# Predicting elastic-plastic response of random periodic composite materials: an ANN-CNN comparative study

---

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

---

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

(Engineering Systems and Environment)

By

Xiushang Wu

October 2022

# Approval Sheet

This dissertation is submitted in partial fulfillment of the
requirements for the degree of
Master of Science (Civil Engineering/Applied Mechanics)
Xiushang Wu

Author

This dissertation has been read and approved by the Examining Committee:

Marek-Jerzy Pindera (Advisor)

Jose Pantaleon Gomez III (Chairman)

James H. Lambert

献给我亲爱的父母和弟弟：

武志金；牛明英；武修林

*To My Dear Parents and Young Brother:*

*Wu, Zhijin; Niu Mingying; Wu Xiulin*

# Acknowledgements

I would like to sincerely thank Professor Marek-Jerzy Pindera for his insightful guidance and unlimited support during my entire MS. study. His kindness, patient and passion are deeply appreciated. The comments and suggestions from committee members: Professor Jose Pantaleon Gomez III and Professor James H. Lambert during my MS defense are gratefully appreciated.

I also want to thank Mollie, Professor Pindera's wife. Her kindness and consideration are greatly appreciated.

The author is very thankful for the funding support from the Department of Civil and Environmental Engineering through Graduate Teaching Assistantships and summer fellowships.

Special thanks go to the author's parents, Xiushang Wu, Mingying Niu and the Young brother, Xiulin Wu. Without their constant support and encouragement, the author would not be here.

Thank you!

# Abstract

In this thesis, a computational scheme was developed to generate thousands of microstructure realizations of unidirectional composites with random fiber distributions employed by the homogenization theory called FVDAM. Once the microstructure was realized using randomly distributed fiber centers, it was then discretized into equally dimensioned subvolumes, and the material assignment matrix was created for input into FVDAM simulation. Subsequently, the FVDAM homogenization theory was incorporated into a python-driven interface that enabled generation of thousands of elastic-plastic stress-strain curves for unidirectional metal matrix composites with random fiber distributions. The generated microstructure realizations, the corresponding homogenized elastic moduli and stress-strain responses were then employed in ANN and CNN architectures that were designed and optimized for predictive purposes.

The calculated homogenized moduli and stress-strain responses under six fundamental loading modes were first correlated with the microstructural realizations to understand the effect of random fiber distributions on the response in the elastic and elastic-plastic regions. Whereas the effect of fiber randomness on the homogenized moduli is small, it is much larger on the elastic-plastic response, but also dependent on the loading direction relative to the fiber orientation. As expected, and confirmed by simulations herein, the microstructural randomness has virtually no effect on the response by uniaxial loading along the fiber direction due to the constraint of the fibers that controls the plastic strain evolution. Large microstructure effects are seen under normal loading transverse to the fiber direction, which become somewhat smaller under transverse shear and smaller still under axial shear.

Subsequently, deep ANN and CNN architectures were designed and optimized to predict both the homogenized elastic moduli and direction-dependent elastic-plastic stress-strain responses of microstructural realizations representative of random fiber composites. Whereas the input to the ANN model consisted of fiber placement locations, the CNN model employed full-field microstructural images discretized into subvolumes or pixels. Nineteen hundred and fifty microstructural realizations were sufficient for training, testing and validation, which produced very good prediction of the homogenized stress-strain responses of the remaining fifty realizations by

both ANN and CNN algorithms. In contrast, 20,000 realizations were required for the prediction of the 13 homogenized elastic moduli based on the CNN algorithm due to the small effect of random fiber microstructures. Nonetheless, the CNN algorithm successfully captured the very small moduli indicative of monoclinic behavior several orders of magnitude smaller than the moduli characteristic of orthotropic response. By contrast, the ANN algorithm did not perform well due to the input data type and the size of the training data set, likely because the small differences in the homogenized moduli produced by fiber placement variations required significantly larger number of features and/or data for accurate prediction.

The algorithms and generated results reported in the thesis are important in developing accurate ML-based computational models for implementation in multi-scale analyses of large-scale composite structures. Perhaps most significantly, the execution times required to predict the homogenized elastic-plastic response of random fiber composites based on the ANN/CNN algorithms are several orders of magnitude smaller that the full-scale calculations based on the FVDAM homogenization theory, enabling multi-scale analysis of composite structures.

# List of Figures

# List of Tables

# Table of Contents

# Chapter 1

# Introduction

## 1.1 Motivation

Fiber-reinforced materials are the building blocks of laminated and woven composites used in modern structural applications. Most traditional composites are fabricated with small-diameter fibers such as glass, carbon, or graphite, producing microstructures that are intrinsically random, requiring large number of microstructural realizations for establishing structure-property relationships. These relationships are expressed in terms of linear or nonlinear equations, often called generalized Hooke's law, that relate average stresses to average strains applied to a small enough volume of a composite material considered as representative of the overall behavior. The term homogenization is employed to describe computational techniques that accomplish this. Linear relations describe linearly elastic behavior whereas nonlinear equations describe behavior that includes plasticity, creep, etc.

Homogenization-based analyses of representative volume elements (RVE) or repeating unit cells (RUC) characteristic of statistically homogeneous or periodic composites, respectively, Fig. 1.1, present computational challenges when large numbers of randomly distributed fibers contained within an RVE or RUC are involved. The difference in the two representations lies in how the boundary conditions are applied to the bounding surface of the two volumes, Drago and Pindera [1]. In the case of statistically homogeneous composites, either homogeneous displacement or homogeneous tractions are applied which are expected to produce the same homogenized moduli. Either of the two boundary conditions are supposed to produce the same boundary deformation. As an example, a square RVE will deform into a rectangular one under stresses transverse to the fibers and into a smaller square under axial stress parallel to the fibers. This can only be approximately achieved if the RVE contains enough fibers because the boundary deformation will be influenced by the fibers adjacent to it. In contrast, in the case of periodic composites, periodic boundary conditions involving both displacements and tractions are applied.

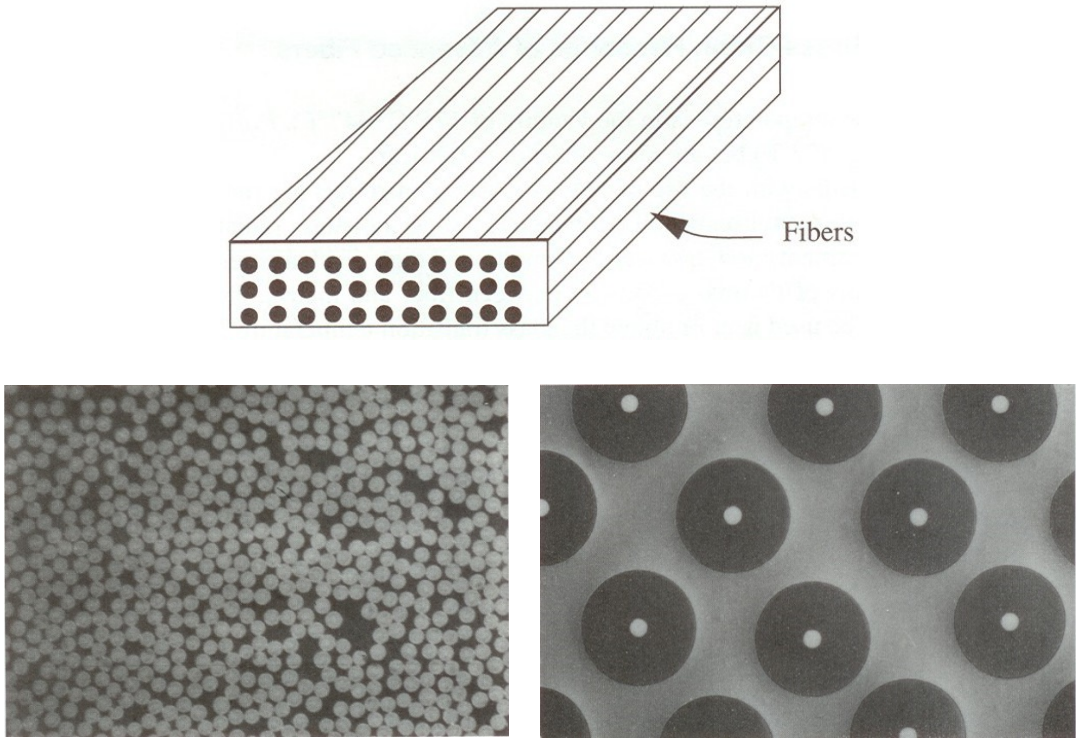



*Figure 1.1. Schematic representation of a unidirectional composite (top), and statistically homogeneous (bottom left) and periodic (bottom right) material microstructures.*

The two representations produce similar homogenized moduli if the volume representative of the microstructure contains enough fibers within it.

Majority of homogenization techniques applied to random fiber composites that satisfy the requirement of local stress field accuracy necessary for accurate elastic and post-elastic homogenization are based on numerical approaches. These include finite-difference, finite-element and finite-volume based solutions of the stress fields within RVE or RUC, with the finite element method being the dominant one, as discussed in the reviews provided in Refs. [2-4]. Some progress has been made in developing alternative computational approaches, such as the finite volume and elasticity-based methods [5,6], but the finite element method continues to dominate despite its limitations vis-à-vis rapid analysis of multi-fiber RUCs of periodic material microstructures. The need to generate thousands of random microstructure realizations for characterizing the response of random fiber unidirectional composites makes the method ill-suited for microstructure-property identification.

An alternative approach to characterize the elastic and elastic-plastic response of random fiber composites is to use machine learning (ML) techniques that establish relationships between microstructural features and the homogenized response under different types of loading. The application of artificial intelligence (AI) techniques in general, and machine learning (ML) techniques specifically, to the area of mechanics of materials is nowadays attracting increasing attention because of demonstrated dramatic reductions in computational times required to characterize the material response at the homogenized level. A properly trained ML algorithm is capable of predicting the homogenized response of a material microstructure that it has not seen before much faster than solving the entire RVE or RUC problem numerically. The development of an accurate ML algorithm requires generation of a large data set that enables to establish an implicit relationship between the microstructural features of a random fiber composite and the resulting homogenized stress-strain curve under different loading directions. Once the data set is generated, an ML algorithm is then trained and then employed for predictive purposes.

## 1.2 Machine learning approaches and background work

ML techniques mimic the way a human brain processes information received from different sensory receptors. The theoretical basis for construction of logical networks that mimic brain activity has been developed nearly 80 years ago [7]. However, it has only been relatively recently that these artificial networks began to play an important role in fluid and solid mechanics communities, and elsewhere, due to the introduction of a back-propagation algorithm [8] that efficiently identifies optimal values of weights associated with nodal connections between adjacent layers of a neural network. These networks now play increasingly important roles in, amongst many applications, image identification and segmentation, microstructural identification, discovery of novel materials for targeted applications. In solid mechanics, the back-propagation neural network algorithm has been employed in a seminal paper to model the nonlinear response of concrete [9], motivating rapid popularization and spread of this data-driven modeling approach. ML algorithms continue to be employed in biomedical applications, discovery of new materials with unique properties, re-construction of fluid flow from limited sensor input data as well as construction of constitutive equations in the elastic and elastic-plastic regimes, amongst many other applications [10-19]. Most recently, there has been an explosion of different neural network architectures

proposed, with emphasis on incorporating physics-based models that target key deformation mechanisms through loss functions, for example [20-22]. These endeavors are limited when applied to heterogeneous materials, however, because of the prohibitively high computational costs in generating large sets of data using the finite-element approach, which continues to be the dominant analysis technique.

Neural network architectures employed to predict homogenized response and recover local fields of heterogeneous materials include deep ANNs (more than three inner layers), CNNs, and recurrent neural networks (RNN) including GRUs, LSTMs and TCNs, with new architectures proposed continuously. Typically, these networks map loading history, material properties, and/or microstructure onto homogenized response or local fields under typically unidirectional loading, but generally not both thus far. Literature search reveals that there are no reported results of network architectures where loading, material properties and microstructure are mapped onto homogenized stress and local stress fields whilst incorporating physics-based considerations that guide the weight optimization and ensure stability of the predicted homogenized stresses in structural analysis applications.

Previous work has shown that both the ANN and CNN algorithms have performed well in predicting the effective homogenized moduli based either on the coordinates of fiber centers in the case of ANN, and on the RUC images obtained from fibers centers and radii in the case of CNN, when the data size is big enough, [23]. These results were obtained from numerical experiments based on 60,000 microstructural realizations of the RUC. In both the ANN and CNN models, 48,000 microstructural realizations were used for training, and 12,000 for testing. The accuracy for the ANN model was about 97% and the CNN around 98%. The microstructural realizations were generated by assigning random values to the fiber centers, which defined the fiber radii for the investigated fiber volume fractions of the RUC, and the homogenized moduli for different numbers of randomly placed fibers within the RUC were calculated by the hybrid homogenization theory (HHT) developed by Yin at al. [24]. Homogenized moduli of 60,000 RUC microstructural realizations could be calculated using HHT due to its efficiency arising from the combined analytical and semi-analytical approaches in determining the fiber and matrix stress fields, and generating the random fiber microstructures. Figure 1.2 illustrates the degree of accuracy of predicting the 13 independent elastic homogenized moduli by the ANN algorithm with similar

results obtained from the CNN algorithm. We note that HHT at present is limited to elastic analysis and therefore, despite its efficiency, elastic-plastic response of thousands of random RUCs could not be generated.

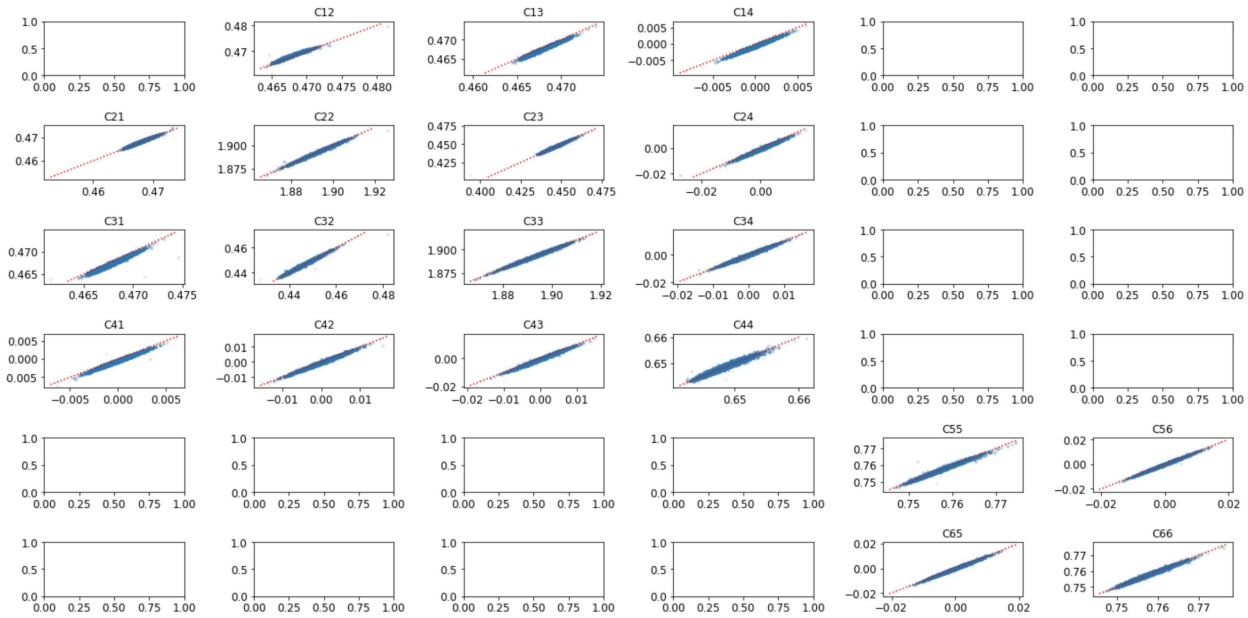



*Figure 1.2 Previous work based on the 60,000 RUCs, illustrating the predictive capability of the ANN algorithm in estimating homogenized moduli of a random unidirectional composite, Ref.* [23].

## 1.3 Rapid homogenization of random composites via FVDAM

Neural networks require large data sets for training purposes. In the context of developing neural network architectures that map a complex material microstructure onto a homogenized stress-strain curve, generation of such data involves repeated solution of a RUC boundary-value problem for a given loading path for each microstructural realization with multiple, randomly situated fibers. Accurate solutions to such problems may only be achieved using numerical techniques, with the finite-element method the most common one.

An attractive alternative to the solution of homogenization problems is offered by the finite-volume method which continues to gain popularity. The finite-volume method was originally developed for the solution of boundary-value problems in fluid mechanics, cf. Versteeg and

Malalasekera [25]. Satisfaction of the governing (transport or equilibrium) field equations within subvolumes of the investigated discretized domain in an integral sense is a key feature of the finite-volume method which distinguishes it from variational techniques such as the finite-element method. In the context of fluid mechanics applications, this is done upon first expressing the field equations in a finite-difference form, and then extrapolating the grid point field variables to the subvolume surfaces surrounding each point to enable the required surface integration, thereby ensuring local field equation satisfaction in the integral sense.

The simplicity and demonstrated stability of the finite-volume method in fluid mechanics applications has motivated the transition of this technique to solid mechanics problems during the past 35 years as an alternative to the finite-element approach. For static elasticity-type problems this reduces to the satisfaction of the equilibrium equations in the integral sense within subvolumes of the discretized analysis domain,

$$\int_{V_q} \left( \frac{\partial \sigma_{ji}}{\partial x_j} + F_i \right) dV_q = \int_{S_q} \sigma_{ji} n_j dS_q + \int_{V_q} F_i dV_q = 0 \tag{1.1}$$

where $n_j$ are components of the unit normal to the bounding surface $S_q$ of the subvolume $V_q$, and Gauss' Theorem was employed to convert the volume integral of stress divergence to the surface integral of traction components. Three versions of this technique can be identified in the analysis of solid mechanics problems, as discussed by Cavalcante et al. [26]. These versions are characterized by different subvolume discretization of the investigated domain and different displacement field representations within subvolumes, which lead to different manner of approximating field variables along subvolume surfaces.

The first two approaches, known as the cell-centered and cell vertex finite-volume techniques originally developed for homogeneous materials and structures, were motivated by the established finite-volume technique for fluid mechanics problems and elements of the finite-element method. The cell-centered finite-volume method is similar to the original fluid mechanics version and employs subvolumes which are centered around grid points at which field variables are defined. Initially, structured meshes based on rectangular or cylindrical subvolumes had been used for domain discretization, which were subsequently generalized to unstructured meshes with arbitrary subvolume topology based on polyhedral shapes. The cell vertex, or vertex based, finite-

volume approach leverages elements of the finite-element method in domain discretization and displacement field approximation. The domain is first discretized into finite elements, and the common vertices of adjacent elements provide grid points at which field variables are defined using shape functions borrowed from the finite-element approach. Subvolumes centered around grid points are then constructed taking contributions from elements with common vertices and using element and face centers as subvolume corners. Thus the subvolume geometry and displacement field approximation are directly linked to element discretization and employed shape functions. Satisfaction of the local equilibrium equations is carried out over all subvolumes containing every common vertex shared by adjacent elements forming grid points. Arbitrarily shaped polygonal control volumes may thus be constructed based on the chosen element type used to mesh the analysis domain.

As discussed by Pindera et al. [27], the third version of the finite-volume method evolved independently and nearly in parallel to model materials with heterogeneous microstructures, including periodic and functionally graded materials. The structural finite-volume theory has its origins in the so-called Higher-Order Theory for Functionally Graded Materials (HOTFGM), developed in a sequence of papers in the 1990's. This theory provided the main framework for the construction of its homogenized counterpart initially named the Higher-Order Theory for Periodic Multiphase Materials. The structural and homogenized versions of these so-called higher-order theories were subsequently re-constructed in a sequence of papers by Pindera and co-workers by simplifying the discretization of analysis domain into rectangular subvolumes which, in turn, facilitated implementation of the efficient local/global stiffness matrix approach, cf., Ref. [28-30]. The re-constructed theories were further extended by incorporating parametric mapping to enable efficient modeling of complex microstructures using quadrilateral subvolumes, cf., Ref. [31-33].

The reconstructed finite-volume theories are similar to the cell-centered techniques that evolved in parallel for homogeneous materials and structures during the same time frame. However, in contrast with the early cell-centered techniques, the re-constructed theories employ explicit displacement field approximation within individual subvolumes and follow an elasticity-based approach in satisfying interfacial displacement and traction continuity conditions in a surface-averaged sense. This is consistent with the satisfaction of equilibrium equations in a surface-

averaged sense and leads to explicit construction of local stiffness matrices for individual subvolumes which, in turn, substantially reduces the number of unknown variables, and allows direct comparison with the finite-element method. Assembly of local stiffness matrices into the global stiffness matrix is then performed such that continuity of surface-averaged tractions and displacements is satisfied. The satisfaction of both traction and displacement continuity across subvolume faces produces a robust solution technique that naturally accommodates heterogeneous material microstructures. A review of the finite-volume method in solid mechanics applications has been recently provided by Cardiff and Demirdzic [34].

In this thesis, the original version of FVDAM based on rectangular or square discretization of the RUC material microstructure developed by Bansal and Pindera [29] will be employed because it is better suited for the generation of thousands of microstructural realizations with random fiber distributions than the parametric version.

## 1.4 Objectives

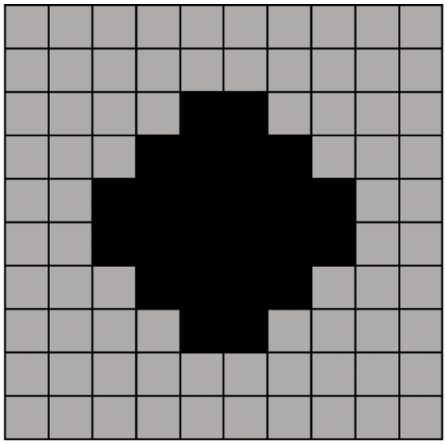
In this thesis, we will explore the performance of two different types of neural network architectures in predicting the homogenized elastic and elastic-plastic stress-strain response of random fiber composites using an extensive set of data generated by FVDAM adopted for this purpose. The two architectures are the artificial neural network (ANN) and the convolutional neural network (CNN) which accept two different types of inputs to establish microstructure-homogenized property relationships. These two architectures are described in Chapter 4.

The ANN architecture maps microstructural details of a RUC described by a few parameters (fiber number, fiber centers and radii, and fiber volume fraction) to the homogenized stiffness matrix and stress-strain curve that corresponds to a particular loading direction. This is the simplest representation of the microstructure. In contrast, the CNN architecture maps all of the microstructural details of a RUC as perceived by direct visual observation, or simply the RUC image, to the homogenized stress-strain curve by identifying and extracting pertinent geometric features. Hence the number of parameters defining the analyzed microstructure is substantially greater. The questions that will be addressed in the investigation include the differences in the convergence, accuracy, and efficiency of the two algorithms.

A related question is the speed-up obtained from the ML algorithms relative to the FVDAM-based simulations. Even though FVDAM is an efficient homogenization scheme in stand-alone applications, the generation of a homogenized stress-strain response is on the order of minutes, depending on the complexity of the analyzed RUC microstructure and the length of the load path. Hence it cannot be efficiently employed in large-scale structural applications as a user-defined subroutine to generate homogenized response at a point in the analyzed structure. Reducing such calculations to a fraction of a second using a trained and accurate ML-based algorithm would produce an enabling multiscale computational capability. This thesis aims to determine which of the two approaches is more accurate and efficient in predicting the homogenized response of random fiber composites.

