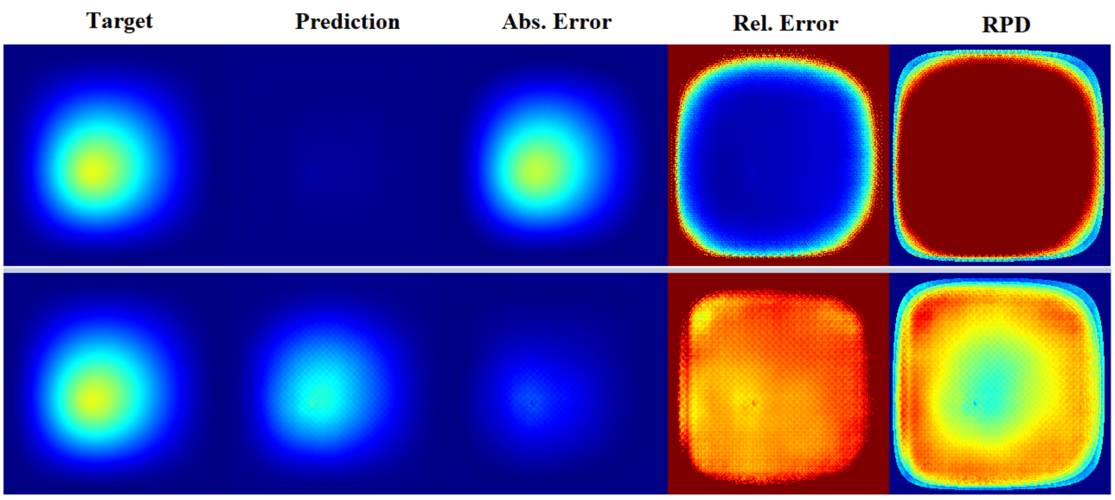


Figure 4.4: Experiment 1 validation results for fourth (top) vs final model Epoch (bottom). Outputs of the validation error metrics across a sample generated prediction. Note the transition of Absolute Error from essentially representing the Target to almost no error.

saved model (200th epoch).

4.3 Experiment 2 - Deflection to Stress mapping

The inputs to the Deflection→Stress mapping experiment are the 6000 paired images of input deflections and target stresses. The input mappings are reflected in Figure 4.11. The training was conducted for 200 epochs with the model being saved to checkpoint every 4 epochs.

4.3.1 Validation Results

The preliminary visual inspection of experiment 2 (see Figure 4.12) again illustrates the gradual improvement of the predictions as the model trains. Objectively,Figure 4.13 presents numerical plots for the Validation error metrics for three random samples. As is evident from these plots the errors gradually converge towards the ground truth values - The results once again fall between a 10-20% error in terms of RPD and below 5% for

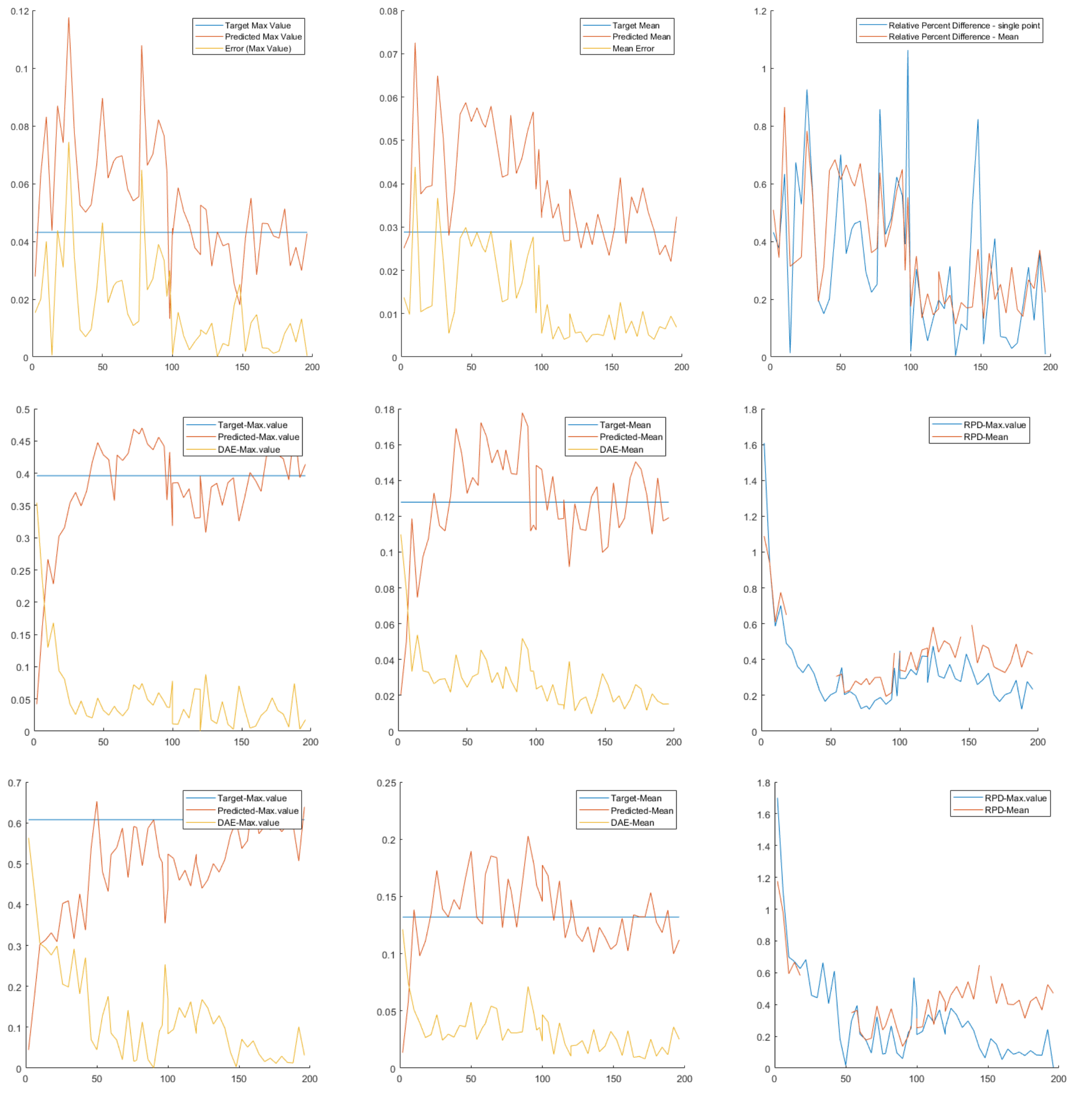


Figure 4.5: Force to Deflection model validation results of random samples across 200 Epochs for Experiement 1

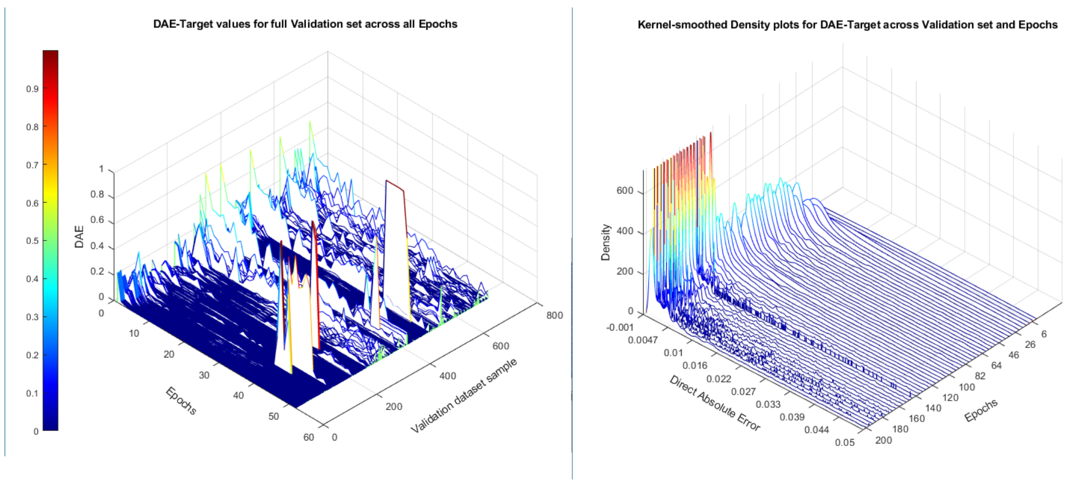


Figure 4.6: Experiment 1: DAE(Target) errors across Epochs. Left:Waterfall of errors for each sample across epochs. Right: Kernel Density plot of all samples per epoch.

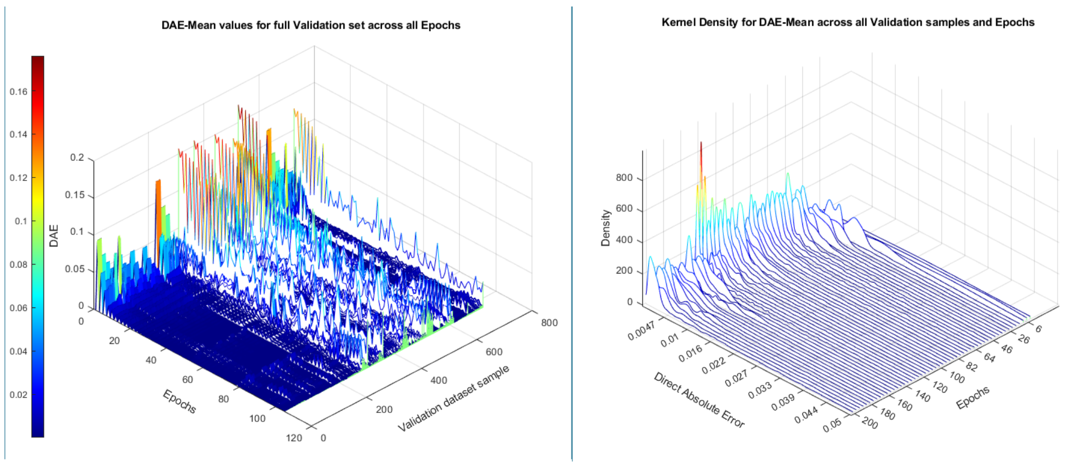


Figure 4.7: Experiment 1: DAE(Mean) errors across Epochs. Left:Waterfall of errors for each sample across epochs. Right: Kernel Density plot of all samples per epoch.