## 1.5 Thesis Outline

The thesis is organized as follows. Chapter 2 describes the computational engine that generates homogenized stress-strain response of a RUC containing random fiber distributions, as well as the automated manner of random microstructure realizations for use in FVDAM simulations that enables rapid generation of thousands of unit cells containing randomly distributed fibers with a fixed volume fraction. The microstructures are defined by material assignment matrices, with entries corresponding to square subvolumes into which the unit cell is subdivided. Each entry defines either the matrix or fiber phase contained within the corresponding subvolume, Fig. 1.3. This information is passed to FVDAM to generate both the elastic and elastic-plastic response for a given loading direction. Chapter 3 presents homogenized elastic moduli and elastic-plastic stress strain responses under six unidirectional stress loading paths obtained from the thousands of generated microstructural realizations, including extensive results that illustrate for the first time the effect of random microstructures on the extent of scatter observed in the homogenized elastic-plastic responses. The extensive data generated in Chapter 3 is then employed to train and predict the homogenized stress-strain response of random unidirectional composites. Chapter 4 describes two different ML algorithms employed for this purpose whereas Chapter 5 contains the predicted results. The main contributions and conclusions of this investigation are summarized in Chapter 6.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 1 | 1 |
| 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 |
| 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 |
| 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

*Figure 1.3 An example of a simple two-material microstructure and the corresponding material assignment matrix*

# Chapter 2

# FVDAM and Microstructural Realization

In this chapter, we describe the computational homogenization engine called FVDAM employed to generate random microstructures of unidirectionally reinforced composites. This homogenization approach is particularly convenient for generating regular as well as large numbers of random microstructures due to the manner in which the microstructural features are mimicked through a subdivision of the unit cell into subvolumes. The different phases or constituents contained within the unit cell are represented by numbers that correspond to materials with different thermo-mechanical moduli assigned to different subvolumes using the material assignment matrix which reflect the unit cell subdivision into rows and columns. The subvolumes may also be interpreted as pixels for construction of a CNN architecture for machine learning purposes. Examples of two regular arrays and one random array of unidirectional composites are illustrated in Fig. 2.1.

The next two sections describe the main features of FVDAM and the automated algorithm employed to generate random microstructures. The FVDAM description follows Ref. [29] whereas the random microstructure generator is based on the recent contribution of Adakroy et al. [35]. The third section describes the python code that employs the generated microstructures to simulate the elastic-plastic response of the unit cell along different loading paths, and the final section summarizes the contributions described in this chapter. The construction of the random microstructure generator and the master code that enables automated simulations of the homogenized response of thousands of unit cells with random microstructures illustrated in Chapter 3 are the two new contributions that set the stage for the implementation of machine learning algorithms described in Chapter 4 and implemented in Chapter 5.

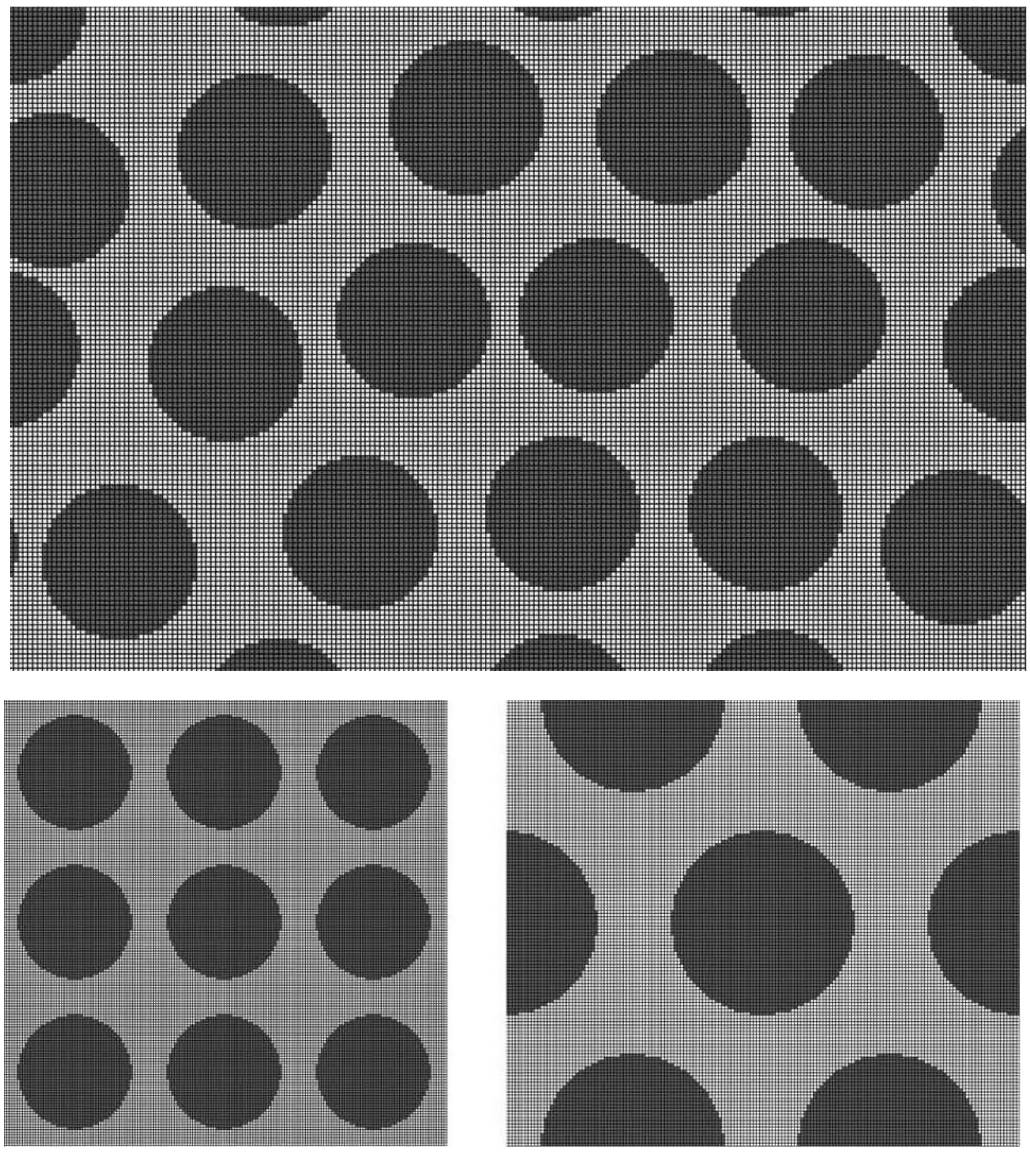


*Figure 2.1. Examples of random (top) and regular (bottom: square and hexagonal arrays) microstructures that may be simulated by FVDAM.*

## 2.1 Finite-Volume Direct Averaging Micromechanics

The original FVDAM theory contributed by Basal and Pindera [29] employs a rectangular grid to mimic the actual microstructure of the repeating unit cell RUC that defines a periodic multiphase material with continuous reinforcement along the $x_1$ axis, Fig. 2.2. The unit cell microstructure is made up of any number of arbitrarily distributed phases that produce fully anisotropic response in the $x_2 - x_3$ plane. unit cell. The global coordinates $(x_1, x_2, x_3)$ are used to

describe the homogenized response of the periodic array, whereas he local coordinates $(y_1, y_2, y_3)$ are associated with the unit cell. The subvolumes employed to discretize the unit cell are labelled (β, γ). The indices $\beta = 1, \ldots, N_\beta$ and $\gamma = 1, \ldots, N_\gamma$ which span the unit cell along the local $y_1$ and $y_2$ axes, respectively, identify the (β, γ) subvolume in the $y_2 - y_3$ plane.

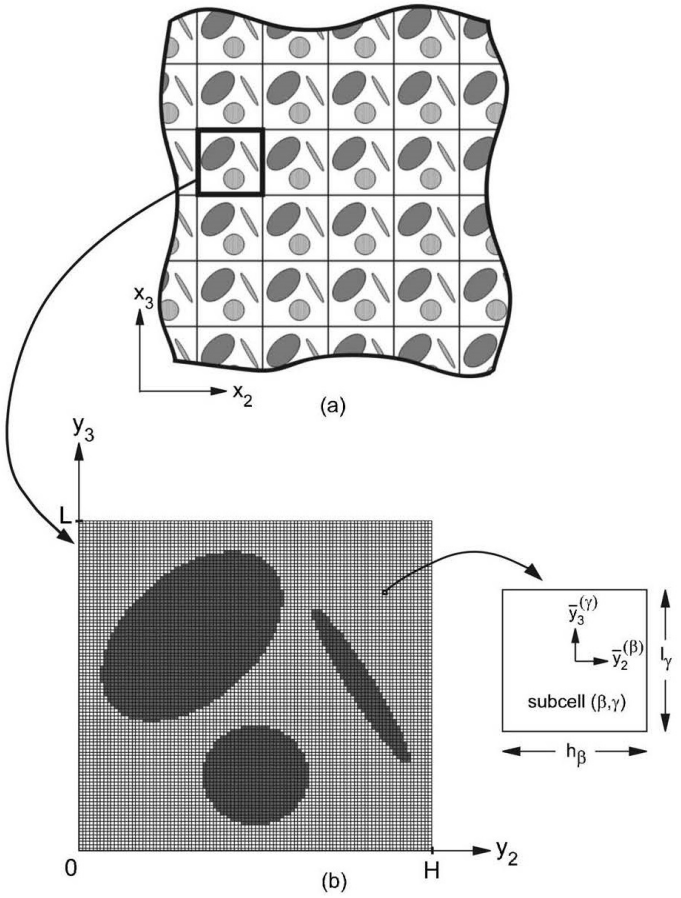



*Figure 2.2. A periodic array of differently shaped fiber reinforcement embedded in a matrix phase, and discretization of the repeating unit cell into square or rectangular subvolumes.*

The local coordinates $\bar{y}_2^{(\beta)}$, $\bar{y}_3^{(\gamma)}$ attached to the subvolume's center specify locations within the particular (β, γ) subvolume. The subvolume dimensions along the $y_2$ and $y_3$ axes are $h_\beta$ and $l_\gamma$, respectively, such that the overall RUC dimensions H and L are: $H = \sum_{\beta=1}^{N_\beta} h_\beta$ and $L = \sum_{\gamma=1}^{N_\gamma} l_\gamma$.

The displacement field in each subvolume is approximated by a two-scale expansion in terms of global and local coordinates,

$$u_i^{(\beta,\gamma)}(x,y) = \bar{\varepsilon}_{ij}x_j + u_i'^{(\beta,\gamma)}(y) \qquad i = 1,2,3 \tag{2.1}$$

where $\bar{\varepsilon}_{ij}$ are the specified homogenized strains which play the role of loading parameters, and the fluctuating displacement field components are given in terms of the local coordinates $\left(\bar{y}_2^{(\beta)}, \bar{y}_3^{(\gamma)}\right)$ attached to the subvolume's centroid,

$$u_i'^{(\beta,\gamma)} = W_{i(00)}^{(\beta,\gamma)} + \bar{y}_2^{(\beta)}W_{i(10)}^{(\beta,\gamma)} + \bar{y}_3^{(\gamma)}W_{i(01)}^{(\beta,\gamma)} + \frac{1}{2}\left(3\bar{y}_2^{(\beta)2} - \frac{h_\beta^2}{4}\right)W_{i(20)}^{(\beta,\gamma)} + \frac{1}{2}\left(3\bar{y}_3^{(\gamma)2} - \frac{l_\gamma^2}{4}\right)W_{i(02)}^{(\beta,\gamma)} \tag{2.2}$$

The unknown fifteen coefficients $W_{i(mn)}^{(\beta,\gamma)}$ are expressed in terms of the surface-averaged displacements, leading to the construction of local stiffness matrices for each subvolume, which are then assembled into the global stiffness matrix for the unit cell such that the tractions and displacements are satisfied in a surface-averaged sense. The construction of the local stiffness matrix involves the satisfaction of the subvolume equilibrium equations in the surface averaged sense.

### 2.1.1 Local stiffness matrix

The local stiffness matrix for the (β, γ) subvolume is constructed by relating the surface-averaged fluctuating displacements to the surface-averaged tractions on each face of the subvolume. The surface-averaged displacements on the four faces of the (β, γ) subvolume are defined by

$$\hat{u}_i'^{2\pm(\beta,\gamma)} = \frac{1}{l_\gamma}\int_{-l_\gamma/2}^{+l_r/2} u_i'^{(\beta,\gamma)}\left(\pm\frac{h_\beta}{2}, \bar{y}_3^{(\gamma)}\right)\bar{d}y_3^{(\gamma)} \;,\; \hat{u}_i'^{3\pm(\beta,\gamma)} = \frac{1}{h_\beta}\int_{-l_\gamma/2}^{+l_r/2} u_i'^{(\beta,\gamma)}\left(\pm\frac{h_\beta}{2}, \bar{y}_2^{(\beta)}\right)\bar{d}y_2^{(\beta)} \tag{2.3}$$

Performing the above surface averaging yields relations between the surface-averaged fluctuating displacements and the unknown coefficients $W_{i(mn)}^{(\beta,\gamma)}$ in the subvolume displacement field approximation. The corresponding traction components, given in terms of stresses through the Cauchy's relations

$$t_i^{(\beta,\gamma)} = \sigma_{ji}^{(\beta,\gamma)}n_i^{(\beta,\gamma)} \tag{2.4}$$

are obtained from the generalized Hookeís law for the (β, γ) subvolume,

$$\sigma_{ij}^{(\beta,\gamma)} = C_{ijkl}^{(\beta,\gamma)}\left(\varepsilon_{kl}^{(\beta,\gamma)} - \varepsilon_{kl}^{p(\beta,\gamma)}\right) \tag{2.5}$$

where the plastic behavior is limited to isotropic subvolumes, whereas the strictly elastic subvolumes may be orthotropic or (transversely) isotropic. The subvolume strains are given in terms of the macroscopic and fluctuating strain components upon use of the strain-displacement relations

$$\varepsilon_{ij}^{(\beta,\gamma)} = \bar{\varepsilon}_{ij} + \frac{1}{2}\left(\frac{\partial u_i'^{(\beta,\gamma)}}{\partial y_j} + \frac{\partial u_j'^{(\beta,\gamma)}}{\partial y_i}\right) \tag{2.6}$$

The surface-averaged tractions on the four faces of the (β, γ) subvolume are defined in the same manner as the corresponding fluctuating displacements,

$$\hat{t}_i'^{2\pm(\beta,\gamma)} = \frac{1}{l_\gamma}\int_{-l_\gamma/2}^{+l_r/2} t_i'^{(\beta,\gamma)}\left(\pm\frac{h_\beta}{2}, \bar{y}_3^{(\gamma)}\right)\bar{d}y_3^{(\gamma)}\,,\ \hat{t}_i'^{3\pm(\beta,\gamma)} = \frac{1}{h_\beta}\int_{-l_\gamma/2}^{+l_r/2} t_i'^{(\beta,\gamma)}\left(\pm\frac{h_\beta}{2}, \bar{y}_2^{(\beta)}\right)\bar{d}y_2^{(\beta)} \tag{2.7}$$

Performing the above surface averaging, the surface-averaged traction components are obtained in terms of the first and second order unknown coefficients $W_{i(10)}^{(\beta,\gamma)}, W_{i(01)}^{(\beta,\gamma)}, W_{i(20)}^{(\beta,\gamma)}, W_{i(02)}^{(\beta,\gamma)}$. Using the definitions for the surface-averaged fluctuating displacement, these coefficients are expressed in terms of the surface-averaged displacements and the remaining unknown zero order coefficients $W_{i(00)}^{(\beta,\gamma)}$. Satisfaction of the subvolume equilibrium equations in the surface-average sense

$$\int_{S_{(\beta,\gamma)}} t_i^{(\beta,\gamma)} dS_{(\beta,\gamma)} = 0 \tag{2.8}$$

produces the remaining set of relations between the surface-averaged fluctuating displacements and zero order coefficients, enabling the construction of the local stiffness matrix that relates the surface-averaged fluctuating displacements to the corresponding surface-averaged tractions,

$$\hat{t}^{(\beta,\gamma)} = K^{(\beta,\gamma)}\hat{u}'^{(\beta,\gamma)} + \Delta C^{(\beta,\gamma)}\bar{\varepsilon} + g^{(\beta,\gamma)} \tag{2.9}$$

where the local $\boldsymbol{\Delta C}^{(\beta,\gamma)}$ is comprised of the differences in the material stiffness matrices of adjacent subvolumes. Explicit expressions for the elements of the local stiffness matrix $K^{(\beta,\gamma)}$ and plastic vectors $g^{(\beta,\gamma)}$ which contain integrals of plastic strains have been provided in closed form by Bansal and Pindera [29].

### 2.1.2 Local stiffness matrix

Assembly of the local stiffness matrices by enforcing the continuity of both surface-averaged displacements and tractions, together with periodic boundary conditions, produces a global system of equations for the unknown surface-averaged fluctuating displacements. In the local/global stiffness matrix approach, the redundant displacement continuity equations are eliminated by setting the surface-averaged displacements at the interfaces associated with the adjacent subvolumes (β, γ), (β+1, γ) and (β, γ), (β, γ+1) to common unknowns,

$$\hat{u}_i'^{2+(\beta,\gamma)} = \hat{u}_i'^{2-(\beta+1,\gamma)} = \hat{u}_i'^{2(\beta+1,\gamma)} \; and \; \hat{u}_i'^{3+(\beta,\gamma)} = \hat{u}_i'^{3-(\beta,\gamma+1)} = \hat{u}_i'^{3(\beta+1,\gamma)} \tag{2.10}$$

for $i$ = 1, 2, 3, upon application of the traction continuity conditions at these common interfaces

$$\hat{t}_i'^{2+(\beta,\gamma)} + \hat{t}_i'^{2-(\beta+1,\gamma)} = 0 \quad and \quad \hat{t}_i'^{3+(\beta,\gamma)} + \hat{t}_i'^{3-(\beta,\gamma+1)} = 0 \tag{2.11}$$

The above relations hold true at $\beta = 1, \dots, N_\beta - 1$ and $\gamma = 1, \dots, N_\gamma - 1$ subvolume interfaces, producing $(3N_\beta - 1)N_\gamma + 3(N_\gamma - 1)N_\beta$ equations containing $(3N_\beta - 1)N_\gamma + 3(N_\gamma - 1)N_\beta$ unknown interfacial surface-averaged displacements in the unit cell's interior and $6(N_\beta + N_\gamma)$ surface-averaged displacements at the external boundaries. The additional equations necessary for the determination of the $6N_\beta N_\gamma$ unknown surface-averaged displacements are obtained from the periodicity conditions imposed on the fluctuating surface-averaged boundary displacements,

$$\hat{u}_i'^{2(1,\gamma)} = \hat{u}_i'^{2(N_\beta+1,\gamma)} \quad and \quad \hat{u}_i'^{3(\beta,1)} = \hat{u}_i'^{3(\beta,N_\gamma+1)} \tag{2.12}$$

and surface-averaged boundary tractions

$$\hat{t}_i'^{2(1,\gamma)} + \hat{t}_i'^{2(N_\beta+1,\gamma)} = 0 \quad and \quad \hat{t}_i'^{3(\beta,1)} + \hat{t}_i'^{3(\beta,N_\gamma+1)} = 0 \tag{2.13}$$

Imposition of the interfacial traction and displacement continuity conditions at the common subvolume faces, together with the periodic boundary conditions, produces the global system of equations for the determination of the common surface-averaged fluctuating displacements

$$K\widehat{\boldsymbol{U}}' = \Delta \boldsymbol{C}\bar{\varepsilon} + \boldsymbol{G} \tag{2.14}$$

where the vector $\widehat{\boldsymbol{U}}'$ contains all the unknown fluctuating surface-averaged displacements, the global $\Delta\boldsymbol{C}$ matrix is comprised of the differences in the material stiffness matrices of adjacent subvolumes, and the vector **G** contains integrals of plastic strains in the subvolumes. The global stiffness matrix singularity is eliminated by constraining the four corner subvolume faces in order to remove rigid body displacements. The remaining interfacial surface-averaged displacements are then determined by solving the reduced stiffness matrix system of equations iteratively at each load increment, given that vector **G** contains surface-averaged plastic strains which depend implicitly on surface-averaged displacements.

### 2.1.3 Homogenized Hooke's Law

The solution for the unknown fluctuating surface-averaged displacements $\widehat{\boldsymbol{U}}'$ at each point $\bar{\varepsilon}$ along the load path enables calculation of the subvolume volume-average strains $\bar{\varepsilon}_{ij}^{(\beta,\gamma)}$. These strains are then related to the applied homogenized strains $\bar{\varepsilon}$ and plastic effects through the localization relations

$$\bar{\varepsilon}_{ij}^{(\beta,\gamma)} = \boldsymbol{A}^{(\beta,\gamma)}\bar{\varepsilon} + \boldsymbol{D}^{(\beta,\gamma)} \tag{2.15}$$

where $A^{(\beta,\gamma)}$ are Hill's elastic strain concentration matrices found by the successive application of one macroscopic strain component at a time without considering plastic deformations. The load path-dependent vector $D^{(\beta,\gamma)}$ contains plastic contributions to the $(\beta,\gamma)$ subvolume deformation and is generated by solving Eq. (2.14) at each increment of the applied macroscopic strain in the manner described in the following subsection. Use of the localization relations at each converged sequence of iterations in the expression for the average composite stress

$$\bar{\sigma} = \frac{1}{HL}\sum_{r=1}^{N_\gamma}\sum_{\beta=1}^{N_\beta} l_\gamma h_\beta \bar{\sigma}^{(\beta,\gamma)} \tag{2.16}$$

where $\bar{\sigma}^{(\beta,\gamma)}$ is obtained from Eq. (2.5), produces the homogenized Hooke's law in the form,

$$\bar{\sigma} = C * (\bar{\varepsilon} - \bar{\varepsilon}^p) \tag{2.17}$$

where the homogenized stiffness matrix $\boldsymbol{C}^*$ is given by,

$$C^* = \frac{1}{HL}\sum_{r=1}^{N_\gamma}\sum_{\beta=1}^{N_\beta} l_\gamma h_\beta C^{(\beta,\gamma)} A^{(\beta,\gamma)} \tag{2.18}$$

and the homogenized plastic strain $\bar{\varepsilon}^p$ is

$$\bar{\varepsilon}^p = \frac{[C^*]^{-1}}{HL}\sum_{r=1}^{N_\gamma}\sum_{\beta=1}^{N_\beta} l_\gamma h_\beta(\bar{\varepsilon}^{(\beta,\gamma)} - D^{(\beta,\gamma)}) \tag{2.19}$$

### 2.1.4 Numerical iterative scheme

The plastic strain fields depend on the loading history and thus the unknown surface-averaged fluctuating displacements depend implicitly on the evolving subvolume plastic fields. Hence the solution of Eq. (2.14) is obtained iteratively at each point along the load history defined by the imposed homogenized strain $\bar{\varepsilon}$. At each load increment, the surface plastic strains are calculated using Mendelson's technique, Mendelson [36], wherein the point-wise plastic strains within the reference subvolume are decomposed into converged contributions from the previous load step plus increments that result from the imposed load increment

$$\varepsilon_{ij}^{p(\beta,r)}\left(\bar{y}_2^{(\beta)}, \bar{y}_3^{(\gamma)}\right) = \varepsilon_{ij}^{p(\beta,r)}\left(\bar{y}_2^{(\beta)}, \bar{y}_3^{(\gamma)}\right)|_{previous} + d\varepsilon_{ij}^{p(\beta,\gamma)}(\bar{y}_2^{(\beta)}, \bar{y}_3^{(\gamma)})$$

Plastic strain increments are calculated using the classical plasticity theory with isotropic hardening based on the Prandtl-Reuss equations reformulated by Mendelson [36 in terms of so-called modified total strain deviators $e'_{ij}$, rather than deviatoric stresses, as follows

$$d\varepsilon_{ij}^p = \frac{e'_{ij}}{\bar{e}_{dff}} d\bar{\varepsilon}^p \tag{2.20}$$

where $e'_{ij} = \varepsilon_{ij} - \frac{1}{3\varepsilon_{kk}\delta_{ij}} - \varepsilon_{ij}^p|_{previous}$, $\bar{e}_{eff} = \sqrt{2/3 e'_{ij} e'_{ij}}$, and the effective plastic strain increment $d\bar{\varepsilon}^p = \bar{e}_{eff} - \bar{\sigma}/3\mu$. The implementation of the reformulated equations is made very efficient by the plastic loading condition $1 - \frac{\bar{\sigma}}{3\mu\bar{e}_{eff}} > 0$, Williams and Pindera [37].

Once the converged solution to Eq. (2.14) is obtained at each point along the load path, the plastic influence matrices $\boldsymbol{D}^{(\beta,\gamma)}$ are calculated from Eq. (2.15), which produces the homogenized plastic strain $\bar{\varepsilon}^p$ in Eq. (2.17).

## 2.2 Microstructural Realization

The key to the generation of the homogenized response is the construction of the RUC. In the case of both finite-element and finite-volume based homogenization, this involves discretization of the RUC into an appropriate number of elements or subvolumes that mimic the RUC microstructure as closely as possible. For RUCs containing large numbers of fibers, such discretization is not a trivial matter, and typically involves a considerable amount of time and effort. Commercial automated mesh generation procedures have been developed for the finite-element method but require considerable work in interfacing them with the actual finite-element code. Automating a finite-element mesh generation procedure to enable thousands of microstructural realizations and subsequent execution in the elastic-plastic domain is a time-consuming matter, particularly if convergence of the solution must be verified. The finite-volume technique employed in this study based on rectangular subvolume discretization is a more efficient approach even when the fiber cross section boundary is approximated by a stair-case pattern. Given sufficient number of subvolumes, past work has shown that very accurate and converged results may be obtained both in the elastic and elastic-plastic domains due to the local satisfaction of the equilibrium equations even in the presence of constituent phases with very large moduli mismatch.

To rapidly realize thousands of microstructures required in this study, a computer code was developed that randomly distributes fibers of circular cross section within the unit cell of specified height and length for a given fiber volume fraction and number of fibers contained within. Once the microstructure is realized using randomly distributed fiber centers, it is then discretized into equally dimensioned subvolumes, and the material assignment matrix is created for input into FVDAM simulation. For a unit cell occupied by fibers of the same properties that are embedded in a matrix phase, the material assignment matrix contains two distinct numbers (0,1 or 1,2, say) which serve as labels that define the fiber and matrix properties. The fiber radii relative to the fiber centers provide boundaries around subvolumes that are assigned fiber labels, with the remaining subvolumes assigned matrix labels.

A naive initial approach to this problem employed recursion to randomly place fiber centers, stepping back when no space remained for an additional required fiber and continuing to choose

random coordinates among the set of free coordinates until all fibers were placed. This approach turned out to be inadequate for unit cells with relatively large numbers of fibers. For example, consideration of a unit cell with ten fibers and volume fraction 0.45 (this was a fairly common specification) indicated that this approach was likely to run forever with low probability of convergence, with an unpredictable and large runtime. As a result, a different method was implemented.

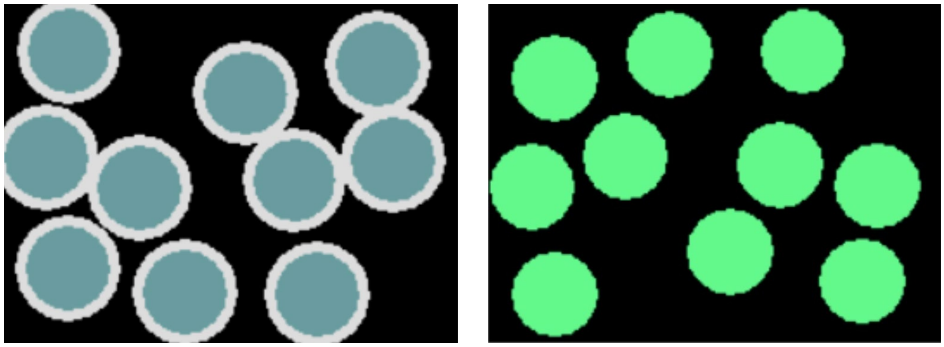


*Figure 2.3. Animation describing fiber placement technique: left-hand picture includes buffers; right-hand picture shows the final microstructure.*

The volume fraction, fiber count, and window area determine the radius of each fiber. First, fibers are placed evenly throughout the window. A buffer is added around each fiber in the form of a radial differential (pictured as silver rings in Fig. 2.3). Random displacements are assigned to each fiber and the fibers are allowed to translate. When two fibers come into contact, their displacements are adjusted to prevent overlap. Each fiber takes on the displacement of the fiber with which it came into contact. Similarly, if a fiber hits a unit cell boundary, a "bounce" is simulated by multiplying the appropriate displacement component by -1. For example, hitting the left boundary of a rectangular-shaped unit cell would produce the adjustment of the x displacement component.) Contacts were determined by finding distances between respective circle centers and walls and checking if any distances dropped below the radius. This procedure was continued until a specified number of steps or microstructural re-configurations had taken place. Figure 2.4 illustrates microstructural configurations at different steps for a procedure involving 1000 microstructural re-configurations. As observed, microstructural evolution after 200 steps appears

apparently random upon visual inspection. Any of the generated microstructures in this manner may be then discretized into rows and columns of subvolumes, or pixels, and the material assignment matrix created for input into FVDAM. For the microstructures comprised of 10 fibers, 1000 microstructural re-configurations require less than one second on a PC.

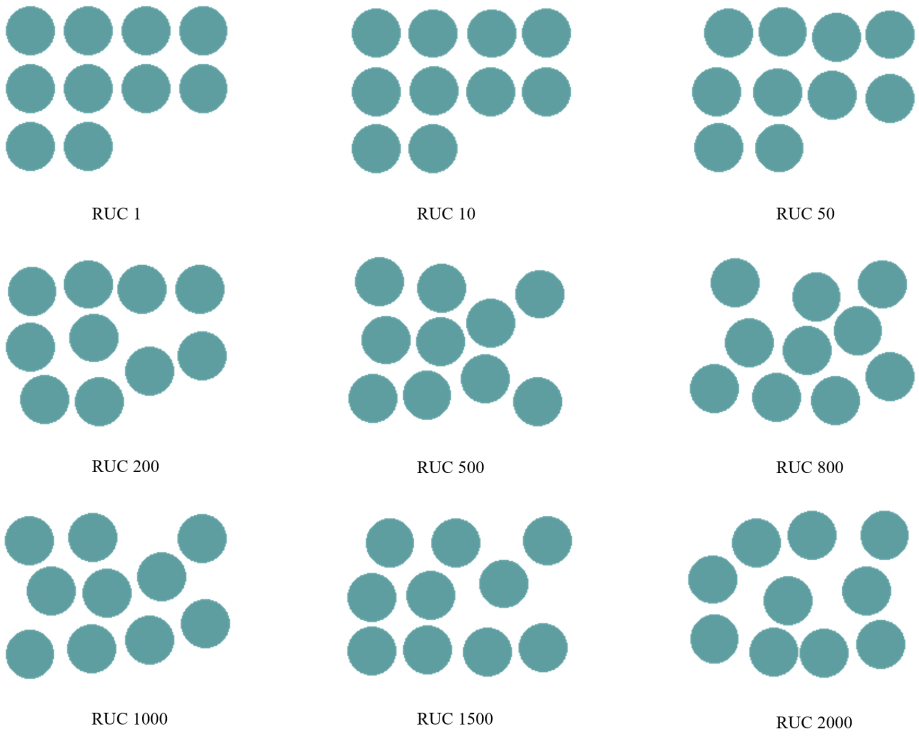



*Figure 2.4. Microstructural evolution produced by the generate_assignment_matrix.py script.*

## 2.3. Homogenized Response Simulation

The simulation of random microstructure unit cells has been automated in order to obtain homogenized stress-strain curves along a specified loading path for all generated microstructural realizations. Six loading paths are presently available that enable generation of homogenized stress-strain curves under uniaxial homogenized stress loading along the axial and transverse directions. The uniaxial loading is achieved by applying homogenized strains in appropriate ratios that yield the specified unidirectional homogenized stress loading. The user specifies the loading option and the number of microstructural realizations to be simulated. The assignment matrices for the

generated microstructural realizations are stored so that they may be simulated for different loading options. In this way, different loading options may be applied to the same set of microstructures for comparison.

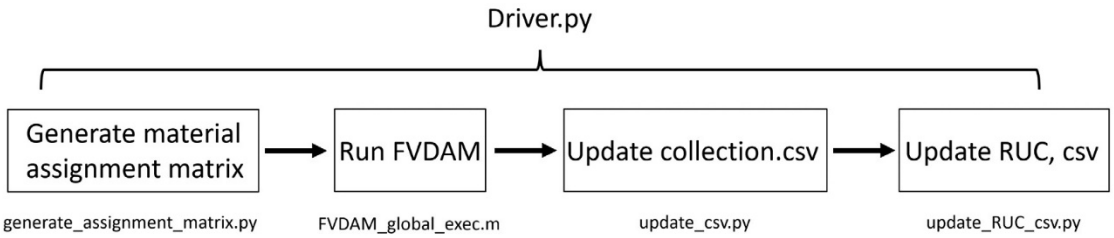



*Figure 2.5. The master script driver.py that automates the simulation of a specified sequence random microstructures by : automatically generating the material assignment matrix; executing FVDAM; writing the homogenized stress-strain curves for the given load option to a csv file; and storing the individual RUC microstructure data for future use*

The master script called automatic_matlab.py is the main calling program written in Python that generates the material assignment matrix, executes FVDAM, writes the homogenized stress-strain curves for the given load option to a file, and stores the individual RUC microstructure data, as illustrates schematically in Fig. 2.5. Key features of the master script include the use of both Python scripts and Matlab files. This is because FVDAM was written in Matlab, while all other forms of data manipulation were allocated to Python. FVDAM is run using a Python Matlab engine, as provided by Mathworks. The individual subscripts called by the master script are:

● *generate_assignment_matrix.py*: randomly distributes fibers within a unit cell of specified dimensions for the specified fiber volume fraction and fiber count. Then the script discretizes the result—based on provided window height and width—into a material assignment matrix in preparation for FVDAM execution. The output of this subscript are material assignment matrices for the specified number of random microstructural realizations.

● *FVDAM_global_exec.m*: executes FVDAM sequentially for each microstructural realization for the specified load path

● *update_csv.py*: homogenized stress-strain curves are saved in a cumulative CSV file that begins with a single column of incremental homogenized strains. Homogenized stresses corresponding to

the specified load option are recorded as additional columns, where each column corresponds to another RUC array, and each row corresponds to the incremental homogenized strain from the first column. The script update_csv.py opens the LOP_?_SIGMA??_collection.csv files and modifies it by extracting the target column in the FVDAM output and merges it with the current file.

● *update_RUG_csv.py*: stores individual RUC arrays. A new csv is created for every microstructure. The number of rows of these csv files matches the fiber count. The first column corresponds to discretized x coordinates, while the second column corresponds to discretized y coordinates.

Finally, a Matlab script was written to best visualize the data in the LOP_?_SIMGA??_collection csv file. The stress-strain curves match the broomstick distribution expected of the microstructures.

The above computational process is sufficiently optimized such that it is possible to receive useful results from a PC. For example, generating, storing and graphing 100 microstructural realizations of unit cells discretized into 159 × 211 subvolumes required approximately 2.5 hours. The master script requires the installation of a Python Matlab engine. It is important that the versions of Python and Matlab installed on the machine are appropriate for integration.

## 2.4 Summary

This chapter describes the computational homogenization engine called FVDAM employed to simulate the response of random microstructure unit cells of unidirectional composites along different uniaxial homogenized stress paths and the related Python-driven code that enables automated generation and subsequent simulations of the response of thousands of microstructural realizations for six uniaxial homogenized stress load paths. This capability facilitates the development of under- standing of the effect of microstructural randomness on the homogenized response for this class of composites in the elastic-plastic region. More importantly from the perspective of this thesis, the homogenized stress-strain responses of the generated microstructural realizations are available for implementation into ML-based algorithms that enable rapid

prediction of the response of random microstructural realizations of unidirectional composites without resorting to actual homogenization calculations which is the topic of subsequent chapters.

# Chapter 3

# Random Microstructure Homogenization

The solution of the system of equations for the unknown surface-averaged fluctuating displacements of a random microstructure RUC given by Eq. (2.14) in Chapter 2, in conjunction with Eq. (2.15) - (2.16), yields the homogenized Hooke's law reproduced below in symbolic form,

$$\bar{\sigma} = \boldsymbol{C}^*(\bar{\varepsilon} - \bar{\varepsilon}^p) \tag{3.1}$$

The homogenized strain components of the strain tensor $\bar{\varepsilon}$ is specified by the load option, the components of the homogenized plastic strain tensor $\bar{\varepsilon}^p$ are obtained from Eq. (2.19) at the converged step along the load path, and the homogenized stiffness matrix $\boldsymbol{C}^*$ is calculated just once from Eq. (2.18). The process of calculating the homogenized response along the specified load path is automated, as described in Chapter 2, enabling to generate thousands of homogenized stiffness matrices $\boldsymbol{C}^*$ and the homogenized stress-strain curves for each of the generated random microstructure. The results obtained from two thousand simulations are reported in this chapter which will be employed in Chapter 5 for training and predictive purposes.

Whereas the homogenized stiffness matrix $\boldsymbol{C}^*$ is calculated just once for each microstructural realization, the homogenized elastic-plastic response depends on the applied loading path. These homogenized responses have been generated under uniaxial stress loading defined by the six load options LOPs. Uniaxial stress loading is obtained by adjusting the homogenized strain components accordingly to produce the desired single non-zero homogenized stress component The load option designations are:

- LOP = 1: uniaxial loading by $\bar{\sigma}_{11}$, all other homogenized stress components are zero
- LOP = 2: uniaxial loading by $\bar{\sigma}_{22}$, all other homogenized stress components are zero
- LOP = 3: uniaxial loading by $\bar{\sigma}_{33}$, all other homogenized stress components are zero
- LOP = 4: uniaxial loading by $\bar{\sigma}_{23}$, all other homogenized stress components are zero

- LOP = 5: uniaxial loading by $\bar{\sigma}_{13}$, all other homogenized stress components are zero
- LOP = 6: uniaxial loading by $\bar{\sigma}_{12}$, all other homogenized stress components are zero

It is well known that for unidirectional composites, the uniaxial response along the fiber direction is not sensitive to the fiber distribution. Hence LOP = 1 is not expected to produce substantial differences in both the initial elastic response and post-elastic or elastic-plastic response of the generated random microstructures. This is due to the axial constraint $\bar{\varepsilon}_{11}^{matrix} = \bar{\varepsilon}_{11}^{fiber}$ and the relatively weak influence of the Poisson's ratio differential which does not significantly affect the axial load sharing between the fiber and matrix phases. In contrast, the load sharing between the fiber and matrix phases is much more affected by the fiber distributions along load paths in planes transverse to the axial reinforcement, yielding more substantial differences. Completely random fiber distributions are expected to produce homogenized stress-strain response that mimic those of transversely isotropic materials.

The new automated computational capability has been employed to generate two thousand microstructures with each RUC composed of ten fibers with the fiber volume fraction of 0.46 following Pindera and Bansal [38]. The material system was boron/aluminum with the fiber and matrix properties given in Table 3.1. Each RUC was discretized into 159× 211 subvolumes, with automatic fiber placement procedure described in Chapter 2.

| Material | $E$ (GPa) | $\nu$ | $\sigma_y$ (MPa) | $H_p$ (GPa) |
|---|---|---|---|---|
| Boron fiber | 400 | 0.2 | - | - |
| Derived in situ aluminum | 72.4 | 0.33 | 85 | 2.0 |

*Table 3.1: Elastic and plastic parameters of boron fiber and aluminum matrix, Ref. [38].*

## 3.1 Elastic Response - Homogenized Stiffness Matrix

For a unidirectional composite reinforced along the $x_1$ axis with a random distribution of fibers in the $x_2 - x_3$ plane, the structure of the homogenized stiffness matrix is that of a monoclinic material with a single plane of material symmetry perpendicular to the fiber direction $x_1$, namely

$$\mathbf{C}^{*}=\begin{bmatrix} C_{11}^{*} & C_{12}^{*} & C_{13}^{*} & C_{14}^{*} & 0 & 0 \\ C_{12}^{*} & C_{22}^{*} & C_{23}^{*} & C_{24}^{*} & 0 & 0 \\ C_{13}^{*} & C_{23}^{*} & C_{33}^{*} & C_{34}^{*} & 0 & 0 \\ C_{14}^{*} & C_{24}^{*} & C_{34}^{*} & C_{44}^{*} & 0 & 0 \\ 0 & 0 & 0 & 0 & C_{55}^{*} & C_{56}^{*} \\ 0 & 0 & 0 & 0 & C_{56}^{*} & C_{66}^{*} \end{bmatrix} \tag{3.2}$$

In the presence of three mutually orthogonal planes of symmetry, the coupling elements $C_{14}^{*} = C_{24}^{*} = C_{34}^{*} = C_{56}^{*}$=0 and the homogenized stiffness matrix structure becomes that of an orthotropic material. Further, if the $x_2 - x_3$ plane is a plane of isotropy, then the following relations ensue

$$C_{12}^{*} = C_{13}^{*},\ \ C_{22}^{*} = C_{33}^{*},\ \ \ C_{44}^{*} = \frac{1}{2}(C_{22}^{*} - C_{23}^{*}),\ \ C_{55}^{*} = C_{66}^{*} \tag{3.3}$$

reducing the number of independent material moduli to five referred to the principal material coordinate system.

Once the elements of $C^{*}$, the homogenized engineering moduli may be determined from the inverse relationship

$$S^{*} = [C^{*}]^{-1}$$

where the homogenized compliance matrix $S^{*}$ is expressed in terms of the homogenized engineering moduli,

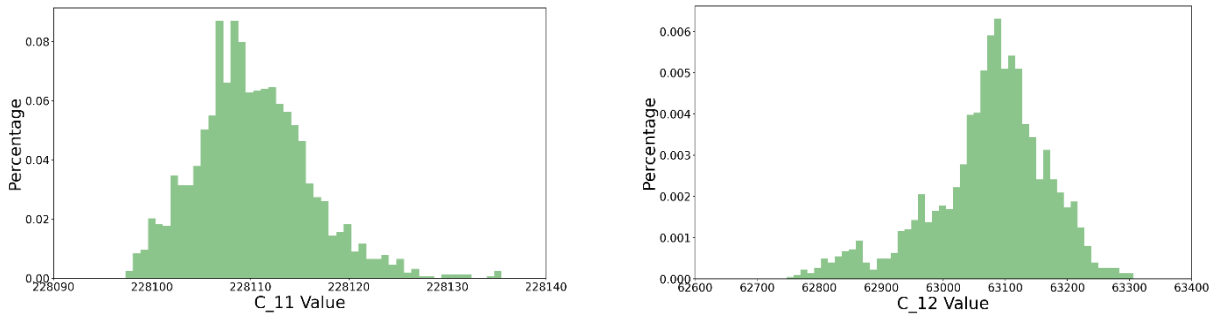
$$\mathbf{S}^{*}=\begin{bmatrix} 1/E_{11}^{*} & -\nu_{21}^{*}/E_{22}^{*} & -\nu_{21}^{*}/E_{33}^{*} & \eta_{1,23}^{*} & 0 & 0 \\ -\nu_{12}^{*}/E_{11}^{*} & 1/E_{22}^{*} & -\nu_{23}^{*}/E_{33}^{*} & \eta_{2,23}^{*} & 0 & 0 \\ -\nu_{13}^{*}/E_{11}^{*} & -\nu_{32}^{*}/E_{22}^{*} & 1/E_{33}^{*} & \eta_{3,23}^{*} & 0 & 0 \\ \eta_{23,1}^{*} & \eta_{23,2}^{*} & \eta_{23,3}^{*} & 1/G_{23}^{*} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/G_{13}^{*} & \eta_{13,12}^{*} \\ 0 & 0 & 0 & 0 & \eta_{12,13}^{*} & 1/G_{12}^{*} \end{bmatrix}$$

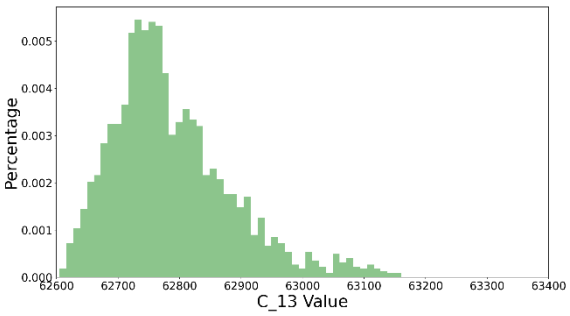
where $E_{11}^{*} = \frac{1}{S_{11}^{*}}, \nu_{12}^{*} = -\frac{S_{21}^{*}}{S_{11}^{*}}$, etc., and $\eta_{1,23}$ etc., are Lekhnitskii's coefficients of mutual influence which provide a measure of normal and shear strain coupling due to material's extent of anisotropy in the $x_2 - x_3$ plane.

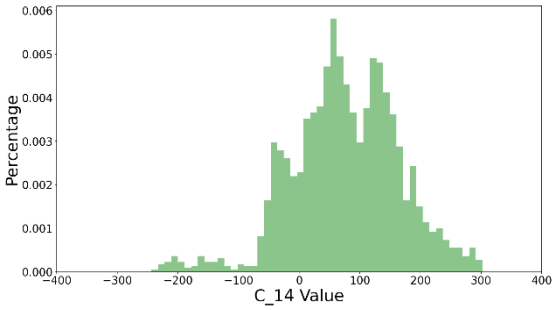
Figure 3.1 presents the density distributions of the thirteen homogenized stiffness matrix elements for the two thousand microstructures. The homogenized stiffness matrix is indeed symmetric and thus only thirteen elements are shown. The density distributions were calculated by dividing the two thousand outcomes for the homogenized stiffness matrix elements into fifty intervals ranging from the smallest to the largest value of each homogenized stiffness matrix element, and plotted against the number of occurrences normalized by the total number of microstructural realizations. Recall that microstructural randomness increases with microstructure realization number because the initial RUC microstructure, which was used to generate RUCs of increasing randomness, was regularly spaced, see Fig. 2.4. The mean values and standard deviations of these distributions are given in Table 3.2.