DAE after 200 Epochs.

An examination of the errors over the entirety of the validation set produces Figure 4.14 through Figure 4.17. We can quite reasonably note the diminishing of errors with the increase in training. Furthermore, noting the included waterfall plots of density per epoch we see that the mean errors decrease with further training.

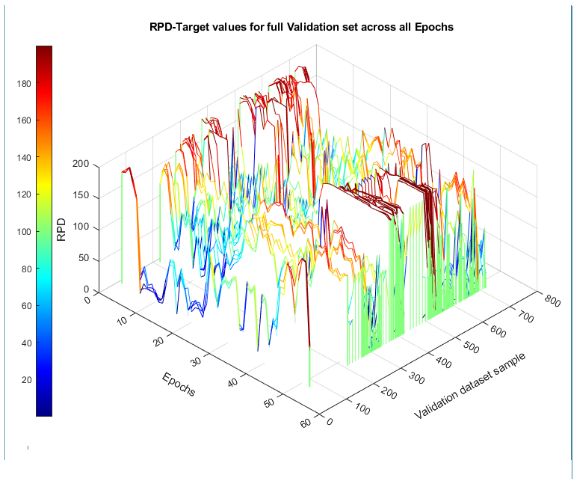


Figure 4.8: Experiment 1: RPD(Target) errors across Epochs. Left:Waterfall of errors for each sample across epochs. Right: Kernel Density plot of all samples per epoch.

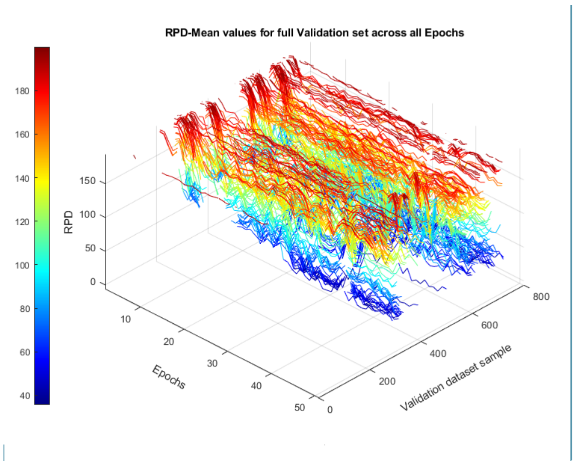


Figure 4.9: Experiment 1: RPD(Mean) errors across Epochs. Left:Waterfall of errors for each sample across epochs. Right: Kernel Density plot of all samples per epoch.

4.4 Experiment 3 - Force to Stress mapping

The inputs to the Force→Stress mapping experiment are the 6000 paired images of input forces and target stresses. The input mapping is reflected in Figure 4.18. For consistency the training was undertaken for 200 epochs with the model being saved to checkpoint every 4 epochs.

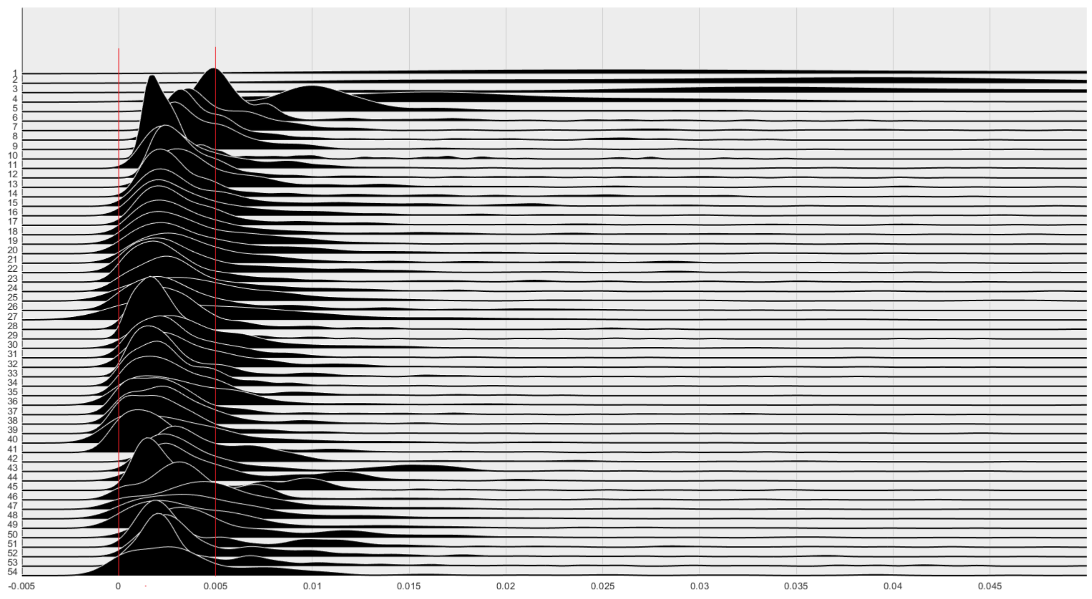


Figure 4.10: Cascade plot of DAE distribution for Experiment 1 across 200 Epochs. Note the rapid drop off of error distribution to between zero and 5%.

4.4.1 Validation Results

In agreement with the prior 2 experiments, experiment 3's preliminary visual inspection (see Figure 4.19) and sample validations Figure 4.20 both subjectively and objectively illustrate the downward progression of prediction error as the generator approaches ground truth accuracy with error values once again falling into the $< 5\%$ range for the 3 random random samples. The more conclusive results reflected in figures Figure 4.22 through Figure 4.25 further illustrate the overall mean error values fall well below 5% when averaged across the entire dataset.

Final mean error values are presented in **??**.

As is evident from the resulting plots in Figure 4.21 the errors gradually converge towards the ground truth values - The results after 200 Epochs fall between a 10-20% error in terms of RPD and below 10% for DAE.

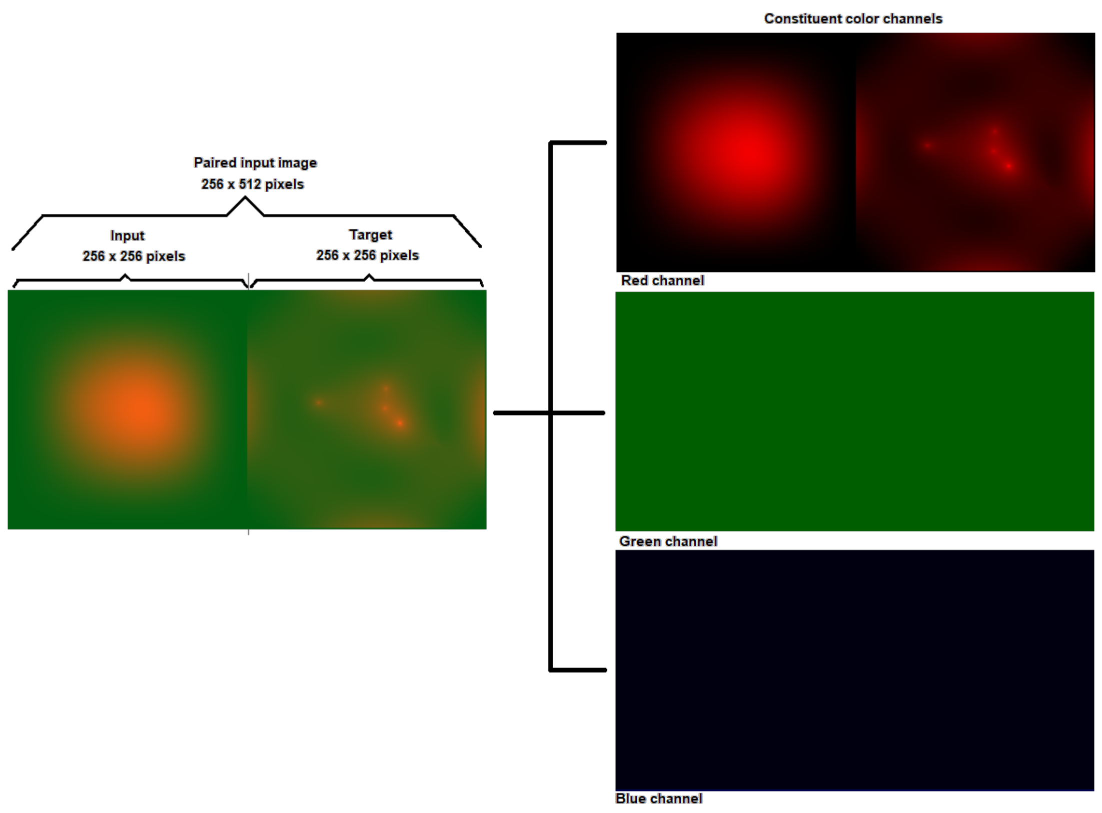


Figure 4.11: Color channel decomposition for Experiment 1. Single sample of 256×512 .png paired image dataset reflecting the color channel mapping: Red channel for input deflection and response stress magnitudes. Green and blue for thickness and modulus of elasticity respectively.