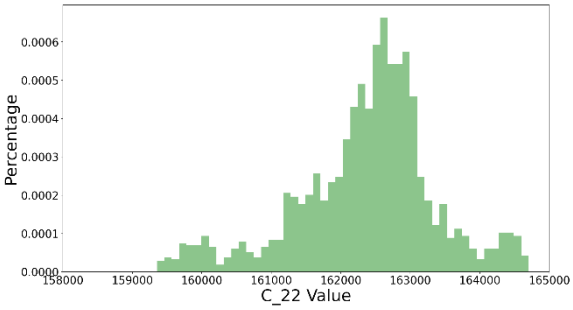
| | Mean Value (MPa) | Standard Deviation (MPa) |
|---|---|---|
| C_11 | 228110 | 6 |
| C_12 | 63074 | 95 |
| C_13 | 62787 | 98 |
| C_14 | 76 | 89 |
| C_22 | 162367 | 10002 |
| C_23 | 67218 | 591 |
| C_24 | 78 | 431 |
| C_33 | 160422 | 713 |
| C_34 | 441 | 541 |
| C_44 | 43631 | 496 |
| C_55 | 46727 | 355 |
| C_56 | 273 | 340 |
| C_66 | 47625 | 345 |

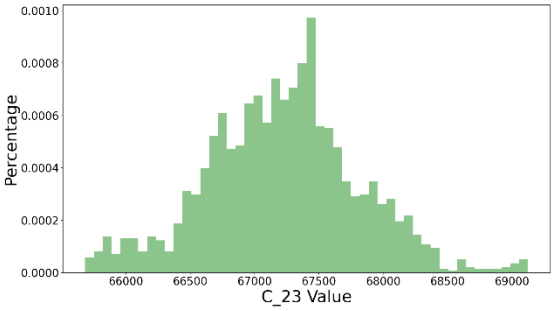
*Table 3. Mean values and standard deviations of 13 homogenized stiffness elements distributions*

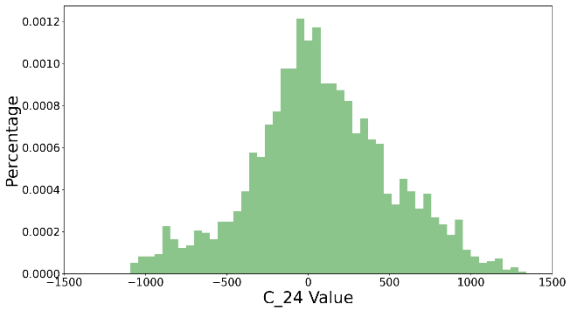


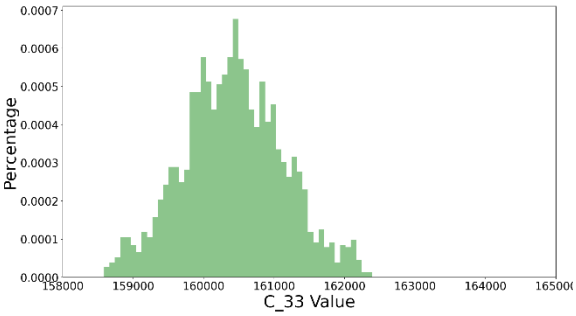

Percentage
C_13 Value

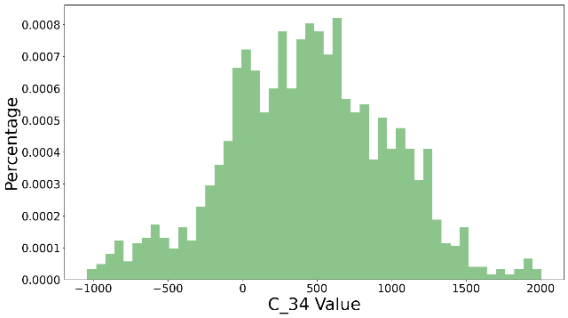

Percentage
C_14 Value

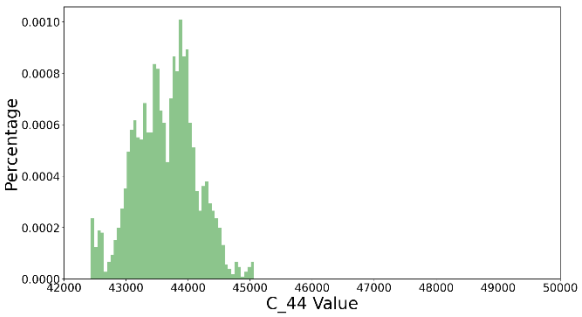

Percentage
C_22 Value


Percentage
C_23 Value


Percentage
C_24 Value


Percentage
C_33 Value


Percentage
C_34 Value


Percentage
C_44 Value

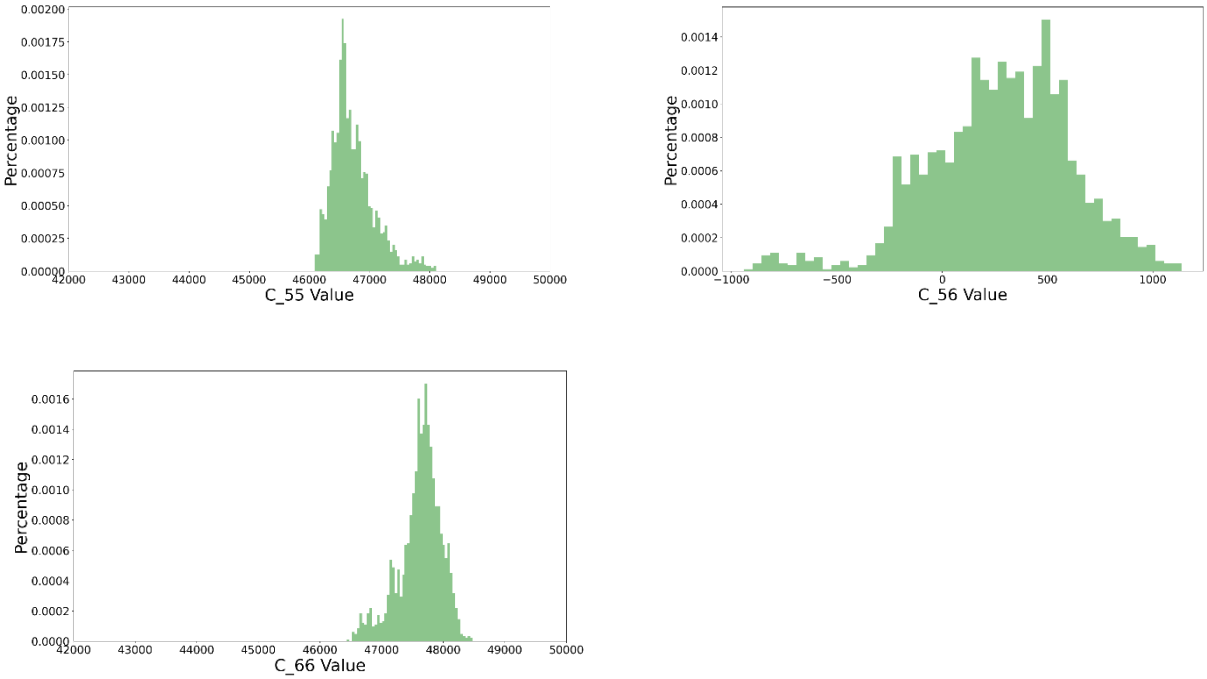



*Figure 3.1. Homogenized elastic moduli for the 2000 microstructural realizations graphed vs RUC number.*

## 3.2 Elastic-Plastic Response

Subsequently, homogenized responses of the two thousand RUC realizations were generated under uniaxial loadings specified by the LOP number. These responses are summarized in Fig. 3.2 for the six LOP cases and discussed below.

### 3.2.1 LOP 1 results

Figure 3.2(a) presents the homogenized stress-strain response due to loading by $\sigma_{11}$ only, with the remaining homogenized stresses set to zero. As observed, the initial elastic response is practically unaffected by the microstructural randomness, as also observed in the small variations of $C_{11}^*$ seen in Fig. 3.1. This homogenized stiffness matrix element is proportional to $E_{11}^*$ which defines the initial response under unidirectional loading by $\sigma_{11}$. Similarly, practically no impact of the fiber distribution is observed in the elastic-plastic region given the much stiffer boron fibers relatively to the aluminum matrix, the relatively large fiber volume fraction and the fiber constraint which produces $\bar{\varepsilon}_{11} = \bar{\varepsilon}_{11}^m = \bar{\varepsilon}_{11}^f$. This suggests that the matrix stress that controls plasticity in the elastic-plastic region is relatively constant and unaffected by the fiber distribution, producing nearly

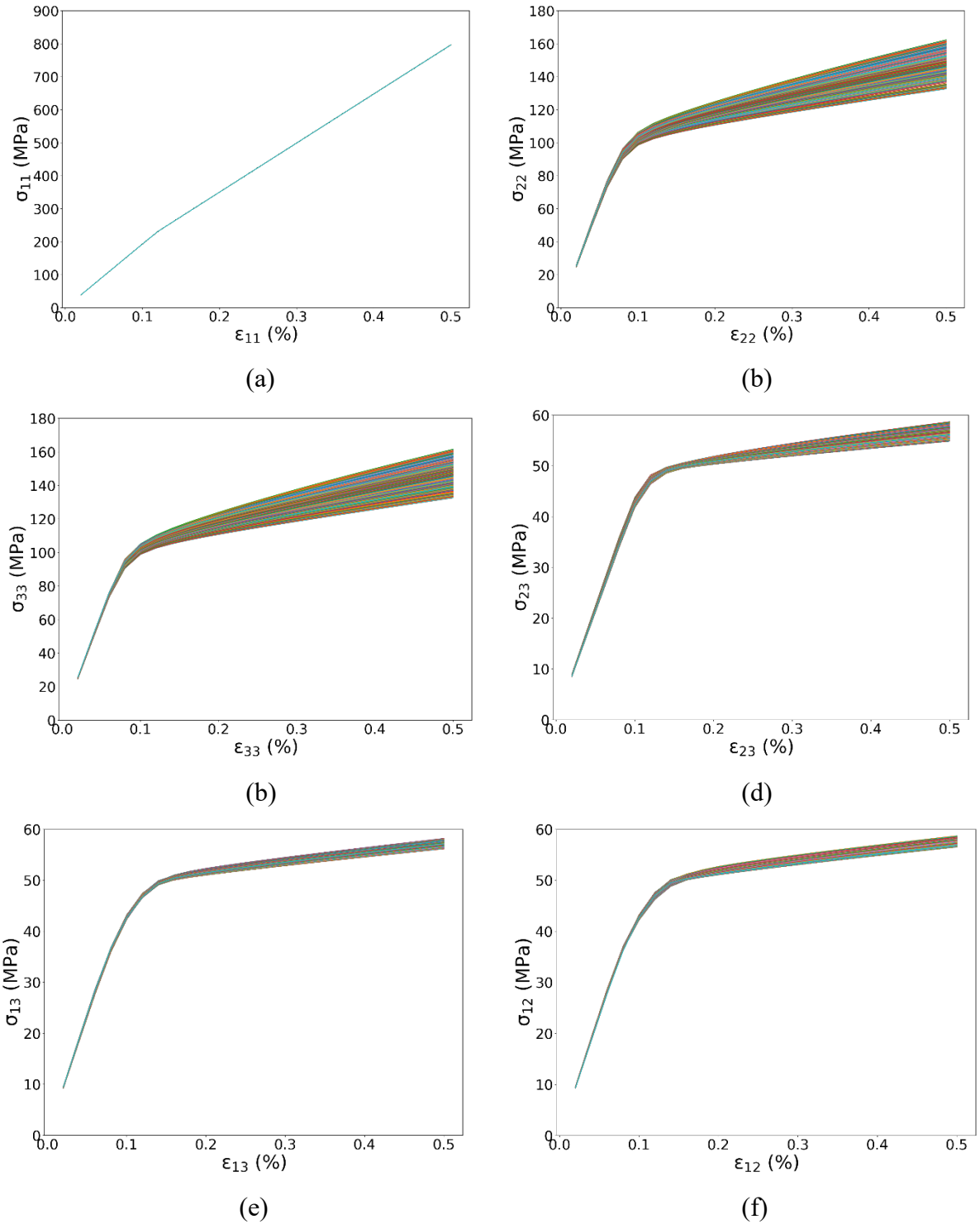



*Figure 3.2. Homogenized stress-stress responses for uniaxial loading by LOP = 1,…,6.*

uniform yielding throughout the entire matrix regardless of the microstructural details. This results in the bilinear character of the homogenized stress-strain curves $\bar{\sigma}_{11} - \bar{\varepsilon}_{11}$ which visually appear the same regardless of the fiber distribution.

### 3.2.2 LOP 2 results

Substantially greater effect of fiber randomness on both the homogenized elastic moduli and stress-strain responses is observed under uniaxial loading by the transverse stress $\sigma_{22}$. The onset and evolution of plasticity magnify the effect in the elastic-plastic region due to differences in the plastic field localization produced by the different fiber distributions. This results in the pattern of the homogenized stress-strain curves that resembles a broom stick, Fig. 3.2(b), with the upper and lower bounds substantially greater than those on the homogenized elastic modulus $C_{22}^{*}$ shown in Fig. 3.1. These substantial differences are produced by highly heterogeneous stress distributions that result in localized plastic strains which are significantly affected by fiber locations relative to the applied load. To illustrate the effect of random fiber distributions on the homogenized stress-strain response under this uniaxial loading, unit cell microstructures have been identified that produce stiffest, most compliant, and intermediate homogenized responses and will be discussed in the sequel.

### 3.2.3 LOP 3 results

As in the case of LOP 2, the homogenized stress-strain response under uniaxial loading by the transverse stress $\sigma_{33}$ exhibits substantial broom stick pattern in the elastic-plastic region, Fig. 3.2(c). The upper and lower bounds on the response are LOP 3 loading are similar to those observed under LOP 2 given that the microstructure is nearly transversely isotropic. The unit cell microstructures that correspond to the stiffest, most compliant and intermediate homogenized stress-strain responses will be illustrated in the sequel. These microstructures are similar to those that produce the corresponding homogenized responses under LOP 2 loading, supporting the transverse isotropy assumption.

### 3.2.4 LOP 4 results

Figure 3.2 (d) presents the homogenized stress-strain response due to uniaxial loading by transverse

shear $\sigma_{44}$ only, with the remaining homogenized stresses set to zero, As observed, the elastic-plastic responses also exhibit a broom stick pattern, but the extent of the difference between upper and lower bound responses is not as great as under LOP 2 and 3 loadings. The microstructures that produce stiffest, most compliant, and intermediate responses will be shown in the sequel.

### 3.2.5 LOP 5 and LOP 6 results

The homogenized stress-strain responses under uniaxial axial shear loadings in the $x_1 - x_2$ and $x_1 - x_3$ planes by $\bar{\sigma}_{12}$ and $\bar{\sigma}_{13}$, respectively, are shown in Figs. 3.2 (e) and 3.2 (f). Broom stick patterns are also observed in these cases, but the extent is even smaller than observed under LOP 4 loading. Little difference is observed in the two sets of responses, suggesting transverse isotropic behavior. The microstructures that produce stiffest, most compliant, and intermediate responses are illustrated in the sequel.

### 3.2.6 Effect of microstructure on the homogenized stress-strain response

In order to investigate the effect of fiber distribution on the homogenized stress-strain behavior under different uniaxial loadings, we compare the stiffest, most compliant and intermediate stress-strain responses generated by uniaxial loadings defined by the six LOP cases in Figs. 3.3 – 3.8. Figure 3.3 illustrates the responses for uniaxial loading by $\sigma_{11}$ only. As observed, it is impossible to visually differentiate the stiffest and softest responses regardless of the substantial variations in the RUC microstructures included in the Figure. As explained in the foregoing, plasticity initiates nearly uniformly throughout the entire RUC because of the axial constraint by the fiber, giving rise to indistinguishable response in the elastic-plastic region for the three different microstructures. This will be a good test for the capability of the investigated ML algorithms given that the responses under uniaxial loading transverse to the fibers exhibit broomstick appearance, the extent of which depends on the stress component.

Figure 3.4 illustrates the effect of RUC microstructure on the stiffest, intermediate and softest responses for uniaxial loading by transverse $\bar{\sigma}_{22}$ stress only oriented along the horizontal axis. The stiffest response is produced by the RUC with the most ordered microstructure characterized by aligned fiber rows and the softest by RUC microstructure with most disordered

fibers. The microstructure with some degree of order produces the intermediate stress-strain response.

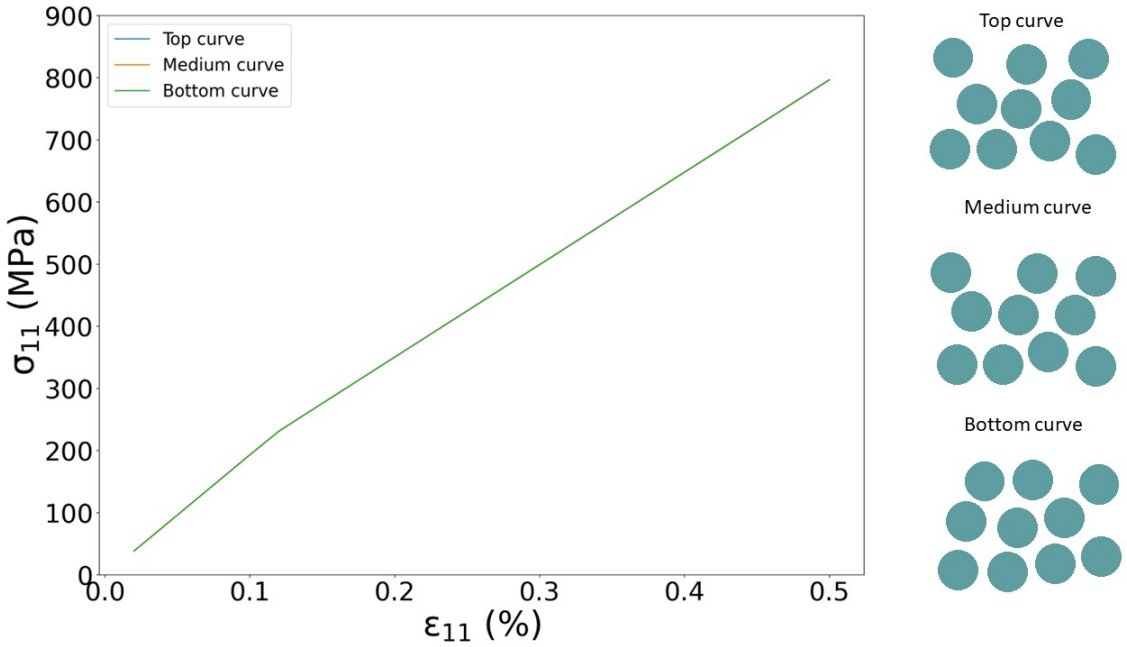



*Figure 3.3. Homogenized stress-stress curves for uniaxial loading by LOP = 1 and microstructures that produce stiffest, intermediate, and most compliant responses.*

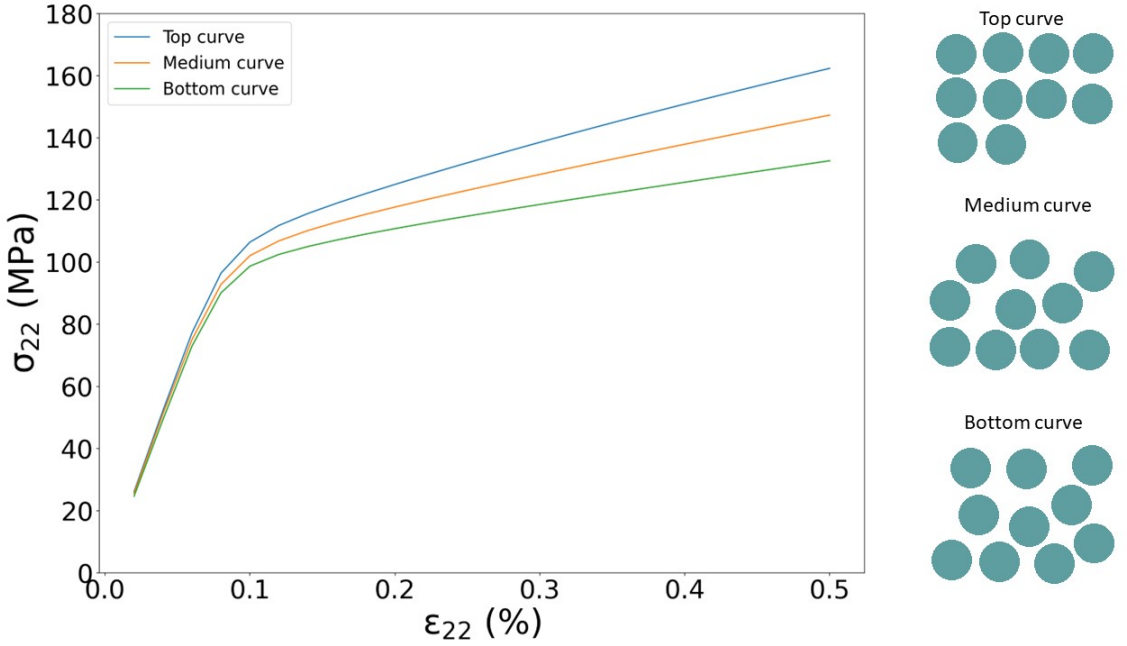



*Figure 3.4. Homogenized stress-stress curves for uniaxial loading by LOP = 2 and microstructures that produce stiffest, intermediate, and most compliant responses.*

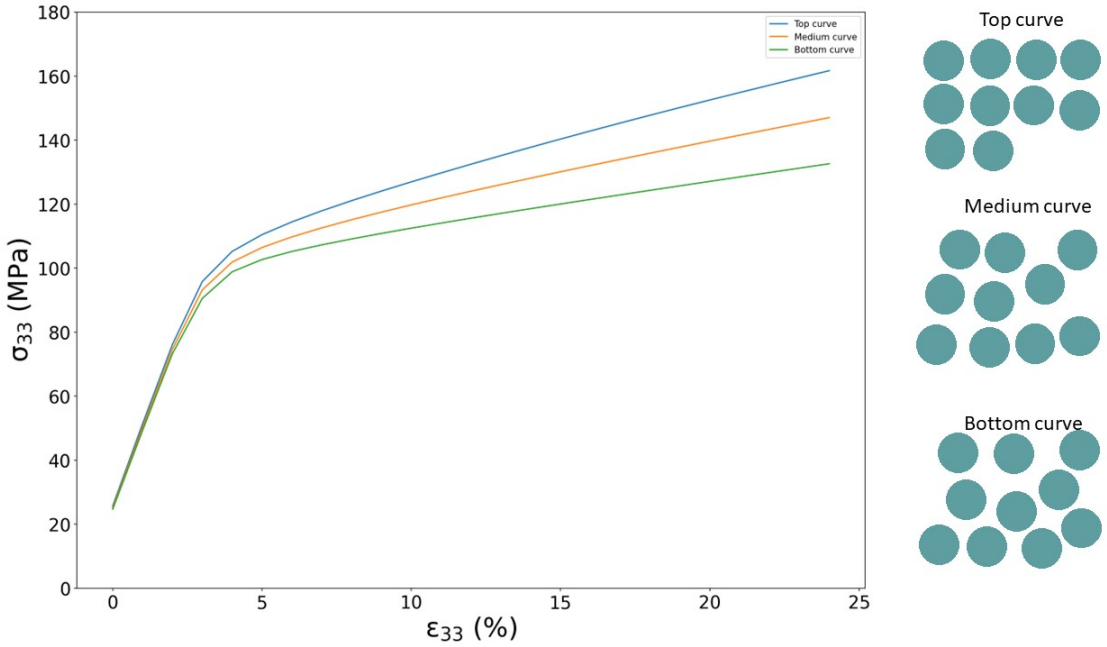



*Figure 3.5. Homogenized stress-stress curve for uniaxial loading by LOP = 3 and microstructures that produce stiffest, intermediate, and most compliant responses.*

The corresponding effect of RUC microstructure on the homogenized stress-strain response for uniaxial loading by transverse $\bar{\sigma}_{33}$ stress only oriented along the vertical axis is illustrated in Fig. 3.5. Similar to the response by uniaxial transverse stress $\bar{\sigma}_{22}$ shown in Fig. 3.4, the stiffest response is produced by the ordered microstructure RUC and the softest by the most disordered one. In both cases the ordered rows of fibers see the same uniaxial stress applied horizontally in the case of LOP = 2 and vertically in the case of LOP = 3, as do the fibers in the disordered RUCs. The stress transfer from the softer matrix to the stiffer fibers occurs through the same mechanism involving local transverse shear stress, with the aligned fibers being more effective in carrying the load.