By gathering the error metrics across *all* the validation datasets predictions we plot the distribution of errors across all model epochs. Evidently, the the mean error for the maximum values (generally the location of force application) rapidly falls between 0 and 5% by the 54^{th} saved model (200^{th} epoch).

4.5 Real-time User Predictions

Utlilizing the purpose built MATLAB application the latest versions for the deflection and stress models are loaded. Utilizing the GUI the user is able to define the plate properties as well as the loadings.

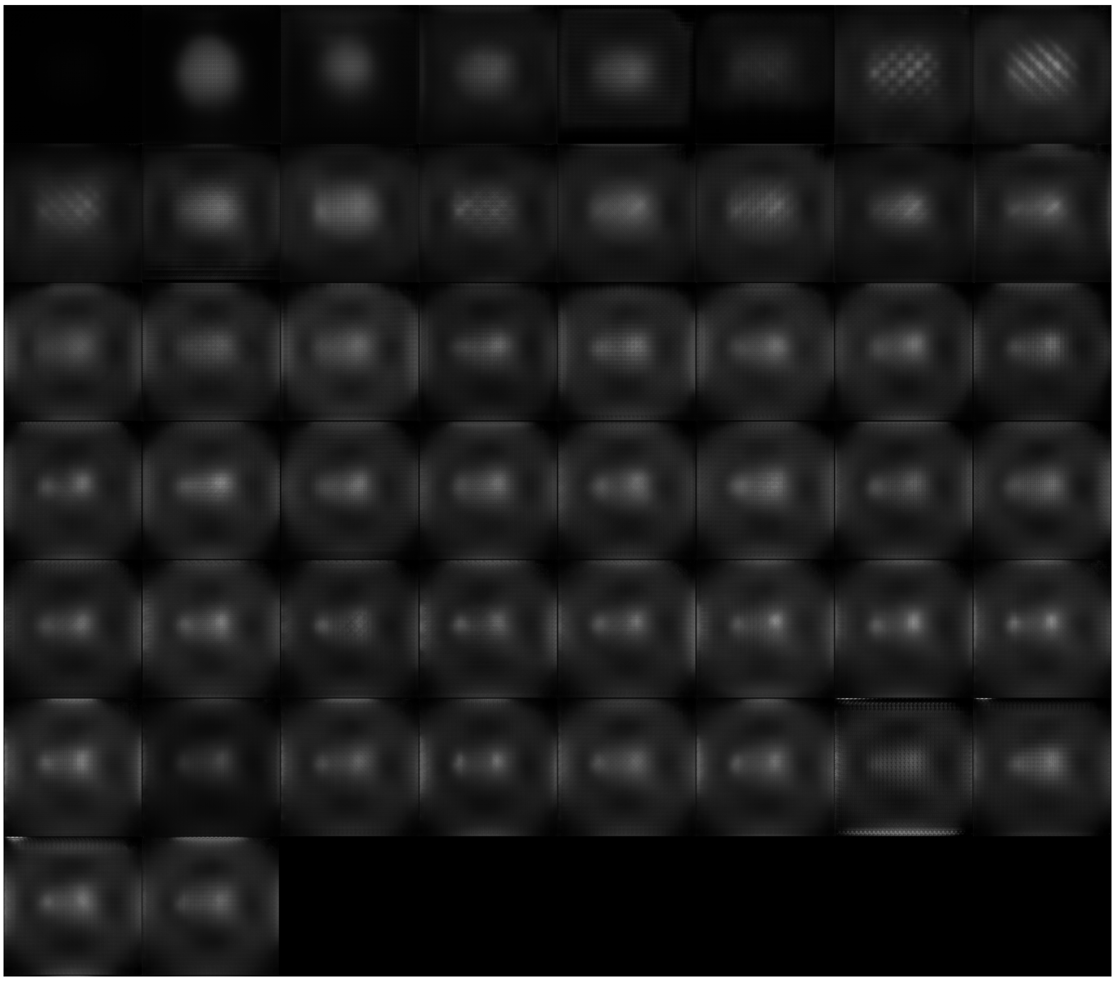


Figure 4.12: Generated predictions for sequential saved models for Experiment 2. 50 saved models which run from left to right and top to bottom. Note how the finer details emerge with continued training resulting in smoother gradients.

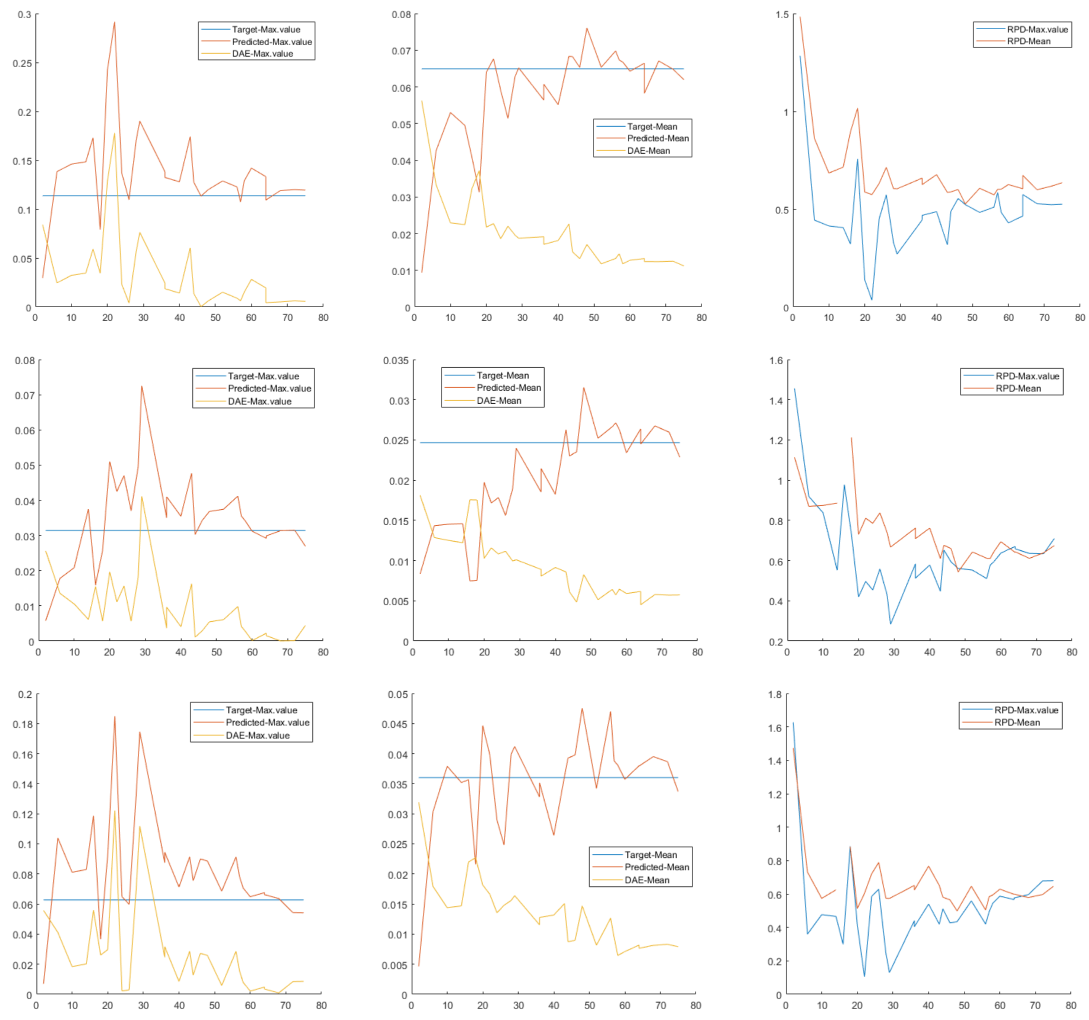


Figure 4.13: Deflection to Stress model validation results of random sample across 200 Epochs

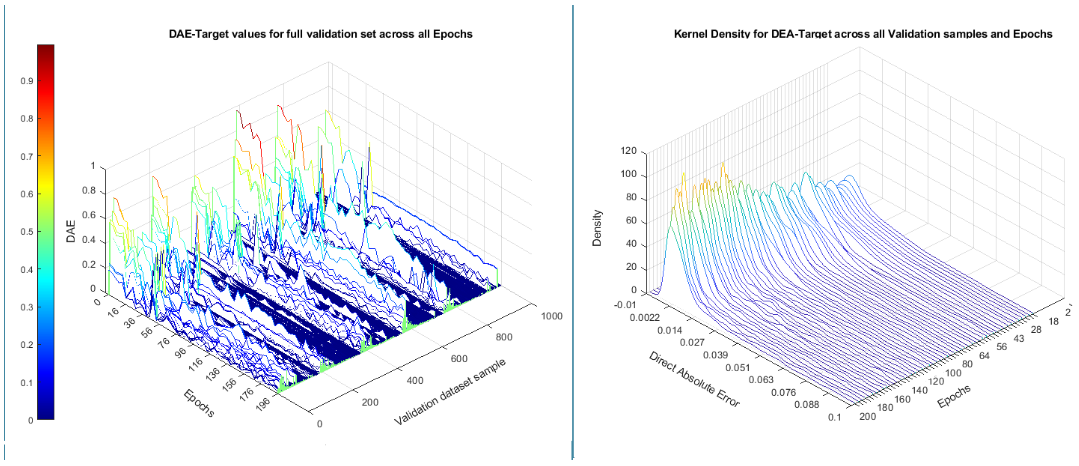


Figure 4.14: Experiment 2: DAE(Target) errors across Epochs. Left:Waterfall of errors for each sample across epochs. Right: Kernel Density plot of all samples per epoch.

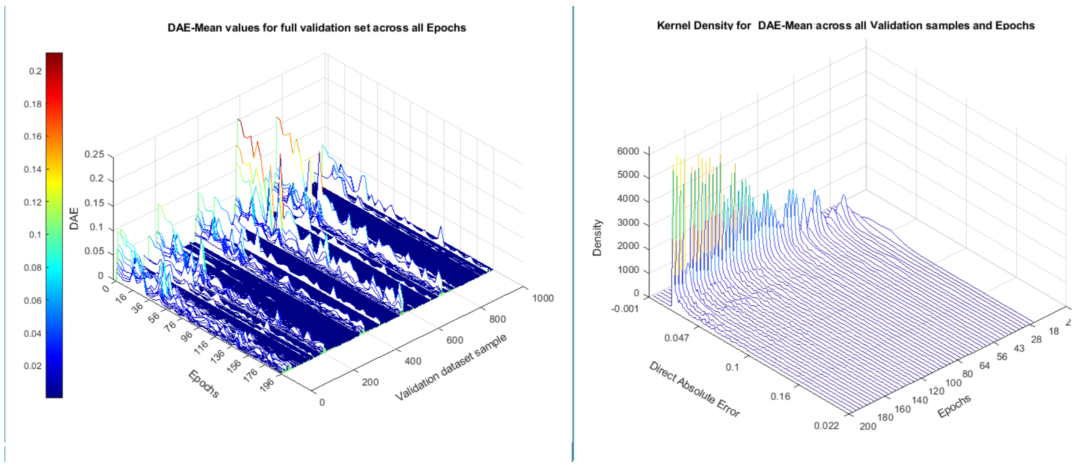


Figure 4.15: Experiment 2: DAE(Mean) errors across Epochs. Left:Waterfall of errors for each sample across epochs. Right: Kernel Density plot of all samples per epoch.

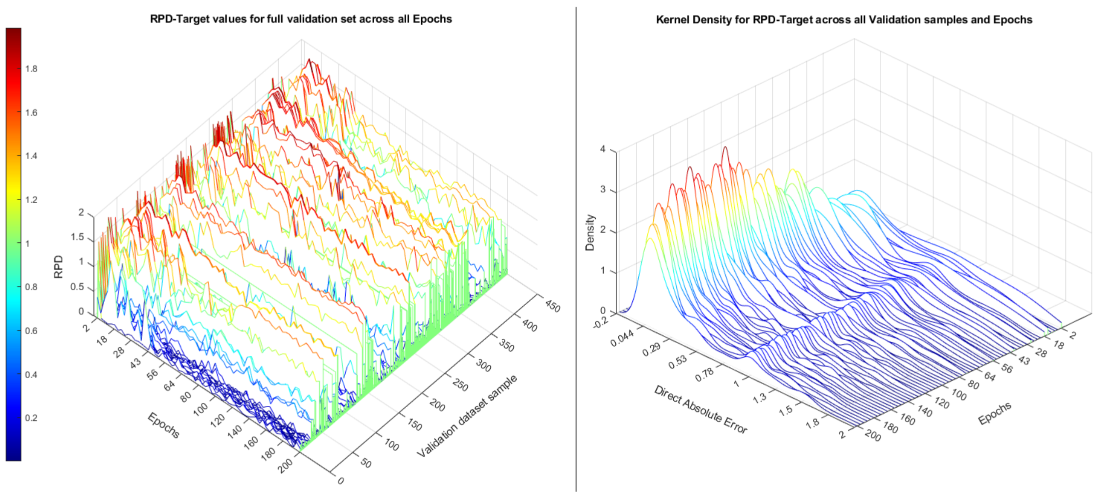


Figure 4.16: Experiment 2: RPD(Target) errors across Epochs. Left:Waterfall of errors for each sample across epochs. Right: Kernel Density plot of all samples per epoch.

Upon request the application is able to produce a result in mere fractions of a second. Furthermore additional loads can be added to the imput an a new updated prediction obtained in quick succession:

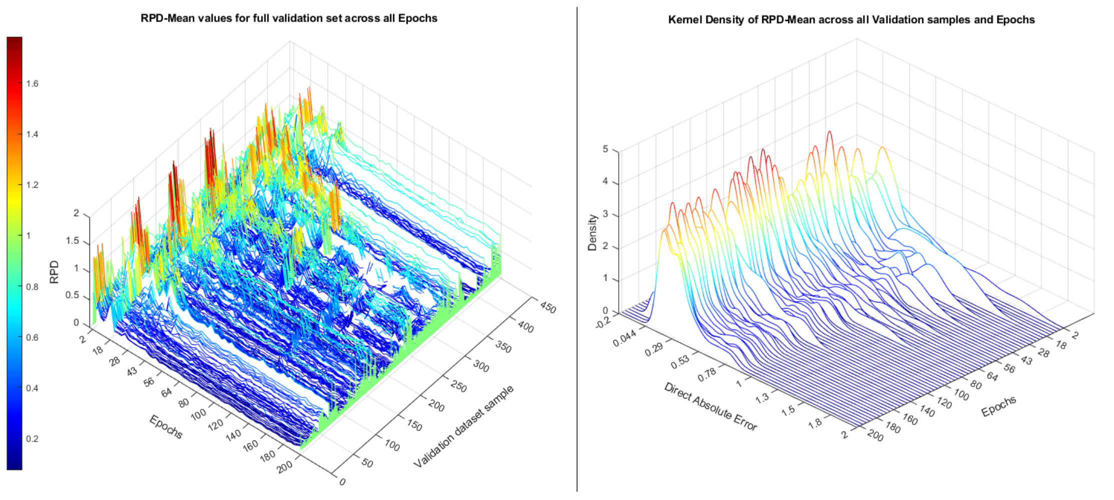


Figure 4.17: Experiment 2: RPD(Mean) errors across Epochs. Left:Waterfall of errors for each sample across epochs. Right: Kernel Density plot of all samples per epoch.

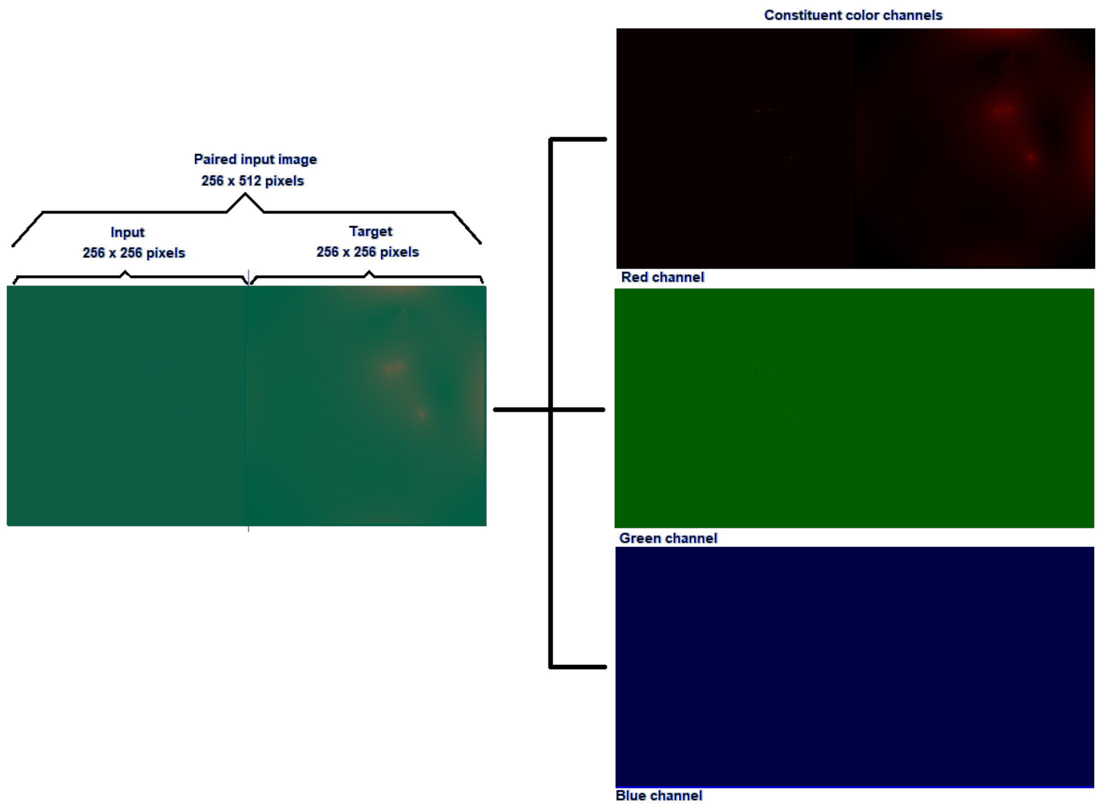


Figure 4.18: Color channel decomposition for Experiment 3. Single sample of 256 × 512 .png paired image dataset reflecting the color channel mapping: Red channel for input force(s) and response stress magnitudes. Green and blue for thickness and modulus of elasticity respectively.