In contrast, under uniaxial transverse shear loading only, it is the most ordered microstructure that produces most compliant response, with the stiffest response generated by the most disordered RUC, Fig. 3.6. The effect of microstructure on the homogenized response is also much less pronounced relative to the two preceding cases involving transverse normal stresses $\bar{\sigma}_{22}$ and $\bar{\sigma}_{33}$. The fibers in the disordered arrays are more effective in carrying the applied transverse shear stress as the mean distance between them, and hence the softer matrix content exposed to the load, is smaller than in the ordered RUC with a large matrix area carrying smaller load.

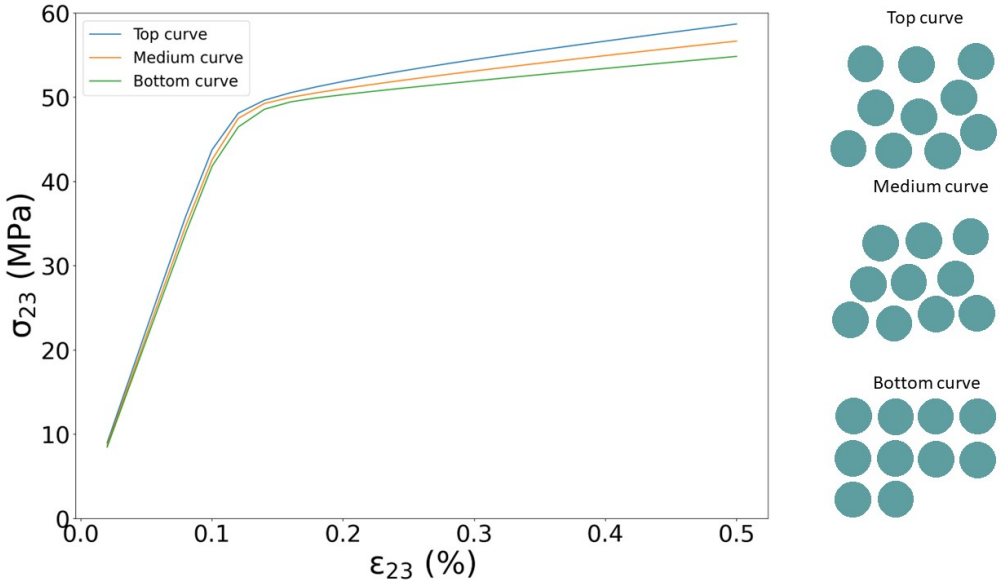



*Figure 3.6. Homogenized stress-stress curve for uniaxial loading by LOP = 4 and microstructures that produce stiffest, intermediate, and most compliant responses.*

Figures 3.7 and 3.8 illustrate the effect of RUC microstructures on the homogenized stress-strain responses by uniaxial shear loading by $\bar{\sigma}_{13}$ and $\bar{\sigma}_{12}$ in the $x_1$-$x_3$ and $x_1$-$x_2$ planes, respectively. Despite differences in the RUC microstructures, the extent of variation in the homogenized stress-strain response is quite small, creating a challenge for the ML algorithm construction.

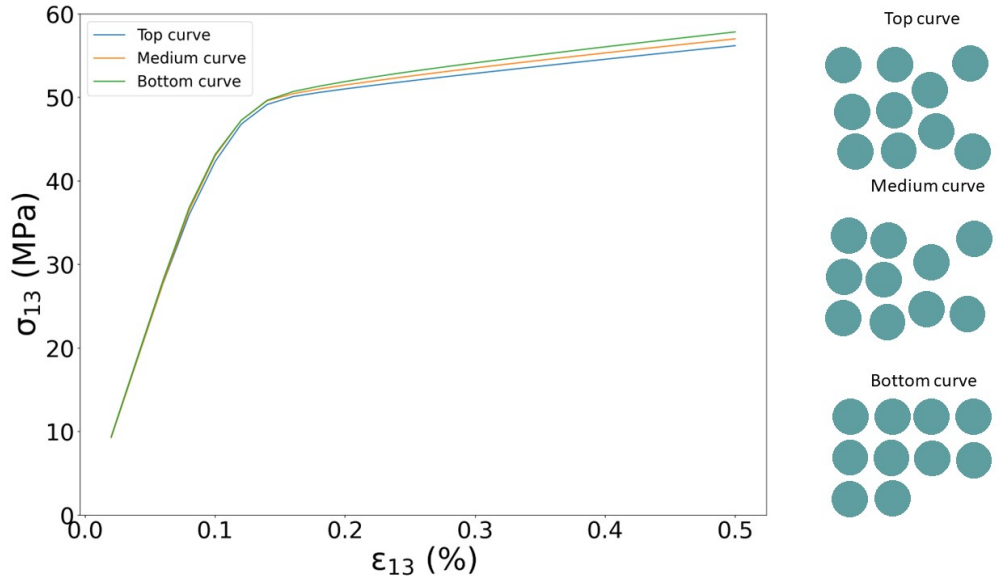



*Figure 3.7. Homogenized stress-stress curve for uniaxial loading by LOP = 5 and microstructures that produce stiffest, intermediate, and most compliant responses.*

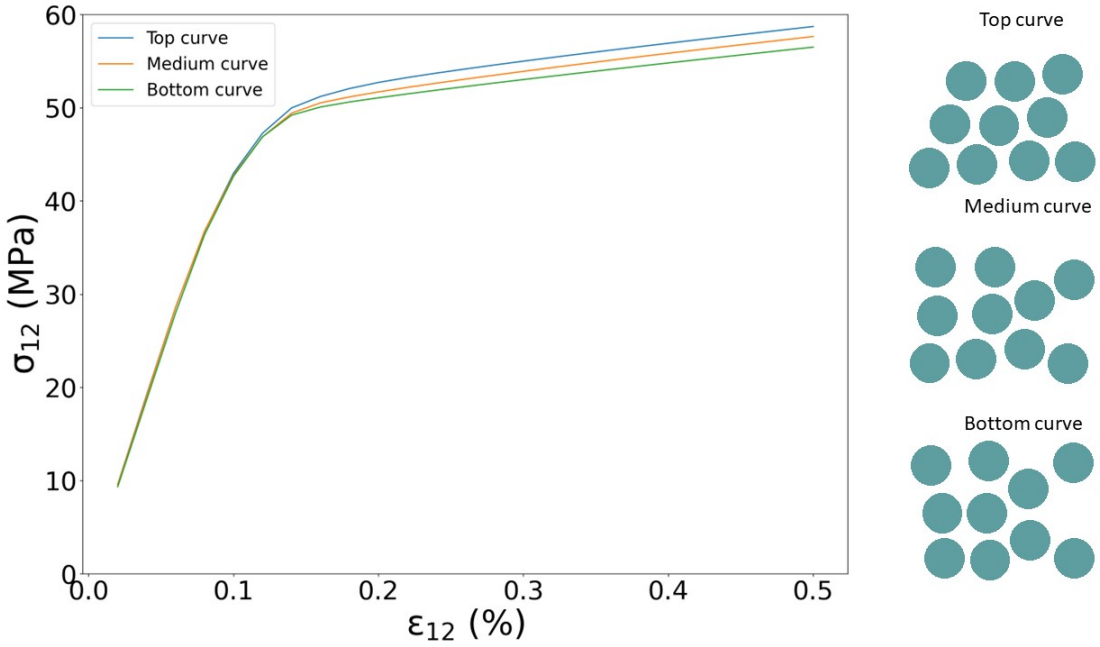



*Figure 3.8. Homogenized stress-stress curve for uniaxial loading by LOP = 6 and microstructures that produce stiffest, intermediate, and most compliant responses.*

### 3.2.7 Homogenized stress-strain response comparison

In order to further investigate the effect of fiber distribution on the homogenized stress-strain behavior under different uniaxial loadings, we compare the stiffest, most compliant and intermediate stress-strain responses under uniaxial transverse loading by $\bar{\sigma}_{22}$ with those obtained under uniaxial transverse loading by $\bar{\sigma}_{33}$ generated using the same microstructures in Fig. 3.9. Similarly, we make the same comparison under axial shear loading by $\bar{\sigma}_{12}$ and $\bar{\sigma}_{13}$ in Fig. 3.10. The reference unit cell microstructures are those that produce the stiffest, most compliant and intermediate responses under LOP 2 and LOP 6. This comparison reveals the extent to which the unit cells response approaches that of a homogenized transversely isotropic material. Specifically, for loading by uniaxial transverse normal stresses $\bar{\sigma}_{22}$ and $\bar{\sigma}_{33}$, Fig. 3.9, the homogenized responses for the three different microstructures that produce stiffest, intermediate and softest responses, are practically identically. Similar behavior is observed under axial shear loading by $\bar{\sigma}_{12}$ and $\bar{\sigma}_{13}$, Fig. 3.10.

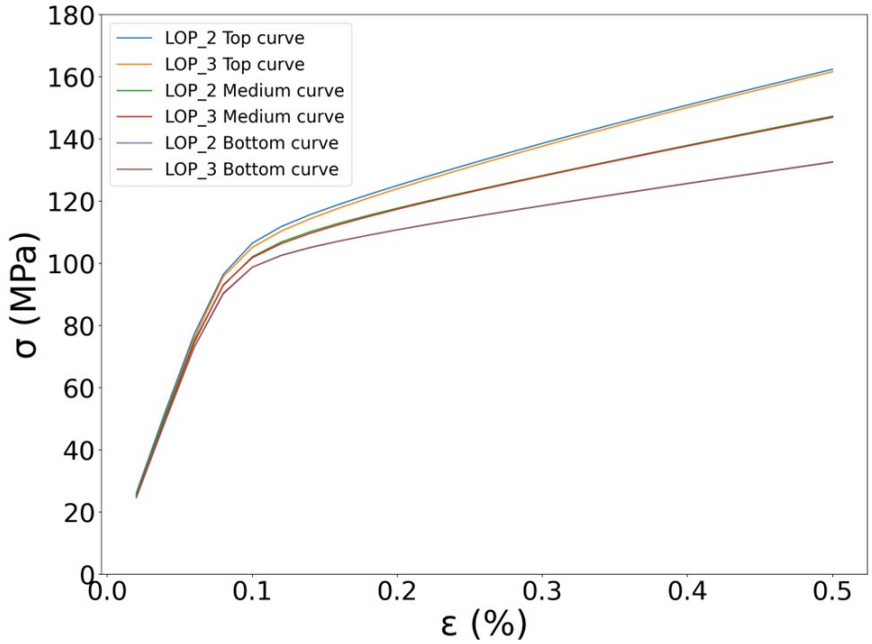



*Figure 3.9. Comparison of homogenized stress-stress curves for uniaxial loading by LOP = 2 and 3 for microstructures that produce stiffest, intermediate and most compliant responses under LOP = 2 loading.*

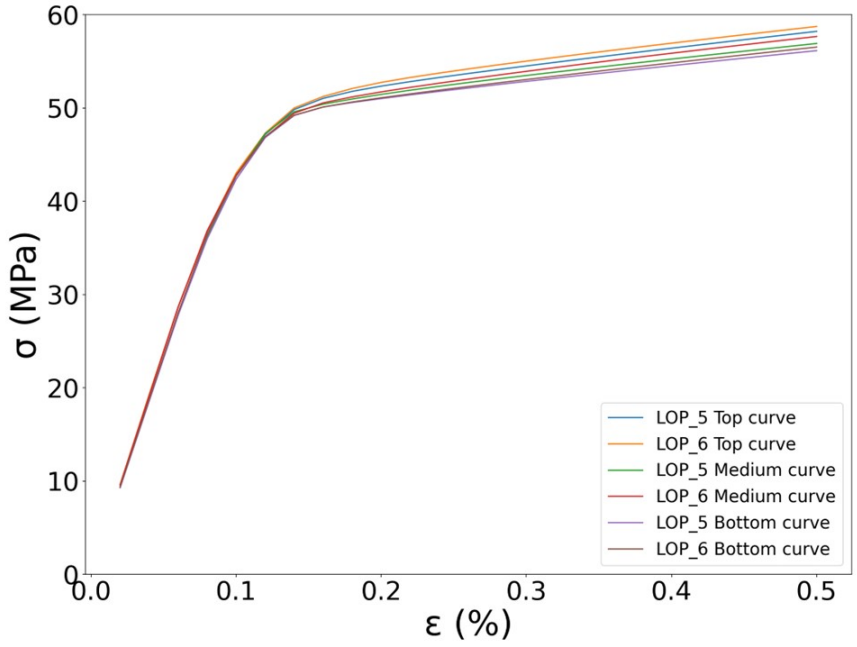



*Figure 3.10. Comparison of homogenized stress-stress curves for uniaxial loading by LOP = 5 and 6 for microstructures that produce stiffest, intermediate and most compliant responses under LOP = 2 loading.*

## 3.3 Summary of Results

The results illustrated in this chapter, which were produced using the automated FVDAM-based computational homogenization tool, will be employed for training and predictive purposes using two different machine learning algorithms. The algorithms will be described in Chapter 4, including the training and accuracy, and the predictions will be presented in Chapter 5.

Nonetheless, the results described in this chapter are important in their own right in understanding the effect of microstructural features on the homogenized response of unidirectional metal matrix composites in both elastic and elastic-plastic regions. Whereas the fiber distribution affects the homogenized elastic moduli to some extent under transverse normal and shear loading, as well as axial shear loading, its effect in the elastic-plastic region is substantially greater. The homogenized elastic moduli are also affected by the mismatch in the fiber/matrix elastic moduli which for the present b/al composite is relatively low, producing relatively small differences in the homogenized moduli of random microstructural realizations of the unit cell. In contrast, substantially greater microstructure-dependent deviations or scatter is observed in the elastic-plastic domain which also depends on the direction of the applied load relative to the fiber direction. The homogenized stress-strain response due to loading in the fiber direction is independent of the RUC microstructure randomness due to the fiber constraint. Hence the stress-strain curved are virtually identically for the 2,000 RUC microstructures generated. Under transverse normal stress loading by $\bar{\sigma}_{22}$ and $\bar{\sigma}_{33}$, however, the extent of scatter is quite large. Somewhat smaller scatter is observed under transverse shear loading, with the stiffest and softest responses generated by microstructures which produce softest and stiffest responses under transverse normal loading. Still smaller scatter is observed under axial shear loading in the two orthogonal axial planes containing the fiber direction.

The observed microstructure-dependent and load-dependent homogenized stress-strain behavior of the investigated unidirectional metal matrix composite presents challenges in developing accurate and efficient ML-based algorithms.

# Chapter 4

# ML Algorithms and Architectures

In this chapter we discuss the architectures and the flow of information that they control of two ML algorithms employed to predict the homogenized elastic moduli and stress strain response of random unidirectional composites, namely the artificial neural networks (ANN) and convolutional neural network (CNN). The individual sections first describe the architectures and related information flow of the two types of networks in general terms. Subsequently, the general discussion with the defined terminology is followed by specific description of the two architectures designed for the explicit purpose of predicting the response of random composites whose homogenized elastic moduli and stress-strain curves were generated using the FVDAM-driven computational tool described in Chapter 2. Because the main focus of the thesis is on the prediction of elastic-plastic response of random unidirectional composites, each section first describes the respective networks developed for this purpose followed by corresponding networks for the homogenized elastic moduli.

## 4.1 ANN Architecture

ANNs are composed of a certain number of layers of nodes. The data flow starts from the input layer, it is then processed within the hidden layers and then displayed in its final form by the output layer, Fig.4.1. Artificial neural networks are the standard deep learning algorithms for machine learning. They were originally based on the inner workings of the human brain, in which over 20 billion neurons propagate signals to each other called neurotransmitters to communicate all kinds of information. The more often a connection is used (learning a new skill), the stronger it becomes, and the less often a connection is used, the weaker it becomes. These principles are embodied in the structure of a feed-forward neural network. Individual nodes are organized as 1-dimensional layers, with each layer densely connected to the one in front of it such that each node

is connected via a weight to every node in the next layer. The weights symbolize the influence the former node has on the value of the latter node.

The overall ANN architecture is illustrated in Fig. 4.1. The indices $i$ and $o$ represent the input and output layers, respectively. The first hidden layer, second hidden layer and the third hidden layer are labelled $h_1, h_2$ and $h_3$. The input 1, input 2, to input $m$ make up the input vector. The output 1, output 2, to output $n$ make up the output vector. Each small circle embodies a single neuron where the data information is processed. The solid lines connecting the nodes in the 5 layers represent the weights, which are the model parameters we want to get after training the model. The first layer of nodes is comprised of the input values, and the last layer represents the output values. In between lie the inner layers, each of which comprises an arbitrary number of nodes. The size of these layers depends on the fundamentals of the problem, the values of $M$ and $N$, and pure intuition. Each non-input node has a value equal to the sum of each node from the previous layer multiplied by their weight. In other terms, it's equivalent to multiplying a 1-dimensional vector of nodes by a 2-dimensional matrix of weights to produce a 1-dimensional vector. Each non-input layer is fed into an activation function, which typically acts per element, after the summations to help organize the data (such as converting negative values to 0). In some models, each layer contains a bias node that isn't affected by the input nodes yet is involved in the summations.

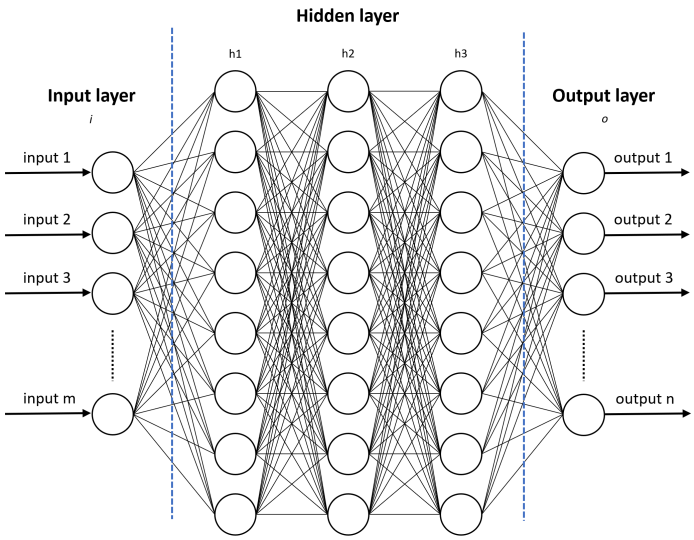



*Figure 4.1: General structure of artificial neural network*

The vector **X**, which consists of the input elements $[x_1, x_2, x_3, \dots, x_m]$ represents the input, and it contains the feature that we give to our model to output a prediction contained in the vector **Y** comprised of output elements $[y_1, y_2, y_3, \dots, y_m]$. Weights control the signal (or the strength of the connection) between two neurons. In other words, a weight decides how much influence the input will have on the output. Biases, which are constant, are an additional input into the next layer that will always have the value of 1. Bias units are not influenced by the previous layer (they do not have any incoming connections) but they do have outgoing connections with their own weights. The bias unit guarantees that even when all the inputs are zero there will still be an activation in the neuron. After enough iterations of forward and backward propagations discussed in the sequel, the neural network minimizes the error between the actual outputs and the predictions to give accurate and robust predictions.

Neural networks are best utilized for problems of the type of $R^M \rightarrow R^N$ where $M$ and/or $N$ are large values and the connection between inputs and outputs is nonlinear and/or non-obvious. If those conditions are not true, then using a neural network may be an overkill solution to the problem. Neural networks function well for both regression and classification problems but must be designed differently for each type. Both types have either discrete or continuous numerical values as inputs, where $M$ is the number of values. For regression, the outputs are just continuous values, where $N$ is the number of values. For classification however, $N$ is the number of possible labels, and typically the output values are between 0 and 1 representing the probability that the label exists. If only one label is allowed, then the largest output value signals the correct label.

## 4.2 Control of information flow in ANN

### 4.2.1 Neurons and information flow within the neural network

Figures 4.2 and 4.3 illustrate how the data is processed within the neurons in each hidden layer. In Figure 4.2, all the input elements $[x_1, x_2, x_3, \dots, x_m]$ in **X** are passed to the nodes in the layer of the neural network. The symbol $w$ represents the weight parameter in each node, the summation $\sum$ represents the transfer function and $\boldsymbol{\varphi}$ represents activation function. The transfer function is given by $\boldsymbol{z} = b + \sum_{i=1}^{N} a_i w_i$. When the net input is calculated from the transfer function, it is fed to the activation and compared to the output from the activation function with the threshold

$\theta$. If the output from the activation surpasses the threshold, the output will be passed to the nodes of the subsequent layer and this process is repeated for each node in the next layer.