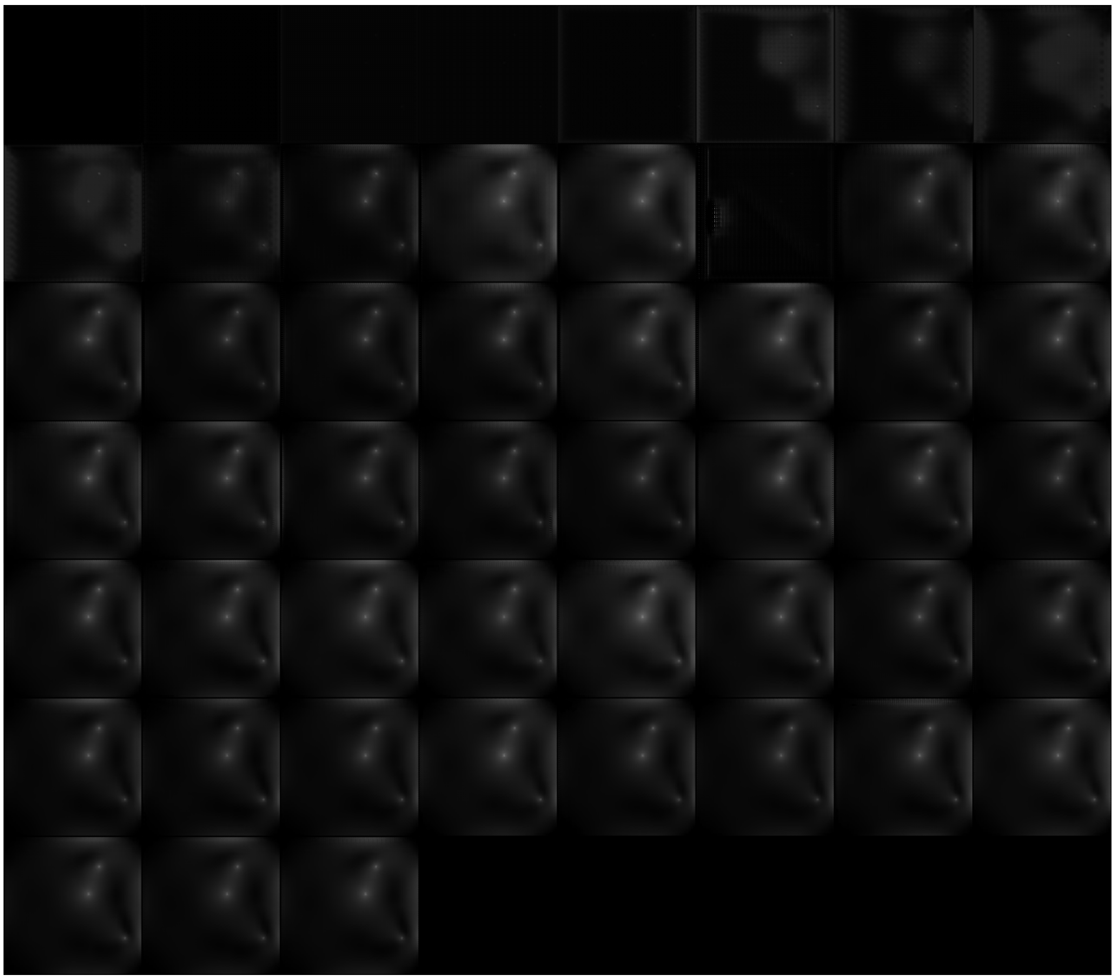


Figure 4.19: Generated predictions for sequential saved models or Experiment 3. 50 saved models which run from left to right and top to bottom. Note how the finer details appear to emerge with continued training.

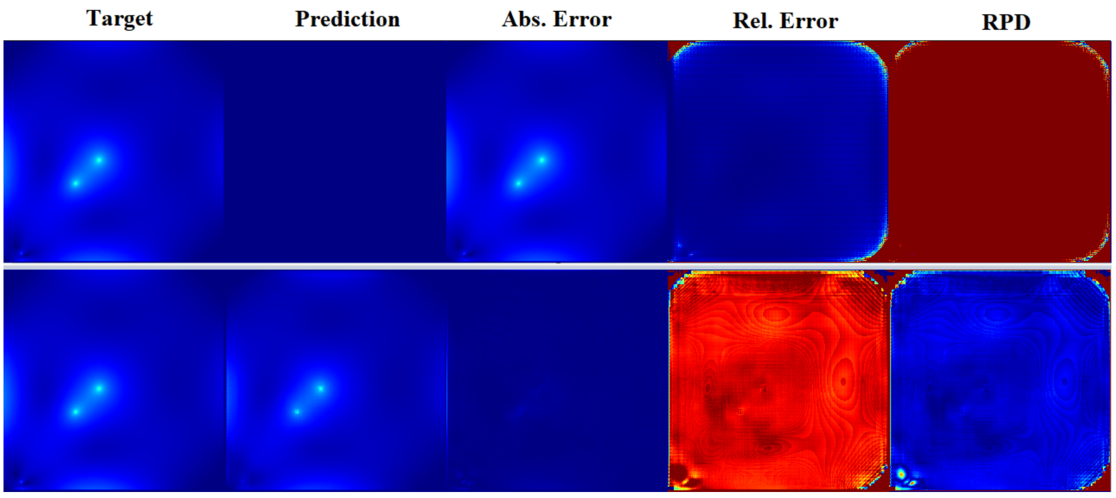


Figure 4.20: Experiment 3 validation results for fourth (top) vs final model Epoch (bottom). Outputs of the validation error metrics across a sample generated prediction. Note the transition of Absolute Error from essentially representing the Target to almost no error.

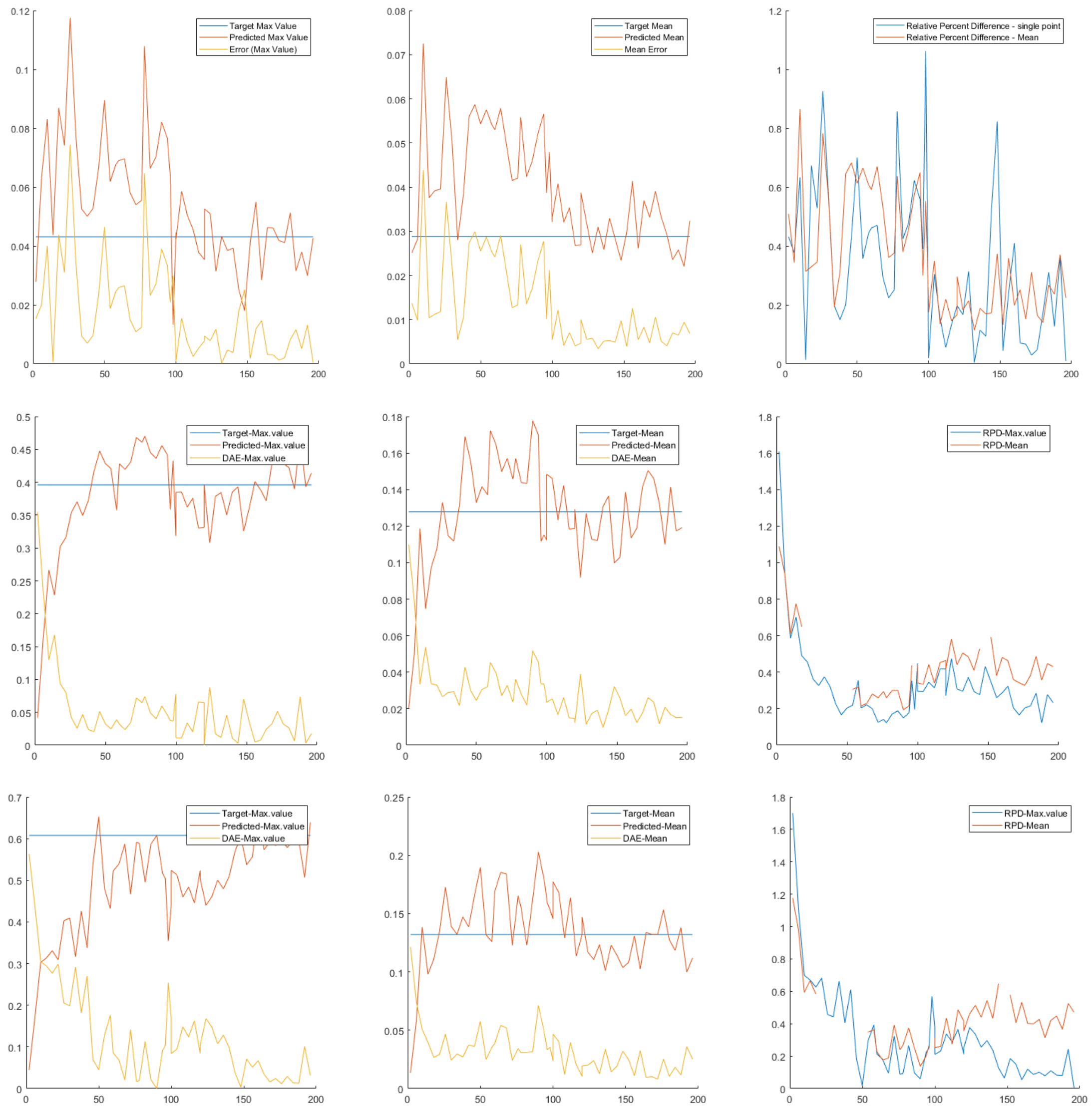


Figure 4.21: Experiment 3: Force to stress model validation results of random samples across 200 Epochs