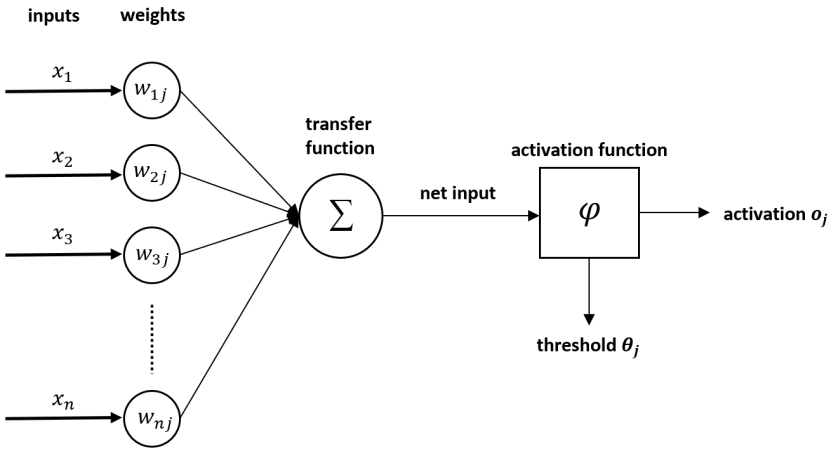



*Figure 4.2. Data flow within each layer*

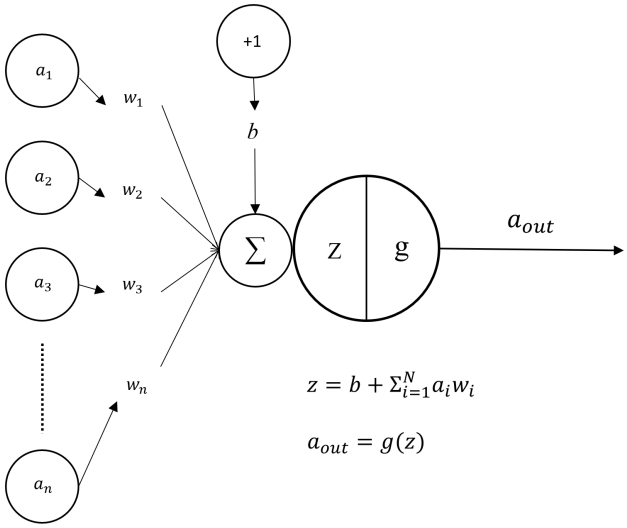



*Figure 4.3. Parameters within one single neuron*

Figure 4.3 transfers the input data $[a_1, a_2, a_3, \ldots, a_n]$ into the output data $\boldsymbol{z}$. The output data $\boldsymbol{z}$ is then passed to the activation function $\boldsymbol{g}$ and the activation function defines how the weighted sum of the input is transformed into an output within this node and then pass this output $g(z)$ to the nodes of the next layer. After the transformation and activation through all the hidden layers, the output $a_{out}$ is obtained from the final output layer.

### 4.2.2 Data Normalization

Before the input data is fed to the model, it is normalized. Data normalization is the process of transforming the input and/or output data to achieve better results. These adjustments serve to better accent the distinctions between values, like using linearization to identify a linear relationship. Often, data normalization involves adjusting the range of data to better suit the desired activation function described in the sequel, which is typically in the range $\{x \in R | 0 < x < 1\}$. Even if all other choices for the model are optimal, without effective data normalization, the model may fail to achieve acceptable results.

Min-max adjustments (linear scaling) involve translating and scaling a set of data as follows:

$$x' = \frac{x - min}{max - min} \tag{4.1}$$

where x is an element of the data, min and max are the minimum and maximum values, respectively, of the data set, and x' is the transformed element. Theoretically, a min-max adjustment on an entire data set together should make no difference, yet in practice it results in less computation time to achieve the same results, Ref [39]. The range $\{x \in R | 0 < x < 1\}$ works very well for neural networks and fits most activation functions, thus it is good practice to transform all data to that range. If different components of the input data have vastly different scaling from each other, applying a min-max adjustment per component across all input data can help avoid a bias towards components that are consistently larger than others.

Standardization is the conversion of a set of data into z-scores as follows:

$$z = (x - \mu)/\sigma \tag{4.2}$$

where x is an element of the data, μ is the mean of the data set, σ is the standard deviation of the data set, and z is the z-score of the element. This transformation is technically a form of linear scaling, but these z-scores can be the input for a normalization function such as the standard bell curve *CDF* or the sigmoid function:

$$x' = normcdf(z, \mu, \sigma), x' = \frac{1}{1 + e^{-x}} \tag{4.3}$$

Both options add emphasis to small differences near the mean and remove emphasis from small differences away from the mean. They also place the transformed data into the range $\{x \in R | 0 < x < 1\}$, which typically leads to better results. For data with a curved *CDF*, this transformation vastly improves results. Do note that if using one of these functions on output data, the inverse function is required to analyze the predicted outputs from the model, thus the restrictive range leads to a restrictive domain of the predicted outputs. In this case, an activation function such as sigmoid with the correct range must be used.

### 4.2.3 Loss Function

A loss function is calculated in the output layer to quantify how close the predicted values are to the target values with the difference called loss, which then determines the magnitude of the weight adjustments during the back propagation stage described in the sequel. Different choices emphasize different characteristics of the data. For regression, the main options are Mean Square Error (*MSE*), Mean Absolute Error (*MAE*), and Mean Absolute Percentage Error (*MAPE*):

$$MSE = mean((y_{pred} - y_{actual})^2) \tag{4.4}$$

$$MAE = mean(abs(y_{pred} - y_{actual})) \tag{4.5}$$

$$MAPE = 100 \times mean(abs((y_{pred} - y_{actual})/y_{actual})) \tag{4.6}$$

Mean Square Error is the general standard. Note that for the range $\{x \in R | 0 < x < 1\}$, calculating percentage error for very small $y_{actual}$ can result in a massive loss. For classification, the choice depends on the activation function used. For a *SoftMax* (one label only), use cross entropy, and for a *sigmoid* (multiple label possibilities), use *binary cross entropy*, Ref [40].

### 4.2.4 Activation Function

An activation function is enacted upon a layer once the values of the nodes values via summation have been computed. Backpropagation calculations require an activation function with a derivative, so a function with a constant sloe such as $y = x$ does not suffice. The standard choice is the Rectified Linear Unit (ReLU):

$$x' = max(x, 0) \tag{4.7}$$

where $x$ is a single node's value, and $x'$ is the new value for future calculations. In other words, *ReLU* converts all non-positive values to 0. Many flavors of *ReLU* exist where the tail end is curved, but the gains are small if any, and just lead to more computation. *ReLU* is widely used due to its quick computation time.

For more normalized values, the sigmoid function does well:

$$x' = \frac{1}{1 + e^{-x}} \tag{4.8}$$

where $x$ is a single node's value, and $x'$ is the new value for future calculations. For this function, the range of $x'$ is $\{x \in R | 0 < x < 1\}$, thus this activation function is essential for the output layer if the data has been normalized to that range. As an alternative, the hyperbolic tangent function works similarly, with a range of $\{x \in R | 0 < x < 1\}$ instead.

For classification problems with a single label, the SoftMax function is a great choice. It squeezes values in the range $\{x \in R | 0 < x < 1\}$, while also ensuring that the layer has a magnitude of 1, thus isolating the most prominent prediction, Ref [41].

### 4.2.5 Optimization Function

Neural networks are trained with the stochastic gradient descent algorithm. Stochastic gradient descent is an optimization function that calculates the error (loss) gradient by using the training data set and updates the neural network weights with backpropagation algorithm. There are many optimization functions employed in neural networks such as Gradient Descent, Stochastic Gradient Descent, Stochastic Gradient descent with momentum, Mini-Batch Gradient Descent, Adagrad, RMSProp, AdaDelta and Adam.

The amount that the weights are updated during training is referred to as learning rate. Learning rate determines how quickly the model is trained measured in terms of epochs. We note that one epoch means training the entire neural network once or one cycle. For a small learning rate, the model requires more training epochs because the changes in weights are small in each update. In contrast, a large learning rate produces rapid changes in the weight parameters and thus requires fewer training epochs. If the learning rate is too small, the training process may take forever.

However, if the learning rate too large, the model might converge too fast to a suboptimal position. Thus, learning rate is a very important hyperparameter in the neural network training. The choice of the leaning rate is related to the performance of the model.

In this study, we employ the Adam optimizers for the following reasons. Adam optimizer is an extension of the stochastic gradient descent that updates the weight in the network during the training. Adam optimizer updates the learning rate for each network weight individually and due to this flexible characteristic, it usually improves model performance and therefore is recommended as default optimization function during training. Moreover, the other advantage of the Adam is that it takes less time to train the model, and requires less computer memory and tuning than other optimization functions.

## 4.3 Backward propagation

The key to neural network learning is a process called back-propagation. A loss function is calculated in the output layer comparing it to the expected output values, returning a single positive value known as loss. The larger the loss, the worse the prediction. The loss is "propagated" backwards through the model, where weights are adjusted via partial derivative calculations, with a larger loss leading to larger adjustments, Ref [8]. As the model trains, the adjustments can be smaller and smaller, even as the loss stabilizes. An optimizer function adjusts the constants in the computations based on a specific learning rate.

If the input $x \in R^d$ without a bias term, we can calculate intermediate variables as follows:

$$z = W^{(1)}x \tag{4.9}$$

where as $W^{(1)} \in R^{h \times d}$ is the weight parameter of the hidden layer. We further achieve the hidden activation vector of length $h$ by applying the activation function $\phi$ to intermediate variables $z \in R^h$,

$$h = \phi(z) \tag{4.10}$$

The output $h$ of the hidden layer is another case of an intermediate variable. Under the assumption that only a weight of $W^{(2)} \in R^{q \times h}$ is retained in the output layer, an output layer variable of length $q$ is denoted as:

$$o = W^{(2)}h \tag{4.11}$$

If the loss function is $l$ and the example label is $y$, we can then write the loss term for individual data example as:

$$L = l(o, y) \tag{4.12}$$

The $l_2$ regularization norm with the hyperparameter $\lambda$ is:

$$s = \frac{\lambda}{2}(\left\|W^{(1)}\right\|_F^2 + \left\|W^{(2)}\right\|_F^2) \tag{4.13}$$

Regularization parameter $\lambda$ is a constant number, which is also called "penalty". Regularization means adding this penalty to the loss function. There are two types of regularization $l_1 = \lambda \sum_{j=1}^{k} |\theta_j|$ and $l_2 = \lambda \sum_{j=1}^{k} \theta_j^2$. Theta is the vector which contains the parameters of the model.

$F$ indicates the Frobenius norm of the matrix, which flattens the $l_2$ norm matrix into a vector. The regularized loss of individual data example is:

$$J = L + s \tag{4.14}$$

From now on, we depict the objective function as $J$.

Mathematically, backpropagation involves the calculation of the neural network parameters' gradients. The backpropagation calculates the gradients from the output layer to the input layer based on the chain rule. The intermediate variables, namely the partial derivatives of the neural network parameters, are stored by the backpropagation algorithm during the calculation. For example, we have $Y = f(X)$ and $Z = g(Y)$, where the input *X, Y* and output *Z* are tensors of any shape. With the application of chain rule, we can calculate the derivative of *Z* with respect to *X* by

$$\frac{\partial Z}{\partial X} = prod\left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X}\right) \tag{4.15}$$

where the "**prod**" operator denotes the multiplication of the arguments after the required operations (transposition, swapping input positions, etc.). This is straightforward for the vectors, which involve just matrix-matrix multiplication. For higher dimension tensors, there are other appropriate counterparts. Operator "**prod**" hides all the notations overhead in Eqn. (4.15).

From Eqns. (4.9) and (4.11), we have the parameters of the neural network $W^{(1)}$ and $W^{(2)}$. To calculate the parameters of gradients $\frac{\partial J}{\partial W^{(1)}}$ and $\frac{\partial J}{\partial W^{(2)}}$, we use backpropagation based on the chain rule. We reverse the order of calculation according to forward propagation to estimate

parameters towards the outcome of the graphic model. First, we calculate the gradients of the objective function including the loss term and the regularization term.

$$\frac{\partial J}{\partial L} = 1 \; and \; \frac{\partial J}{\partial s} = 1 \tag{4.16}$$

Next, we use the chain rule to update the gradient of the objective function variables in the output layer:

$$\frac{\partial J}{\partial o} = prod\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial o}\right) = \frac{\partial L}{\partial o} \in R^q \tag{4.17}$$

Meanwhile, we update the gradients of the regularization term:

$$\frac{\partial s}{\partial W^{(1)}} = \lambda W^{(1)} \; and \; \frac{\partial s}{\partial W^{(2)}} = \lambda W^{(2)} \tag{4.18}$$

The chain rule given by Eqn. (4.19) below returns the gradient $\frac{\partial J}{\partial W^{(2)}} \in R^{q \times h}$ of the last layer before the output layer:

$$\frac{\partial J}{\partial W^{(2)}} = prod\left(\frac{\partial J}{\partial o}, \frac{\partial o}{\partial W^{(2)}}\right) + prod\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(2)}}\right) = \frac{\partial J}{\partial o} h^T + \lambda W^{(2)} \tag{4.19}$$

We use Eqn. (4.20) to calculate the gradients involved in the hidden layer output $\frac{\partial J}{\partial h} \in R^q$, to acquire the gradient in $W^{(1)}$:

$$\frac{\partial J}{\partial h} = prod\left(\frac{\partial J}{\partial o}, \frac{\partial o}{\partial h}\right) = {W^{(2)}}^T \frac{\partial J}{\partial o} \tag{4.20}$$

We multiply the activation function $\phi$ by the gradient $\frac{\partial J}{\partial z} \in R^h$ of the intermediate variable $z$ using the elementwise multiplication operator, which is denoted by $\odot$:

$$\frac{\partial J}{\partial z} = prod\left(\frac{\partial J}{\partial h}, \frac{\partial h}{\partial z}\right) = \frac{\partial J}{\partial h} \odot \phi'(z) \tag{4.21}$$

Lastly, the gradient $\frac{\partial J}{\partial W^{(1)}} \in R^{h \times d}$ of the parameters in the first layer of the model is achieved by the chain rule:

$$\frac{\partial J}{\partial W^{(1)}} = prod\left(\frac{\partial J}{\partial z}, \frac{\partial z}{\partial W^{(1)}}\right) + prod\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(1)}}\right) = \frac{\partial J}{\partial z} x^T + \lambda W^{(1)} \tag{4.22}$$

Since the gradient of the parameters $\frac{\partial J}{\partial W^{(1)}}$ of the first layer is obtained, the neural network knows how to tune the parameters to make the "Loss" move towards the low-lying place to minimize it. The deep neural network repeats this process again and again with an appropriate selection of learning rate until the loss decreases to the minimum value and remains at a constant level. Then the model's training is finished, and we can start to check the model performance.

## 4.4 ANN structure for the stress-strain prediction

The ANN model designed in this study to predict the homogenized stress-strain curves has 12 neural layers comprised of 1 input layer with 256 nodes, 10 hidden layers with each containing 256 nodes, and 1 output layer with 25 output nodes. The loss function adopted for this ANN model is the mean squared error (MSE) denoting the averaged squared difference between the actual values and the predicted values. The activation function applied in this model is "RELU" since the RELU is a linear transformation function employed especially for regression problems and it outperforms all the other activations for regression models. The optimizer used here is "Adam", not only because Adam applies stochastic gradient descent for deep learning models which ensures robustness during training, but also can adapt its learning rate for sparse data.

The detailed structure of the employed ANN model is shown in Fig. 4.4. The input vector comprised of [input 1, input 2…, input 22] in the input layer represents the information of each RUC. Input 1 and input 2 are 159 and 211, namely the height and width of the RUC images. Input 3 to input 22 represent the 10 (x, y) coordinates of the 10 fiber centers. Similarly, [output1, output2…, output25] are the predictions of the equally spaced 25 data points that represent homogenized stress-strain curves shown in Chapter 3.

Whereas the homogenized stiffness matrix $\boldsymbol{C}^*$ is calculated just once for each microstructural realization, the homogenized elastic-plastic response depends on the applied loading path. These homogenized responses have been generated under uniaxial stress loading defined by the six load options LOPs described in Chapter 3 that characterize the composite response. These loading

options produce uniaxial normal loading along and transverse to the fiber direction, and axial and transverse shear loading in three planes, two of which contain the fiber axis and one transverse to it. Uniaxial stress loading is obtained by adjusting the homogenized strain components accordingly to produce the desired single non-zero homogenized stress component.

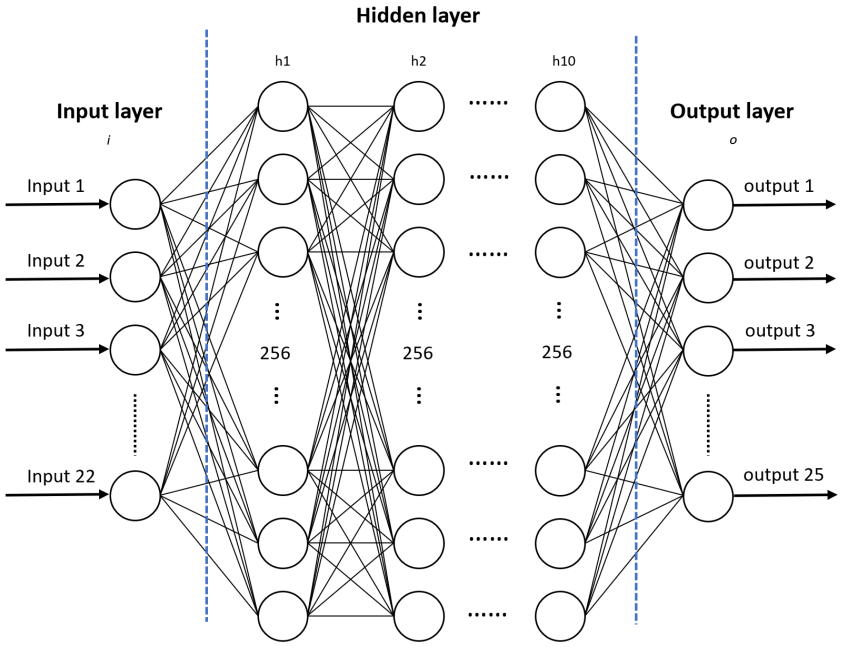



*Figure 4.4. Artificial Neural Network (ANN) structure for the stress strain prediction*

## 4.5 ANN structure for the homogenized moduli prediction

The ANN model designed in this study to predict the 13 homogenized moduli has 12 neural layers comprised of 1 input layer with 256 nodes, 10 hidden layers with each containing 256 layers, and 1 output layer with 13 output nodes. The loss function adopted for this ANN model is the mean squared error (MSE), which represents the averaged squared difference between the actual values and the predicted values. The activation function applied in this model is "RELU" since the RELU is a linear transformation function employed especially for regression problems and it outperforms all the other activations for regression models. The optimizer used here is "Adam", not only because Adam applies stochastic gradient descent for deep learning models which ensure the robustness for the training, but also can adapt its learning rate for sparse data. The detailed structure for the applied ANN model is shown in Fig 4.5.

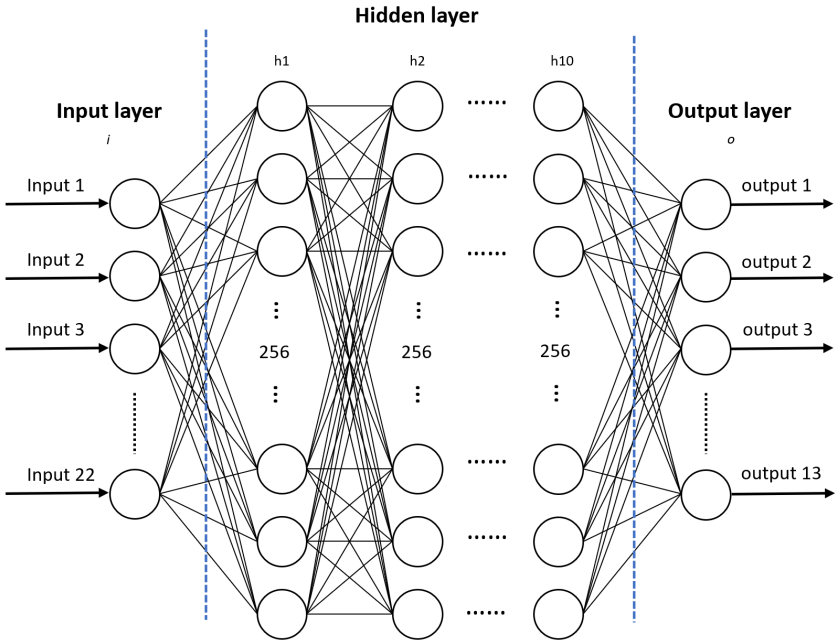



*Figure 4.5. Artificial Neural Network (ANN) structure for the homogenized moduli prediction*

In Fig. 4.5, same as in Fig. 4.4, [input 1, input 2…, input 22] in the input layers represent the input information of each RUC. Input 1 and input 2 are 159 and 211, namely the height and width of the RUC images. Input 3 to input 22 represent the 10 (x, y) coordinates of the 10 fiber centers. Similarly, [output 1, output 2…, output 13] are the predictions of the 13 homogenized moduli calculated just once for a given microstructural realization.

## 4.6 Convolutional Neural Network (CNN)

When dealing with image data, such as image recognition or image classification, the convolutional neural network is needed to reduce the computation amount. For example, just suppose we have 500×500-pixel RGB image, the total amount of the pixels is 500×500×3=750000. If we design the ANN with 1000 neurons in the first layer, then the total number of weights in the first layer of the ANN will be 750000×1000=750000000. The computational consumption for the number of weights is far beyond the PC power and therefore results in huge time consumption for the training. To resolve this problem, we can adopt the convolutional neural network and utilize this advantage in reducing the dimensions of the data and thus lead to fewer calculations and computations.

Convolutional neural network is most widely used to analyze image data, like image recognition and image classification. The structure of the Convolutional Neural Network can be divided into 2 parts, convolutional components and fully connected layers used in ANNs. The numerous filters in each convolutional layer convolve across the one-dimensional image in Figure 4.6 and extract the features to generate feature maps. During this process, the network automatically learns to optimize the filters through the forward propagation and back propagation learning process, and this is a splendid advantage compared to these traditional algorithms where manual feature engineering is needed, including data set manipulation such as addition, deletion, combination, and mutation to improve machine learning model training in order to produce better performance and higher accuracy. Then the network applies max pooling layer (a pooling operation that selects the maximum element from the region of the feature map covered by the filter, the filter is a matrix that moves over the inputs and extract similarities between different locations in the input image) to reduce the feature map size and reduce sampled feature map to fewer parameters. Figure 4.6 illustrates the max pooling operation.

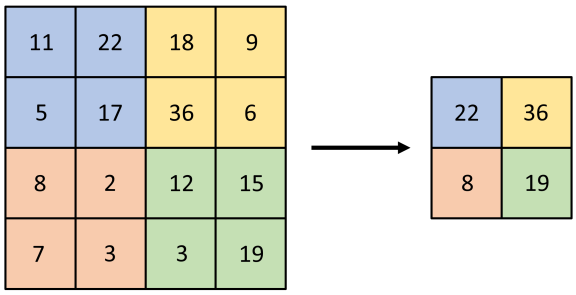



*Figure 4.6. Max pooling operation*

The max pooling layer also can help to extract low level features such as edges and points. After the convolutional stage, the CNN flattens the feature map that contains the features and information extracted by the filters and feeds it to the fully connected layers to generate the output. The flattening process is conducted by the flattening layer, which collapses the two-dimensional matrices of the input from the convolutional layer into the one dimension vector and then passes the data to the dense layers.

In this thesis, we fed the CNN model with the microstructure to generate the corresponding homogenized response, and within the back propagation process, the model compares the predictions and true values and calculate the loss. Loss of the final prediction decrease with the optimization of the filters and the weight within each neuron during the forward/backward propagation and gradient descend process. Then the prediction accuracy increases and lead to good performance of the CNN model.