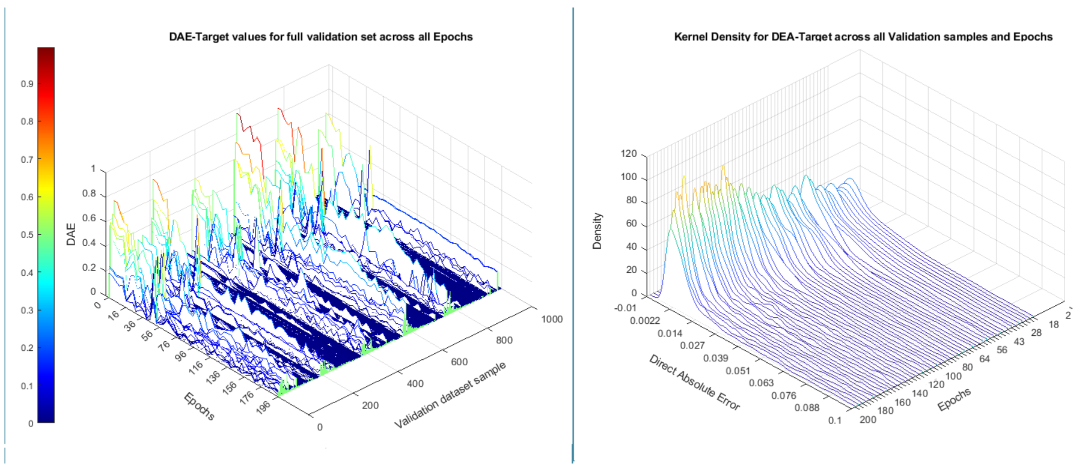


Figure 4.22: Experiment 3: DAE (Target) results plot for all validation samples across all model Epochs. Left: Waterfall of progressive DAE target values. Right: Kernel Density of error distribution.

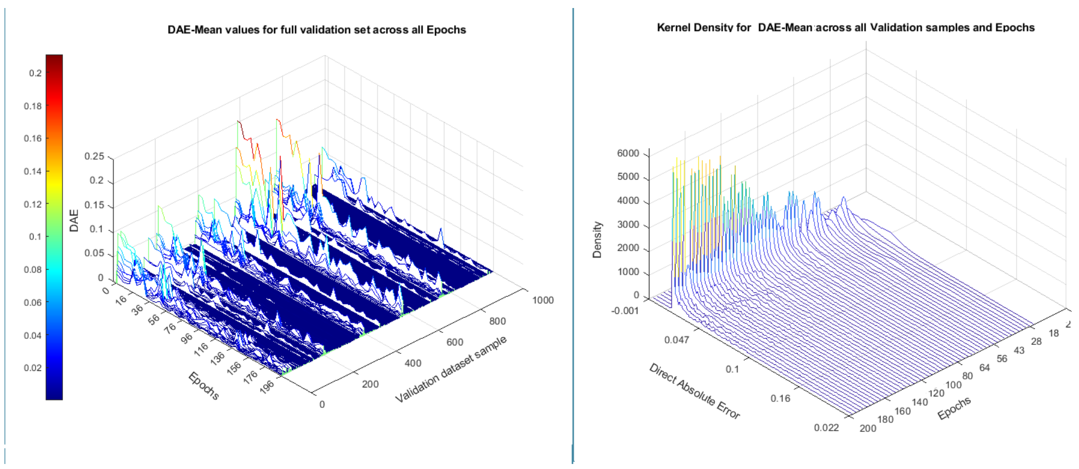


Figure 4.23: Experiment 3: DAE (Mean values) results plot for all validation samples across all model Epochs. Left: Waterfall of progressive DAE mean values. Right: Kernel Density of error distribution.

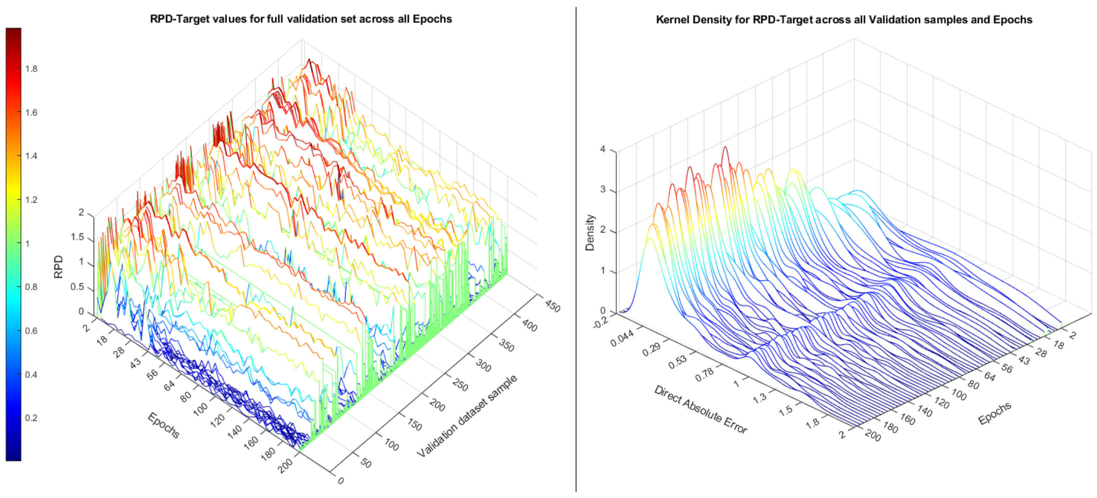


Figure 4.24: Experiment 3: RPD (Target) results plot for all validation samples across all model Epochs. Left: Waterfall of progressive RPD target values. Right: Kernel Density of error distribution.

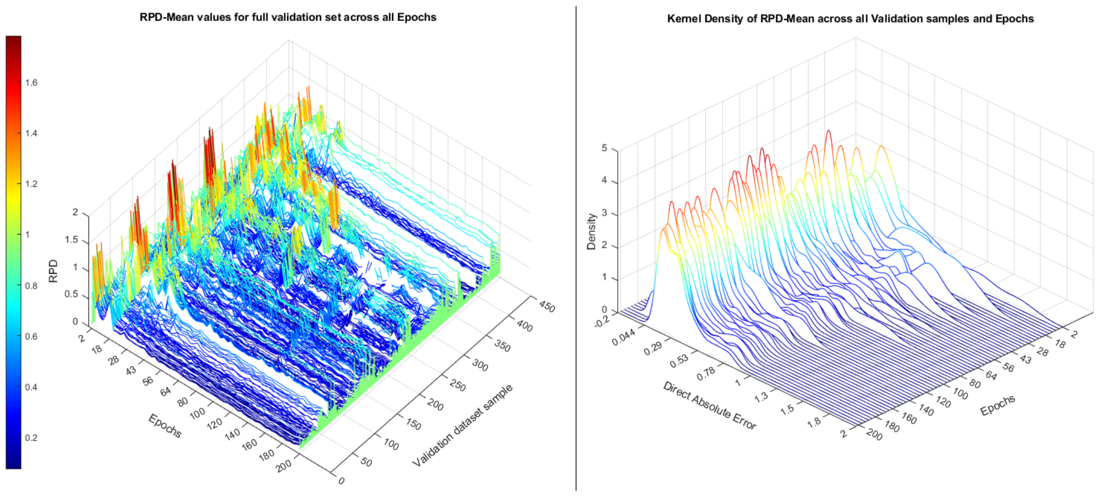


Figure 4.25: Experiment 3: RPD (Mean values) results plot for all validation samples across all model Epochs. Left: Waterfall of progressive RPD mean values. Right: Kernel Density of error distribution.

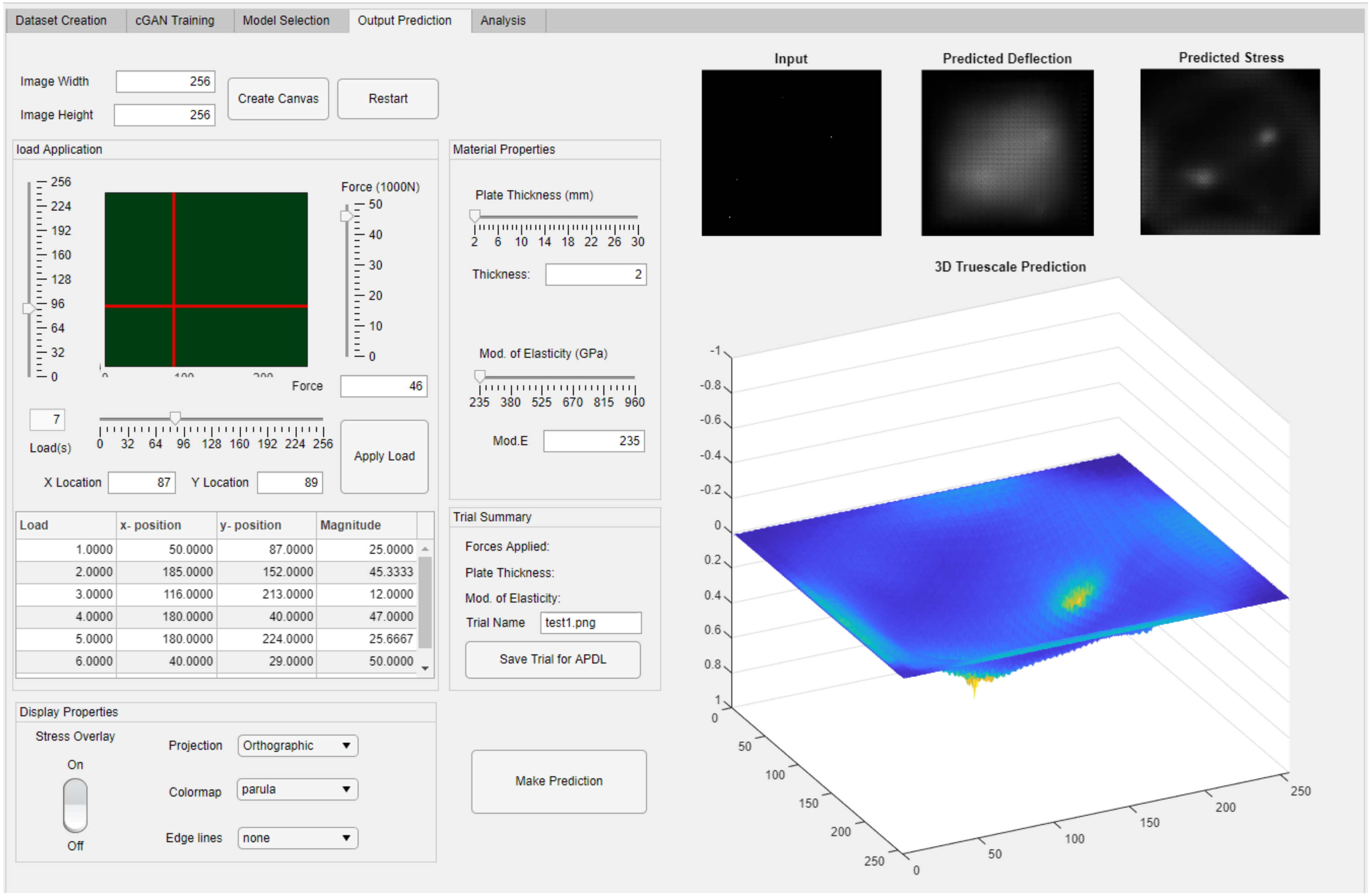


Image Width 256

Image Height 256

Force 46

Load(s) 7

X Location 87 Y Location 89

Thickness: 2

Mod.E 235

Trial Name test1.png

Load	x- position	y- position	Magnitude
1.0000	50.0000	87.0000	25.0000
2.0000	185.0000	152.0000	45.3333
3.0000	116.0000	213.0000	12.0000
4.0000	180.0000	40.0000	47.0000
5.0000	180.0000	224.0000	25.6667
6.0000	40.0000	29.0000	50.0000

Projection Orthographic

Colormap parula

Edge lines none

Figure 4.26: Example prediction using purpose built Graphical User Interface

5 Conclusion

"All models are wrong but some are useful" - George Box

5.1 Summary

Recall that the primary aim of this thesis is the development of a purpose-built ML model capable of learning the inherent physical domain of commonplace FEA problems in order to produce user requested predictions in real-time. To this end a literature study was conducted on the theoretical development, working methodologies and optimization techniques employed in ML, with focus towards CNNs and ultimately GANs. The general structure of the "Pix2Pix" cGAN network architecture was selected for its potential, but as yet untested predictive abilities with respect to FEA problems. A large FEM dataset was created using conventional FEA practices for isotropic thin-plates subjected to varying static loads for a range of thicknesses and modulus of elasticity. The FEA dataset was restructured into paired-image datasets to meet the usage requirements of the cGAN. Three separate models were trained in order to map forces to deflections, deflections to stresses and forces to stresses. These models were validated and found to be consistently in the range of 90-95% accurate. Following validation, a purpose-built GUI application was developed. The application allows for finer granularity of input than the dataset for which the models were trained, yet is capable of repeatedly producing results (deflection and stress predictions) in a fraction of the time of the conventional FEA method.

5.2 conclusions and future work

Given the positive results of the thesis, one might be drawn to conclude that they point favourably towards the usage of ML, more specifically cGANs, as potential alternatives to conventional FEA methodologies especially when seeking a means to produce results rapidly (read real-time). The inescapable requirement of training datasets and training stages, both of which require time to obtain and execute, likely negates the time benefits for all but the most specific usage cases, however that is not to dismiss that they do exist. Hypothetically similar techniques and models could be trained once and used repeatedly for VR training in the medical field, or on structural components such as aircraft fuselages and control surfaces or the stress monitoring of wind turbine blades. These trained models could speculatively be deployed across entire fleets of the same aircraft model or wind turbine for the purposes of active condition monitoring and feedback. Yet another caveat to consider is the limitation of accuracy confidence to within the bounds of the initial training datasets' domains - our results show reasonable agreement to conventional FEA results within these bounds but without a means to input data beyond them (at least as developed here) we are unable to assess accuracy beyond them either. In conclusion, the cGAN model and methodology as presented in this thesis does achieve the desired objectives as initially set out but is limited to the domain of the problem dataset it was presented with. This initial success as well as the relatively robust and adaptive nature of the cGAN architecture does however display promise and merits further study and development. Proposed future works could thus look towards:

- analysis of non-linear response as forces are initially applied at large distances, followed by incremental convergence towards one another.

- the inclusion of additional input channels to convey various physical conditions such as varying geometries, boundary conditions or discontinuities.
- varying the otherwise uniform thickness and modulus of elasticity channels to simulate anisotropic and/or composite materials.
- using additional channel layers to create three-dimensional volumes for three-dimensional geometries.
- utilizing a dataset of non-linear inputs and responses.
- optimization of the network architecture by elimination of neurons of low/no activation
- obtaining the reaction forces for usage as inputs into subsequent connected models- perhaps towards obtaining a "models-as-elements" approach.

REFERENCES

[1] M. Turner, R. Clough, H. Martin, and L. Topp. Stiffness and Deflection Analysis of Complex Structures. *JOURNAL OF THE AERONAUTICAL SCIENCES*, 23(9), September 1956.

[2] H. Tyson Jr. Caltech's Role in NASTRAN. *California Institute of Technology, ME Centennial Publication*, February 2007.

[3] Coen E. Kennaway R. Volumetric finite-element modelling of biological growth. *Open Biology*, 9(190057), 2019.

[4] M. Driscoll. The Impact of the Finte Element Method on Medical Device Design. *Journal of Medical and Biological Engineering*, 39:171–172, 2019.

[5] J. Li, Y. Zhou, L. Zhang, Y. Tian, M. Cassidy, and L. Zhang. Random finite element method for spudcan foundations in spatially variable soils. *Engineering Geology*, 205:146–155, 2016.

[6] S.J.Hawkins, A.L.Allcock, A.E.Bates, L.B.Firth, I.P.Smith, S.E.Swearer, and P.A.Todd. *Oceanography and Marine Biology An Annual Review*, volume 57. CRC Press, 2019.

[7] M. Dinniman, X. Asay-Davis, K. Galton-Fenzi, P. Holland, A. Jenkins, R. Timmermann, and A. Wegener. Modeling Ice Shelf Ocean Interaction in Antarctica: A Review. *Oceanography*, December 2016.

[8] Xiaolin Chen and Yijun Liu. *Finite Element Modeling and Simulation with ANSYS Workbench*. CRC Press, 2014.

[9] O.A.Bauchau and J.I.Craig. *Structural Analysis With Applications to Aerospace Structures*. Springer, 2009.

[10] T. Zohdi. *A Finite Element Primer for Beginners*. 01 2018.

[11] R.C.Hibbeler. *Statics and Mechanics of Materials*. Pearson Education, Inc., 2019.

[12] F.P.Beer, E.Russel Johnson Jr, J.T.DeWolf, and D.F.Mazuerk. *Mechanics of Materials*. McGrawHill, eighth edition, 2019.

[13] Alan B. Palazzolo. *Vibration Theory and Applicationswith Finite Elements and ActiveVibration Control*. Wiley, 2016.

[14] Erdogan Madenci and Ibrahim Guven. *The Finite Element Method and Applications in Engineering Using ANSYS®*. Springer, second edition, 2015.

[15] Mary Kathryn Thompson and John M. Thompson. *ANSYS Mechanical APDL for Finite Element Analysis*. Elsevier, 2017.

[16] Dragan Marinkovic and Manfred Zehn. Survey of Finite Element Method-Based Real-Time Simulations. *Applied Sciences*, 9:2775, July 2019.

[17] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[18] Serban Georgescu, Peter Chow, and Hiroshi Okuda. GPU acceleration for FEM-based structural analysis. *Archives of Computational Methods in Engineering*, 20, May 2013.

[19] Laurian Gherman, Ovidiu Moșoiu, and Vasile Bucinschi. The 12th International Scientific Conference eLearning and Software for Education INFORMATION

RESOURCES MANAGEMENT BASED ON FEEDBACK THEORY IN A MILITARY ORGANIZATION. *eLearning and Software for Education Conference eLSE 2016*, 1:115–122, 06 2016.

[20] I.Nikitin, L.Nikitina, P.Frolov, G.Goebbels, M.Göbel, and S.Klimenko. Real-time simulation of elastic objects in virtual environments using finite elementmethod and precomputed Green's functions. In *Proceedings of the workshop on virtual environments*, pages 30–1, Barcelona (Spain);, 2002.

[21] J.M.Huang, S.K.Ong, and A.Y.C.Nee. Real-time finite element structural analysis in augmented reality. *Advances in Engineering Software*, 87:43–56, 2015.

[22] Proscilla Cerracchio, Marco Gherlone, and Alexander Tessler. Real-time displacement monitoring of a composite stiffenedpanel subjected to mechanical and thermal loads. *Meccanica*, 50:2487–2496, 2015.

[23] Qiu Guan, Xiaochen Du, Yan Shao, Lili Lin, and Shengyong Chen. Three-dimensional simulation of scalp soft tissue expansion using finite element method. *Computational and mathematical methods in medicine*, 2014:360981, 2014.