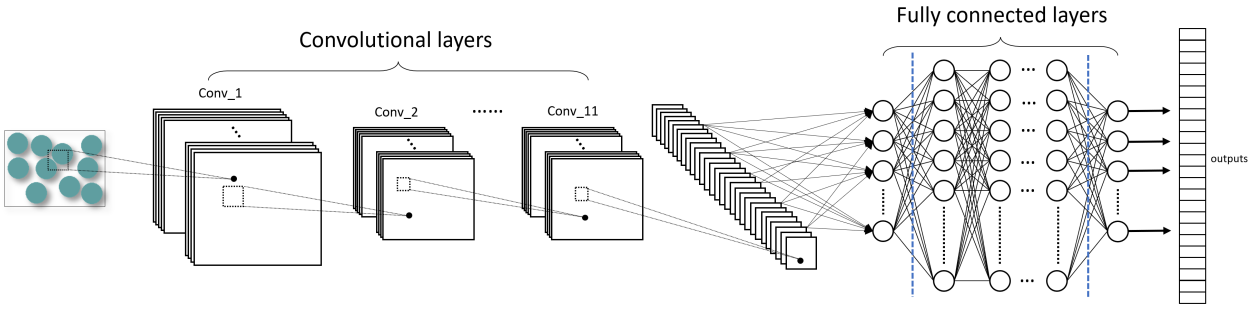



*Figure 4.7. Convolutional neural network structure*

## 4.7 CNN structure for the stress-strain prediction

The CNN model designed to predict the homogenized stress-strain curve has 12 neural layers comprised of 1 input layer with 256 nodes, 9 convolutional layers with "ELU" activation functions with each layer containing 256 layers across the depth, 10 hidden layers with "RELU" activation functions with each layer containing 256 nodes, and 1 output layer with 25 output nodes. The loss function adopted for this ANN model is the mean squared error (MSE) which is the averaged squared difference between the actual values and the predicted values. The optimizer used here is "Adam", not only because Adam applies stochastic gradient descent for deep learning models which ensures robustness during training, but also can adapt its learning rate for sparse data. The detailed structure for the applied CNN model is shown in Fig. 4.8.

The image on the extreme left side in Fig. 4.8 is the RUC image, the dashed line squares are the $2 \times 2$ filters that convolve across the image. The RUC image is the input data, and the 25 outputs are the predictions of the stress-strain points that make up the stress-strain curves.

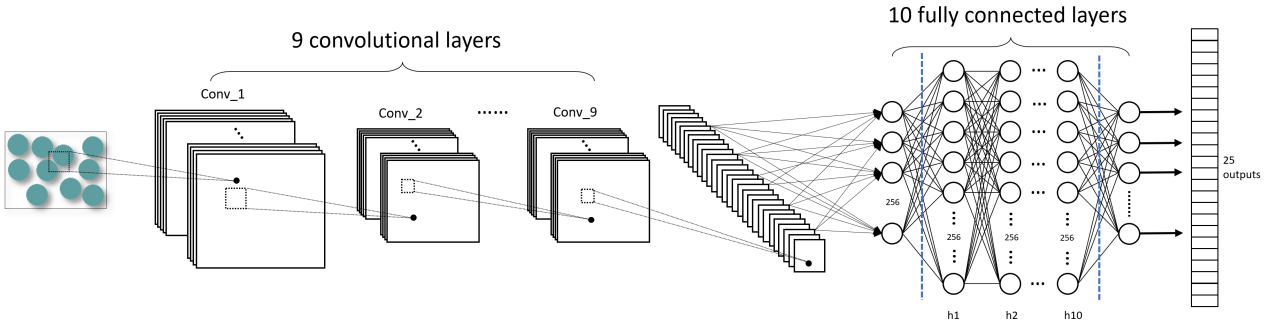



*Figure 4.8. Convolutional neural network (CNN) structure for the stress strain prediction*

## 4.8 CNN structure for the homogenized moduli

Similar to the CNN model designed to predict the homogenized stress-strain curves, the CNN model designed to predict the 13 homogenized moduli also employs the same architecture, the only difference is that the output layer of the CNN model for the homogenized moduli has 13 output nodes that represent the 13 homogenized moduli instead of the 25 nodes employed in the predictions of the homogenized stress-strain curves. The detailed structure is shown in Fig. 4.9.

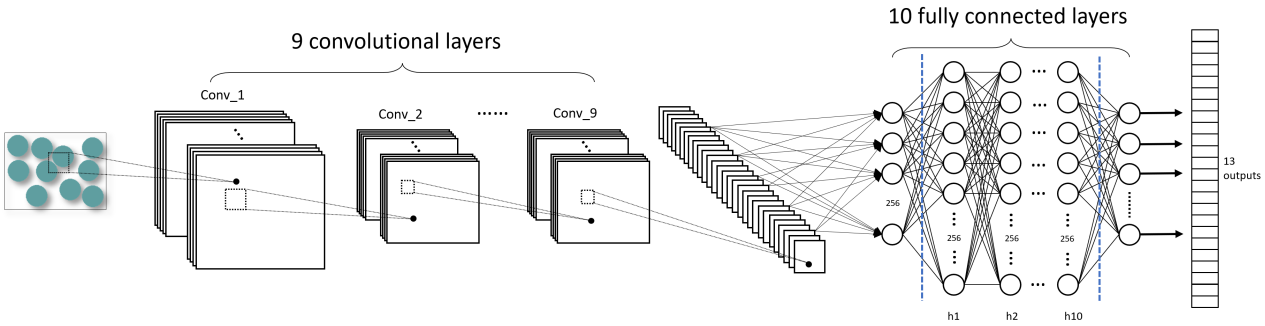



*Figure 4.9. Convolutional neural network (CNN) structure for the homogenized moduli prediction*

Similar to Fig. 4.8, the image on the left side is the RUC image, the dashed line squares are the 2 × 2 filters that convolve across the image. The RUC image is the input data, and the 13 outputs are the predictions of the 13 homogenized elements.

## 4.9 Summary

This chapter described the ANN and CNN models that were designed for the prediction of the homogenized stress-strain responses and corresponding elastic moduli of a unidirectional metal matrix composites with random fiber distributions based on the generated RUC microstructures containing 10 fibers. The ANN model developed to predict the stress-strain data employs the RUC fiber center coordinates as inputs and outputs the predicted stress-strain curves. Likewise, the ANN model developed to predict the homogenized moduli also employs the RUC fiber center coordinates as inputs and outputs the predicted 13 homogenized moduli elements. Both ANN model architectures are identical, as are the loss, activation, and optimization functions. The CNN model developed to predict the stress-strain curves employs the RUC images as the input data and generates the predicted stress-strain curves. Likewise, the CNN model that predicts the homogenized moduli also employs the RUC images as the input data and generates the predicted 13 homogenized moduli elements. The performance of all these 4 neural networks will be discussed in Chapter 5.

The ML algorithms described in this chapter are employed in Chapter 5 to predict both the stress-strain response and the 13 homogenized moduli of random microstructures of a unidirectional boron/aluminum composite. The specific ANN and CNN algorithms have been refined in an iterative process to obtain optimal predictions. In addition to illustrating the predictive capabilities of the two algorithms and their differences, the reduction in execution time is also discussed relative to full scale and computationally intensive homogenization based on the homogenization theory called FVDAM. This is the driving motivation for this study to enable multiscale analysis of large structures.

# Chapter 5

# Model Results

In this chapter we discuss the performance of the designed ANN and CNN models for predicting the stress-strain curves and homogenized moduli of a boron/aluminum unidirectional composite with random fiber distributions. The ANN and CNN models for the stress-strain responses are trained based on 6 data sets generated under applied loading by six uniaxial homogenized stresses (LOPs) discussed in Chapter 4, with each LOP data set size of 2,000 RUC microstructural realizations. The training set size is 1800, randomly selected from the 2,000 data samples. The test set is comprised of 100 randomly selected data. 50 data sets are used for validation and the remaining 50 sets are used to predict and compare the performance of the models relative to the actual FVDAM generated curves. In contrast, the ANN and CNN models for predicting the 13 homogenized moduli are trained based on 20,000 microstructural realizations. Similarly, the training set size is 18,000, randomly selected from the 20,000 data samples. 1,000 data sets are used for testing and 500 are used for validation. The remaining 500 data sets are used to predict and compare the performance of the models. The accuracy and the loss are plotted to check the model performance. The performance of the models is displayed graphically by plotting the predicted and actual stress-strain responses for the six LOP loading cases. As for the homogenized moduli, the actual and predicted homogenized moduli are plotted on scatter plots to determine how far the data points fall from the regression line $y = x$.

## 5.1 ANN model performance on the stress-strain data sets

Figure 5.1 illustrates the ANN model accuracy rate during the training epochs for the 6 LOPs loading cases. The evolving accuracy of the predictions for the 6 LOPs is similar. The accuracy rate calculated based on the training set increases rapidly with the number of epochs and attains a constant level around 100%. Similarly, the validation accuracy rate calculated based on the validation set also increases rapidly epoch count and attains a constant level around 100%.

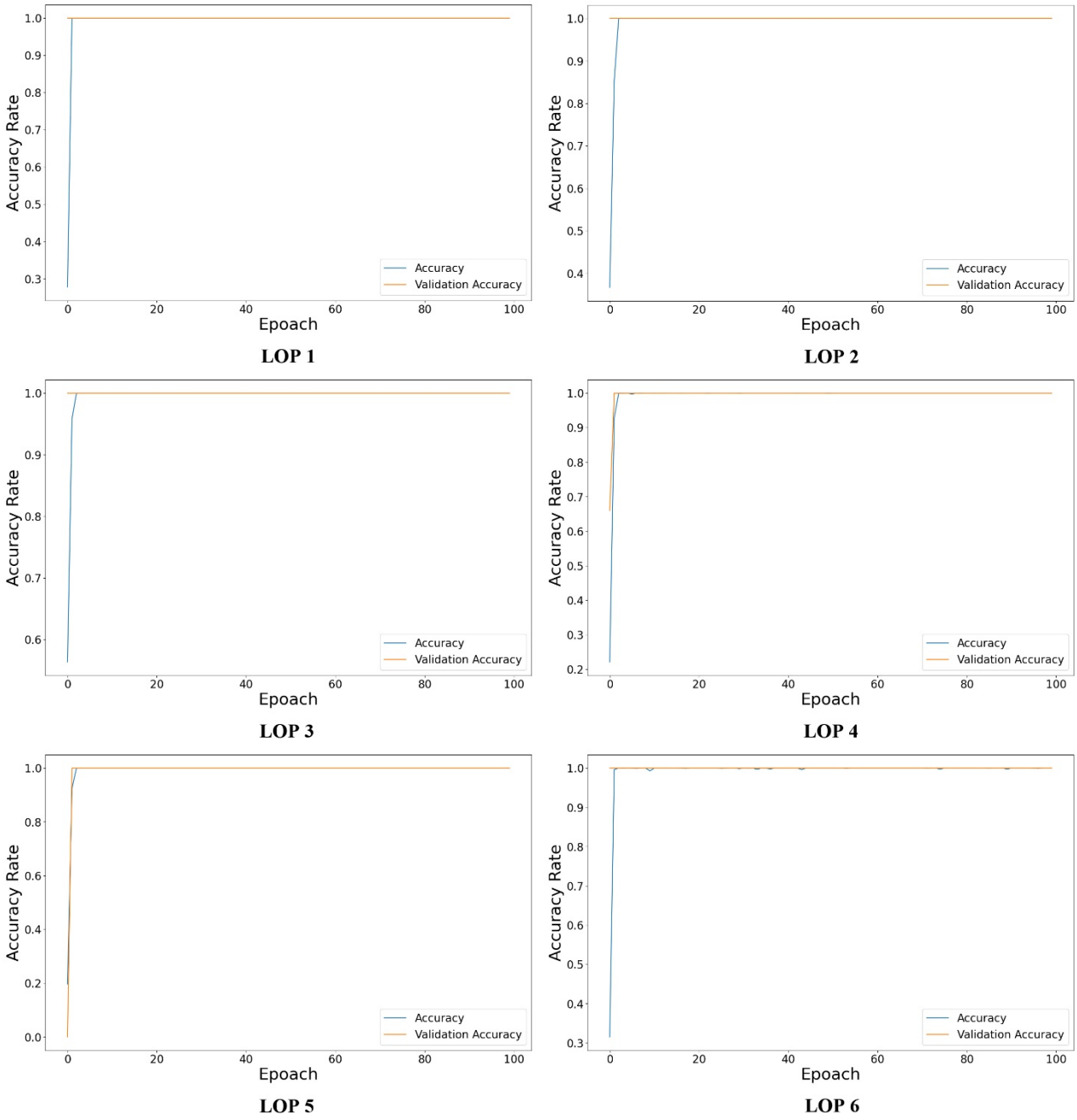



*Figure 5.1. ANN model for stress strain curve accuracy rate of the 6 LOPs*

Figure 5.2 illustrates the loss variation for the 6 LOPs. All the 6 Loss functions, which are calculated from the difference between the actual training response and the predictions, decrease rapidly during the first 2 epochs and finally remain constant somewhere at 0. The validation loss functions, which are calculated from the difference between the actual validation response and the predictions,

start from somewhere a little higher than 0, decrease in the first 2 epochs and finally remain constant at around 0.

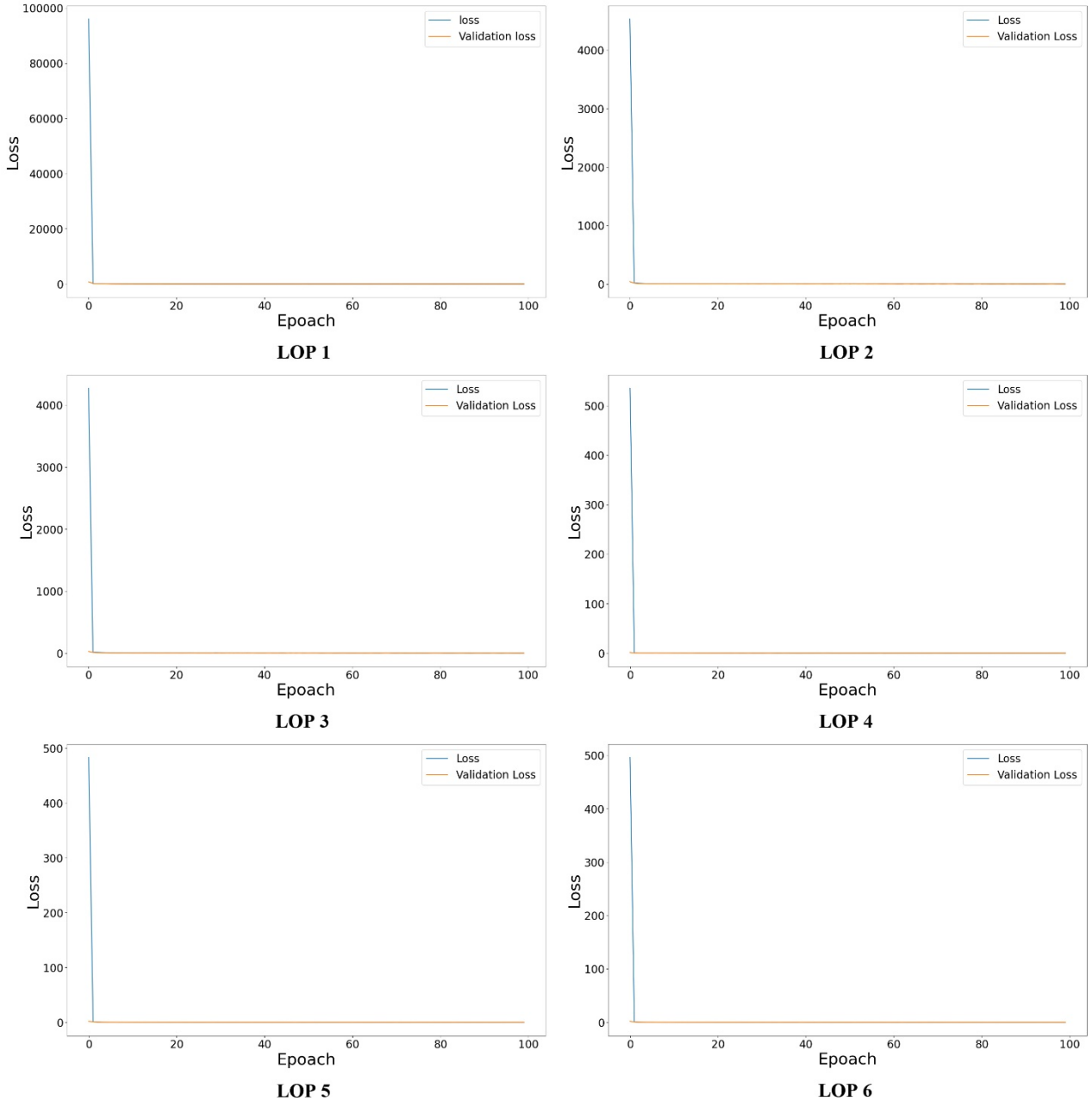



*Figure 5.2. ANN model for stress strain curve loss on the 6 LOPs*

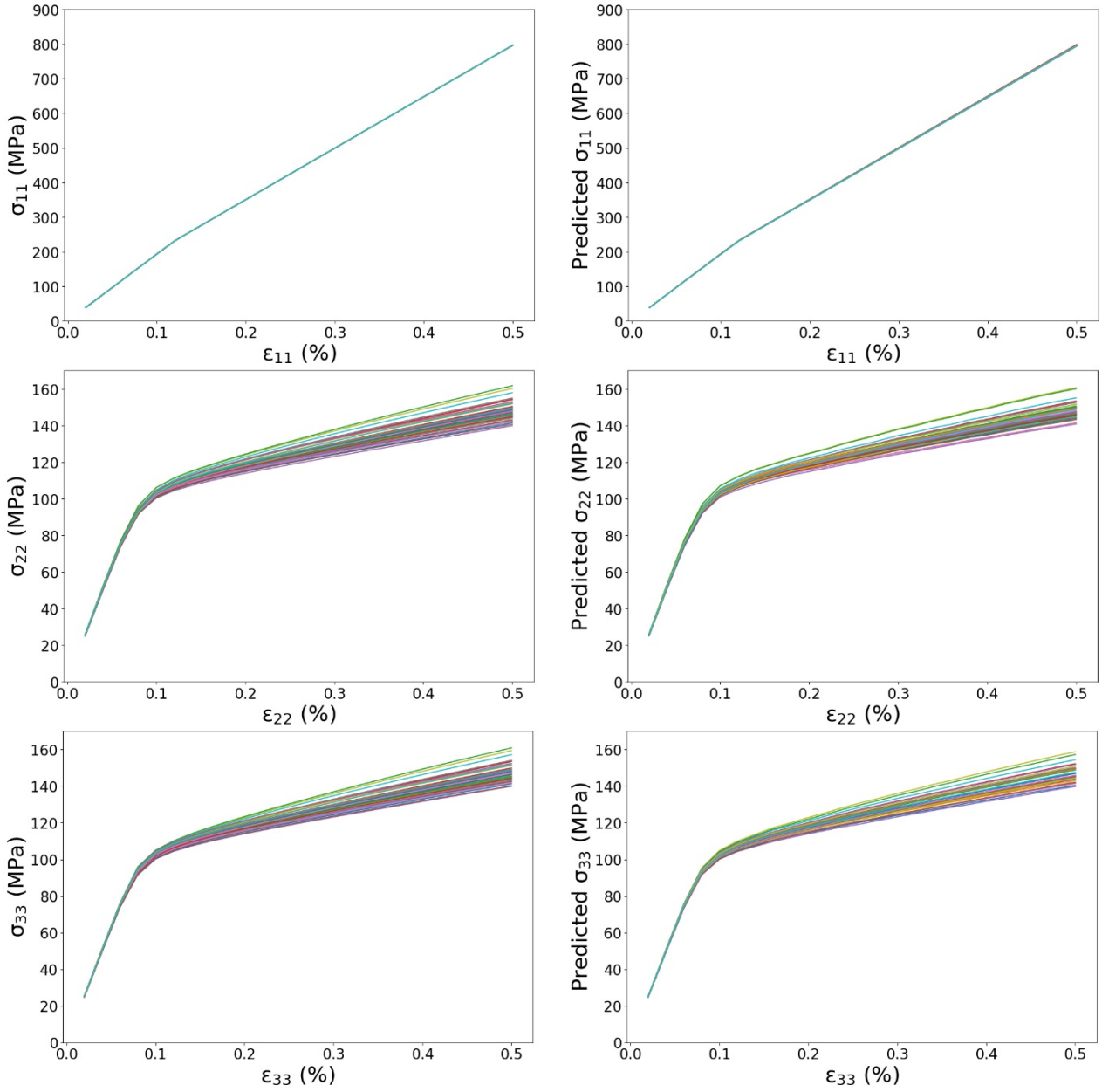



*Figure 5.3. ANN model predication comparison of stress strain curve on LOP1 to LOP3*

Figure 5.3 illustrates the actual 50 stress-strain curves on the left-hand side versus the predicted 50 stress-strain curves from the verification set on the right-hand side for the first 3 LOPs. The actual curves and the predicted curves look almost identical. More selective comparison is presented in the sequel.

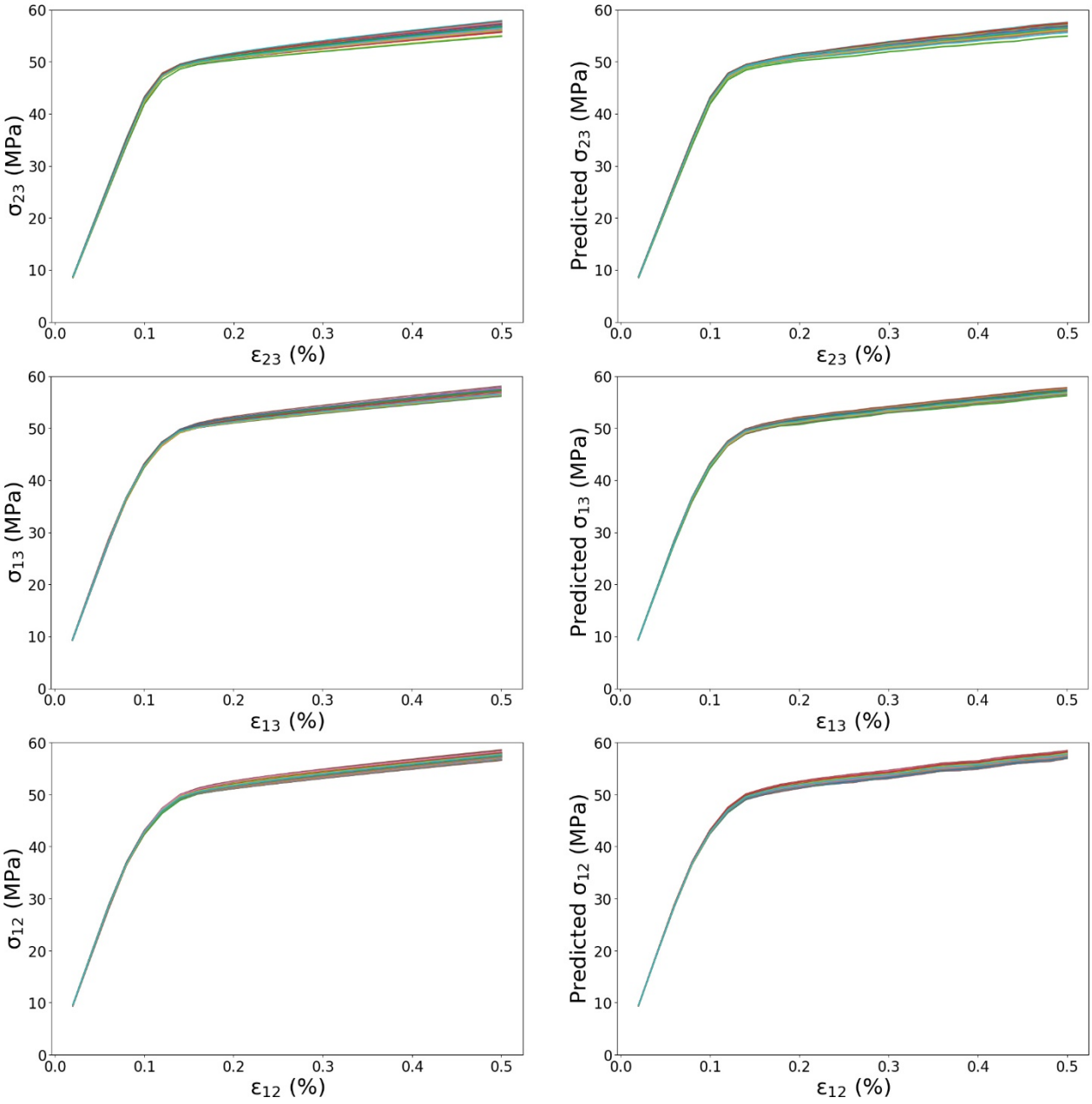



*Figure 5.4. ANN model predication comparison of stress strain curve on LOP4 to LOP6*

Similarly, Fig. 5.4 illustrates the actual 50 stress-strain curves on the left-hand side versus the predicted 50 stress-strain curves from the verification set on the right-hand side for the last 3 LOPs. The actual curves and the predicted curves look almost identical. To further check if the actual and predicted curves match for the 50 microstructural realizations, the two sets of curves were compared individually and the results indicated that the majority of the actual and predicted curves overlapped

each other. For brevity, here we select the bottom, intermediate and top curves for the comparison to avoid redundancy.