[24] Xiaoya Li, Chenlin Li, Zhangna Xue, and Xiaogeng Tian. Investigation of transient thermo-mechanical responses on the triple-layered skin tissue with temperature dependent blood perfusion rate. *International Journal of Thermal Sciences*, 139:339 – 349, 2019.

[25] Thomas Heidlauf, Thomas Klotz, Christian Rode, Tobias Siebert, and Oliver Röhrle. Force enhancement and stability of finite element muscle models. *PAMM*, 16:85–86, 10 2016.

[26] Thomas Heidlauf, Thomas Klotz, Christian Rode, Tobias Siebert, and Oliver Röhrle. A continuum-mechanical skeletal muscle model including actin-titin interaction predicts stable contractions on the descending limb of the force-length relation. *PLOS Computational Biology*, 13:e1005773, 10 2017.

[27] Satish Panda and Martin Buist. A finite element approach for gastrointestinal tissue mechanics. *International Journal for Numerical Methods in Biomedical Engineering*, 35, 10 2019.

[28] Ibrahim El Bojairami and Mark Driscoll. A 3-DIMENSIONAL FINITE ELEMENT MUSCLE MODEL TO PREDICT FORCES IN THE CLINIC, August 2018.

[29] M. Ferrant, A. Nabavi, B. Macq, F. A. Jolesz, R. Kikinis, and S. K. Warfield. Registration of 3-d intraoperative MR images of the brain using a finite-element biomechanical model. *IEEE Transactions on Medical Imaging*, 20(12):1384–1397, 2001.

[30] Guillaume Picinbono, Jean-Christophe Lombardo, Herve Delingette, and Nicholas Ayache. Improving realism of a surgery simulator: linear anisotropic elasticity, complex interactions and force extrapolation. *The Journal of Visualization and Computer Animation*, 13(3):147–167, 2002.

[31] Wen Wu, Jian Sun, and Pheng-Ann Heng. A hybrid condensed finite element model for interactive 3D soft tissue cutting. *Studies in health technology and informatics*, 94:401–3, February 2003.

[32] Erol Onal and Veysi Isler. Cloth Tearing Simulation. 2014.

[33] Yu Wang, Shuxiang Guo, and Baofeng Gao. Vascular elasticity determined mass-spring model for virtual reality simulators. *International Journal of Mechatronics and Automation*, 5, January 2015.

[34] Erhan Ferhatoglu, Ender Cigeroglu, and H. Nevzat Özgüven. A novel modal superposition method with response dependent nonlinear modes for periodic vibration analysis of large MDOF nonlinear systems. *Mechanical Systems and Signal Processing*, 135:106388, 2020.

[35] Karl Pearson. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[36] Alaa Tharwat. Principal component analysis - a tutorial. *International Journal of Applied Pattern Recognition*, 3:197, 01 2016.

[37] Ridha Hambli, Abdssalam Chamekh, and Hedi Bel Hadj Salah. Real-time deformation of structure using finite element and neural networksin virtual reality applications. *Finite Elements in Analysis and Design*, 42:985–991, 2006.

[38] Keny Ordaz-Hernández, Xavier Fischer, and Fouad Bennis. Model reduction technique for mechanical behaviour modelling: Efficiency criteria and validity domain assessment. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 222:493 – 505, 2008.

[39] Ken'ichi Morooka, Xian Chen, Ryo Kurazume, Seiichi Uchida, Kenji Hara, Yumi Iwashita, and Makoto Hashizume. Real-time nonlinear FEM with neural network for simulating soft organ model deformation. *Medical image computing and*

computer-assisted intervention : MICCAI ... International Conference on Medical Image Computing and Computer-Assisted Intervention, 11(Pt 2):742—749, 2008.

[40] Ameet V. Joshi. *Machine Learning and Artificial Intelligence*. Springer, 2019.

[41] Niklas Kühl, Marc Goutier, Robin Hirt, and Gerhard Satzger. Machine Learning in Artificial Intelligence: Towards a Common Understanding, 2020.

[42] Andriy Burkov. *The Hundred Page Machine Learning Book*. Amazon, 2019.

[43] F. Siddique, S. Sakib, and M. A. B. Siddique. Recognition of Handwritten Digit using Convolutional Neural Network in Python with Tensorflow and Comparison of Performance for Various Hidden Layers. In *2019 5th International Conference on Advances in Electrical Engineering (ICAEE)*, pages 541–546, 2019.

[44] Vivek Kumar, Denis Kalitin, and Prayag Tiwari. Unsupervised Learning Dimensionality Reduction Algorithm PCA For Face Recognition. In *International Conference on Computing, Communication and Automation*. International Conference on Computing, Communication and Automation (ICCCA2017), 05 2017.

[45] Alexander Mey and Marco Loog. Improvability Through Semi-Supervised Learning: A Survey of Theoretical Results, 2019.

[46] Kai Arulkumaran, Antoine Cully, and Julian Togelius. AlphaStar: an evolutionary computation perspective. pages 314–315, July 2019.

[47] Yue Deng, Feng Bao, Youyong Kong, Zhiquan Ren, and Qionghai Dai. Deep Direct Reinforcement Learning for Financial Signal Representation and Trading. *IEEE Transactions on Neural Networks and Learning Systems*, 28:1–12, February 2016.

[48] Terry Meng and Matloob Khushi. Reinforcement Learning in Financial Markets. *Data*, 4:110, July 2019.

[49] Xue Bin Peng, Michael Chang, Grace Zhang, Pieter Abbeel, and Sergey Levine. MCP: Learning Composable Hierarchical Control with Multiplicative Compositional Policies, 2019.

[50] COGNUB. COGNITIVE COMPUTING AND MACHINE LEARNING. Online, 2019.

[51] Kailash Ahirwar. Everything you need to know about neural networks. [Online], November 2017.

[52] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning:From Theory to Algorithms*. Cambridge University Press, 2014.

[53] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2016.

[54] Léon Bottou. *Neural Networks: Tricks of the Trade*, chapter Stochastic Gradient Descent Tricks, pages 421–436. Springer, second edition, 2012.

[55] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[56] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014.

[57] Stan Li and Anil Jain. pages 883–883. Springer US, Boston, MA, 2009.

[58] N. Kwak. Principal Component Analysis by L_p -Norm Maximization. *IEEE Transactions on Cybernetics*, 44(5):594–609, 2014.

[59] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propogating errors. *Nature*, 323:533–536, October 1986.

[60] Hermann König and Vitali Milman. *The Chain Rule*, pages 53–73. Springer International Publishing, Cham, 2018.

[61] Michael A. Nielson. *Neural Networks and Deep Learning*. Determination Press, 2015.

[62] Z. Li, M. Dong, S. Wen, X. Hu, P. Zhou, and Z. Zeng. CLU-CNNs: Object detection for medical images. *Neurocomputing*, April 2019.

[63] Messaoud Hameurlaine, Abdelouahab Moussaoui, and Benaissa Safa. Deep Learning for Medical Image Analysis. December 2019.

[64] Mariusz Bojarski, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Larry Jackel, Urs Muller, and Karol Zieba. VisualBackProp: visualizing CNNs for autonomous driving. November 2016.

[65] Navin Ranjan, Sovit Bhandari, Hong Zhao, Hoon Kim, and Pervez Khan. City-Wide Traffic Congestion Prediction based on CNN, LSTM and Transpose CNN. *IEEE Access*, PP:1–1, April 2020.

[66] Keiron O'Shea and Ryan Nash. An Introduction to Convolutional Neural Networks. *ArXiv e-prints*, November 2015.

[67] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Y. Bengio. Generative Adversarial Networks. *Advances in Neural Information Processing Systems*, 3, June 2014.

[68] Pradeep Pujari, Md. Rezaul Karim, and Mohit Sewak. *Practical Convolutional Neural Networks*. Packt Publishing, 2018.

[69] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[70] T. Park, M. Liu, T. Wang, and J. Zhu. Semantic Image Synthesis With Spatially-Adaptive Normalization. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2332–2341, 2019.

[71] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila. Analyzing and Improving the Image Quality of StyleGAN. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8107–8116, 2020.

[72] Stephen Timoshenko and S. Woinowsky-Krieger. *Theory of Plates and Shells*. McGraw-Hill Book Company, Inc, 1959.

[73] Ansel C. Ugural. *PLATES AND SHELLS Theory and Analysis*. CRC Press, fourth edition edition, 2018.

[74] *Ansys® Mechanical APDL, 2019 R2, help system, Element Reference, ANSYS, Inc*, chapter SHELL181. 2019.

[75] Sustainability of Digital Formats: Planning for Library of Congress Collections, December 2011.

[76] Richard Wiggins, H. Davidson, H. Harnsberger, Jason Lauman, and Patricia Goede. Image File Formats: Past, Present, and Future. *Radiographics : a re-*

view publication of the Radiological Society of North America, Inc, 21:789–98, 11 2000.

Appendix A: APDL Code

A.1 Dataset Generation script

File added to the portfolio electronically under the name “LPAV7.m”.

Appendix B: MATLAB Functions

B.1 PairedImageDataset

File added to the portfolio electronically under the name “PairedImageDataset.lxm”.

B.2 FileCount

File added to the portfolio electronically under the name “FileCount.m”.

B.3 ReadRaw

File added to the portfolio electronically under the name “ReadRaw.m”.

B.4 GenDataset

File added to the portfolio electronically under the name “GenDataset.m”.

B.5 ImgGen

File added to the portfolio electronically under the name “ImgGen.m”.