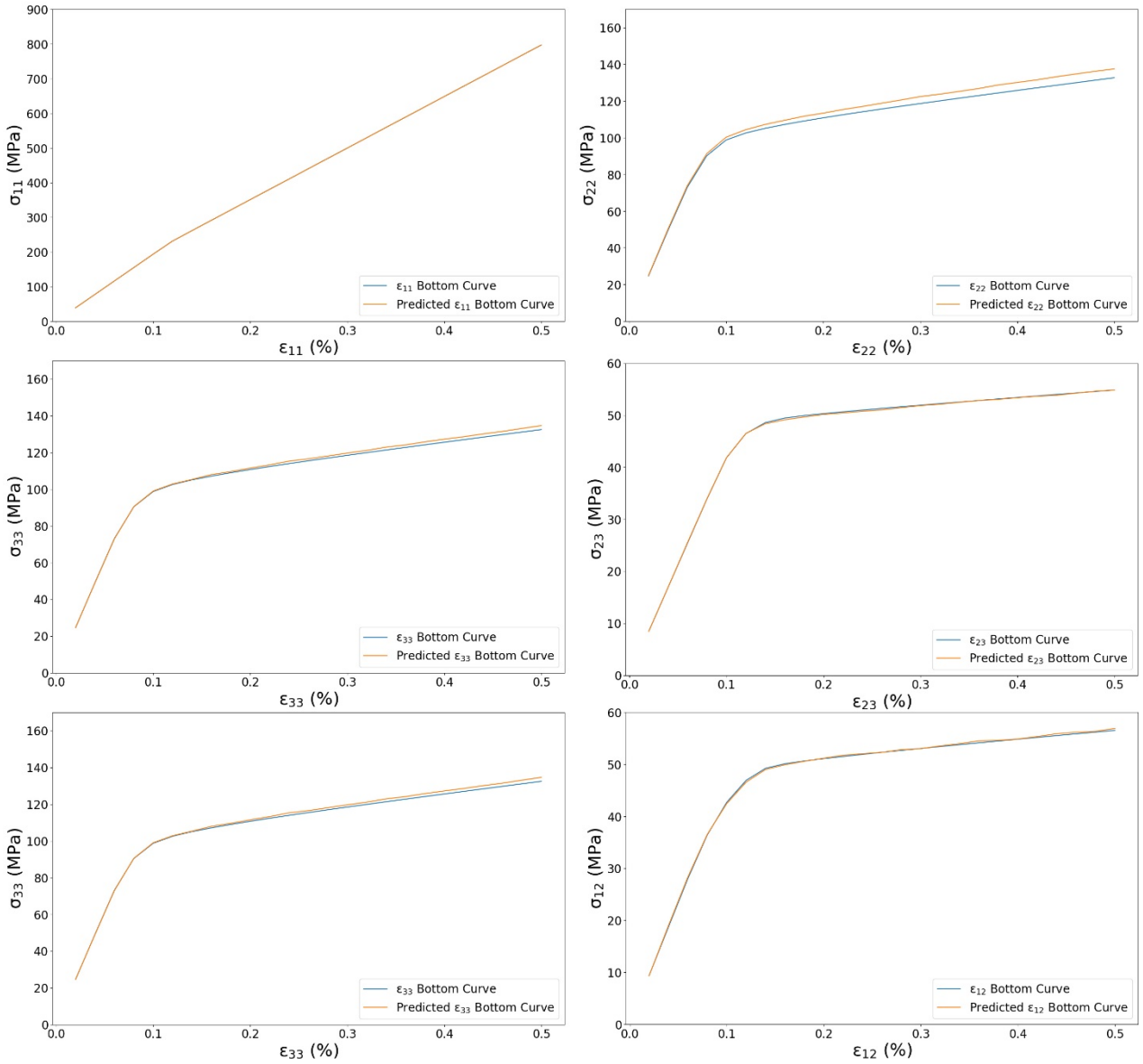



*Figure 5.5. ANN model predication comparison of the top stress strain curve for 6 LOPs*

Figure 5.5 presents comparison of the actual and predicted bottom stress-strain curves for the 6 LOPs. With the exception of a small discrepancy between the actual and predicted curves for LOP2, the actual and predicted curves for the remaining 5 LOPs almost overlap each other, which indicates good performance of the ANN model.

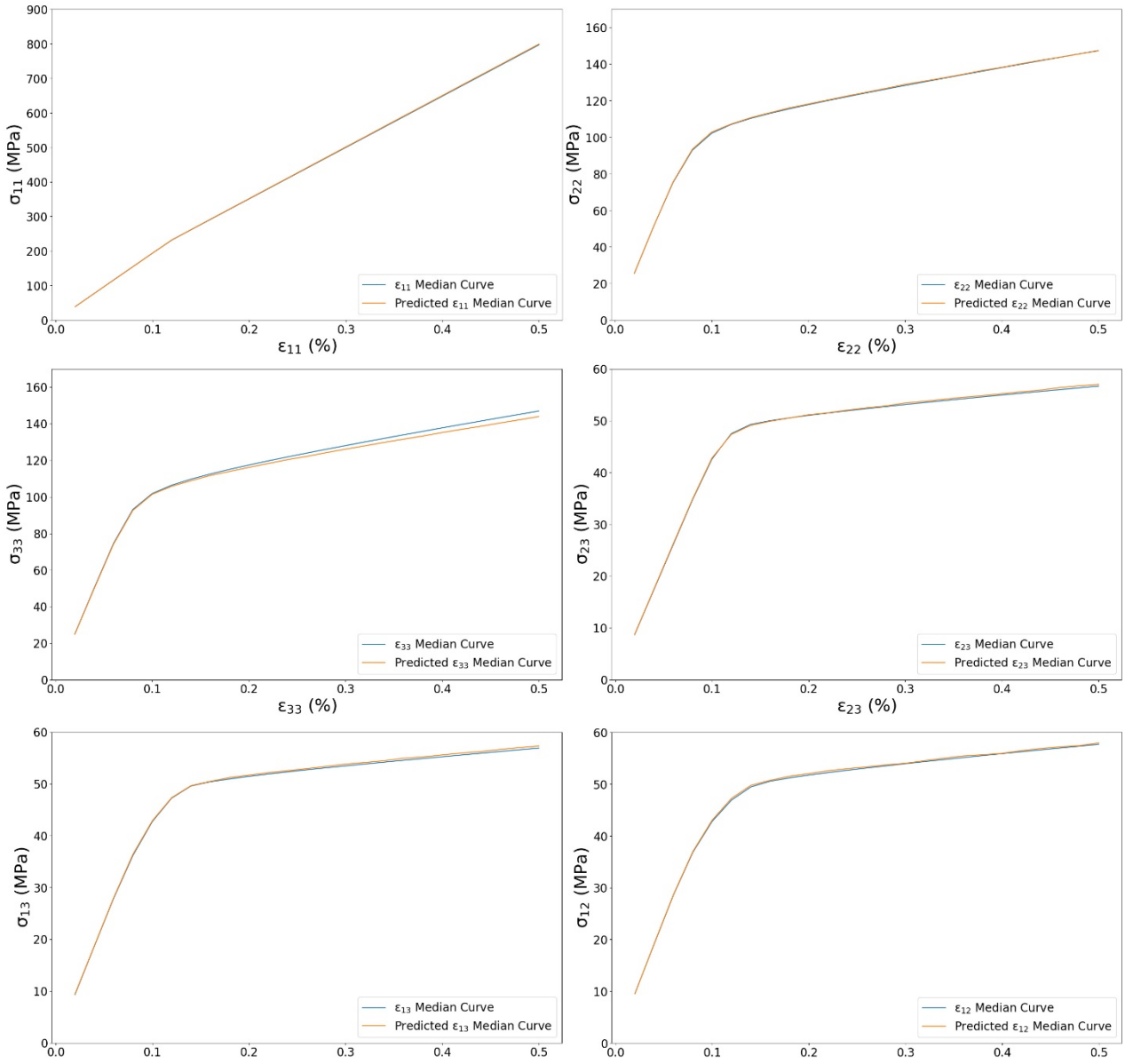



*Figure 5.6. ANN model predication comparison of the median stress strain curve for 6 LOPs*

Figure 5.6 illustrates the same comparison for the intermediate actual and predicted stress-strain curves for the 6 LOPs. Like the comparison of the bottom stress-strain curves in the preceding figure, the actual and predicted intermediate stress-strain curves for the 6 LOPs almost overlap each other.

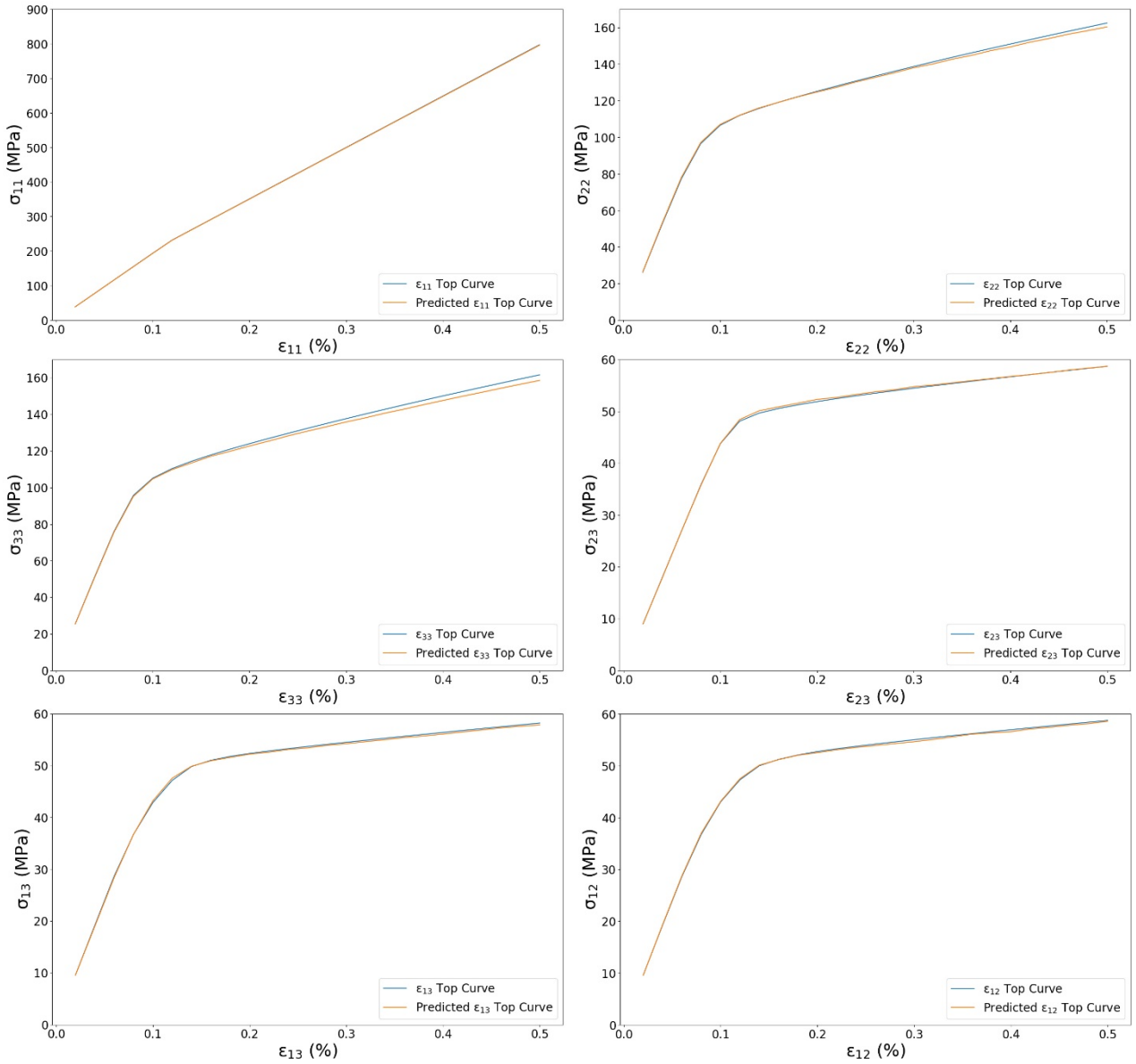



*Figure 5.7. ANN model predication comparison of the bottom stress strain curve for 6 LOPs*

Finally, Fig. 5.7 illustrates the comparison of the actual and predicted top stress-strain curves for the 6 LOPs. As in the case of the bottom and intermediate stress-strain curve comparison, the actual and predicted top stress-strain curves for the 6 LOPs also almost overlap each other.

## 5.2 CNN model performance on the stress-strain data sets

Figure 5.8 illustrates the CNN model accuracy rate during the training epochs of the 6 LOPs. The accuracy evolution with increasing epoch count for the five loading options LOP1,2,3,5,6 is similar. For the loading option LOP4, both the accuracy rate and the validation accuracy rate during

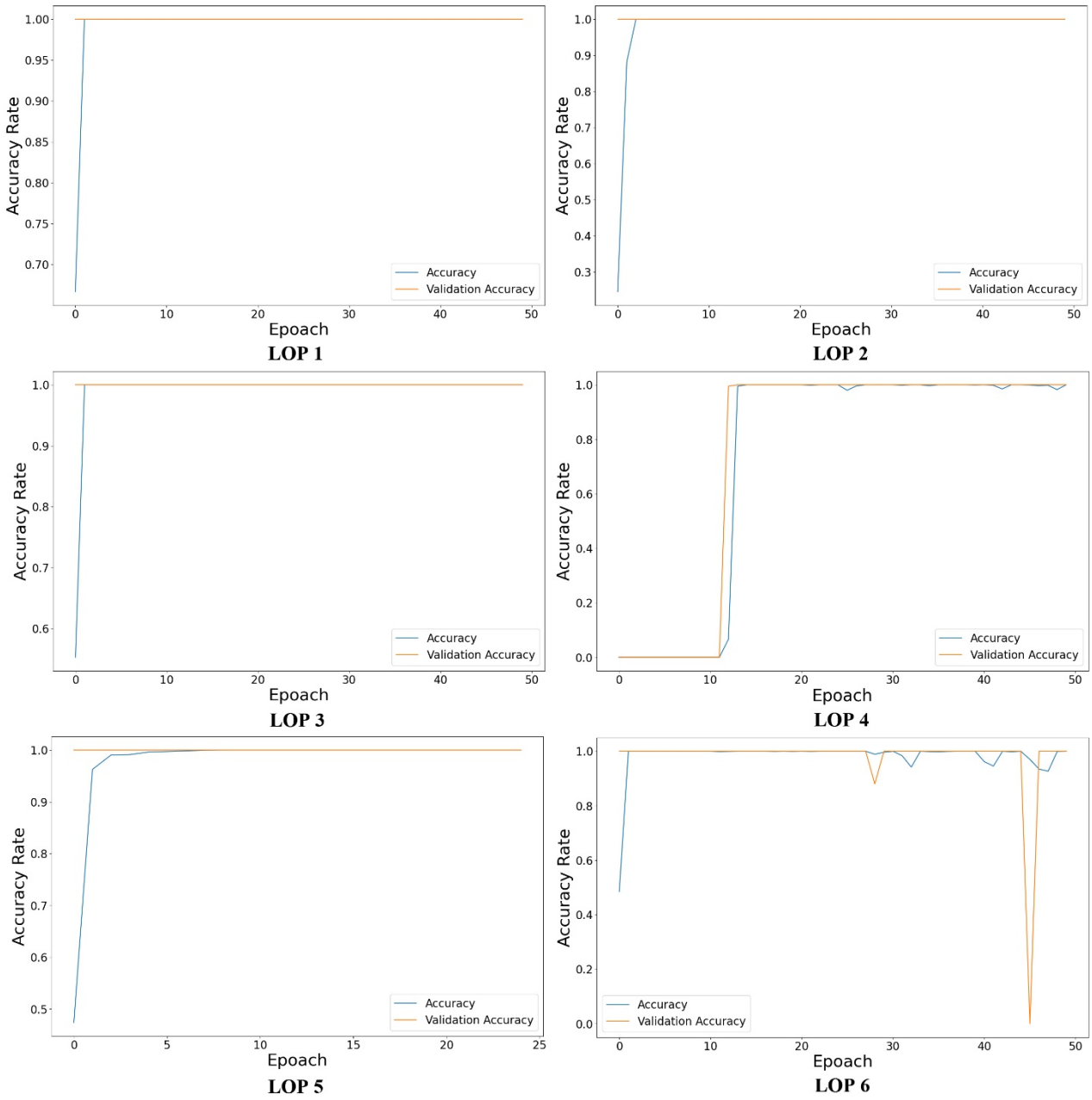



*Figure 5.8. CNN model for stress strain curve accuracy rate on the 6 LOPs*

the first 10 epochs are somewhere around 0, then increase rapidly and remain constant at the level around 100%. The accuracy rate and the validation accuracy rate for LOP6 shift up and down after the 25 epochs, which indicates the validation set size is small for the CNN model based on LOP6.

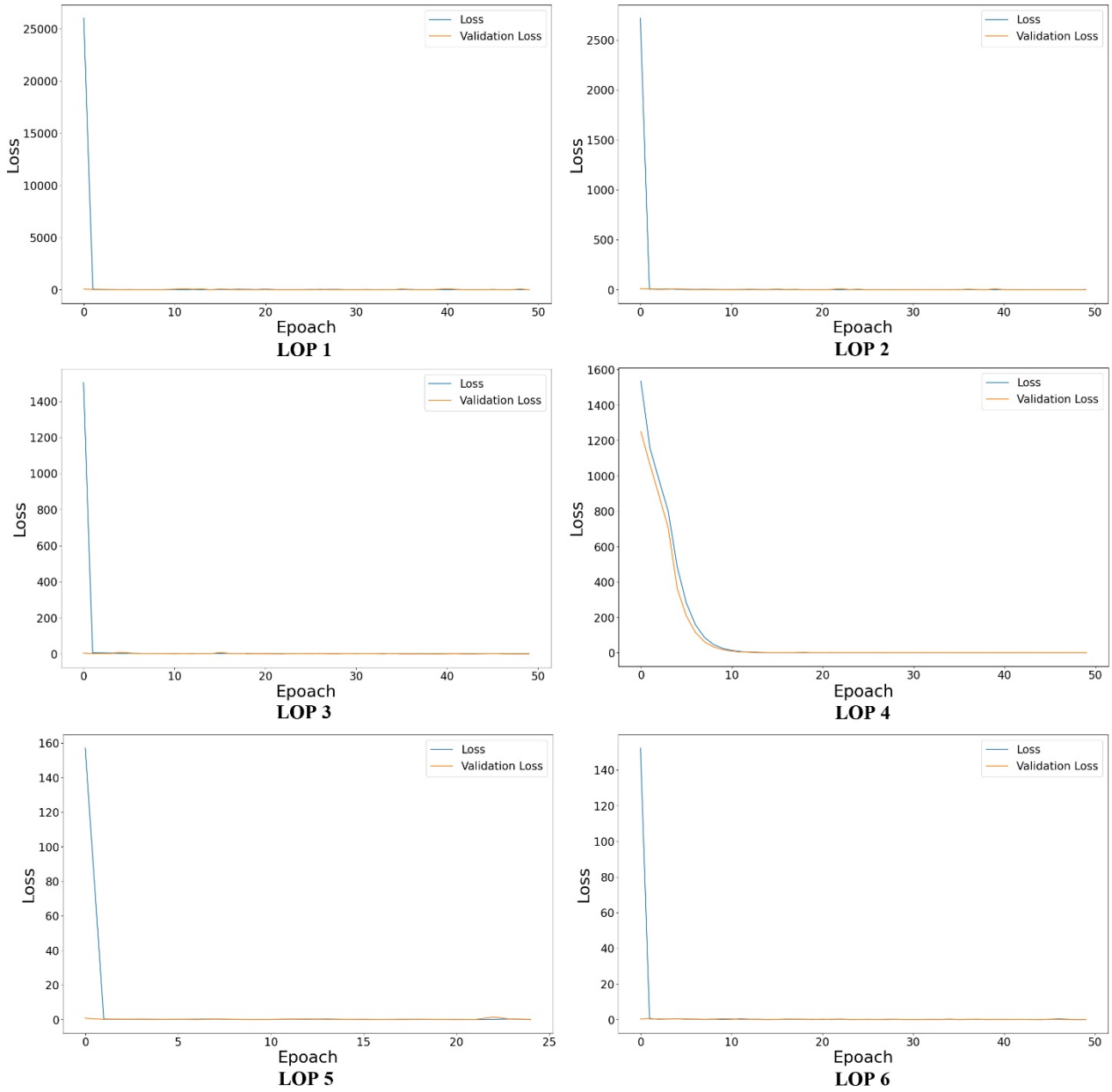



*Figure 5.9. CNN model for stress strain curve loss on the 6 LOPs*

Figure 5.9 illustrates the Loss change for the 6 LOPs. The 5 Loss functions for the five loading options LOP1,2,3,5,6, which are calculated from the difference between the actual training response and the predictions, decrease rapidly during the first 2 epochs and finally remain constant

somewhere at 0. The validation loss functions, which are calculated from the difference between the actual validation response and the predictions, start from somewhere a little greater than 0, decrease in the first 2 epochs and finally remain constant at around 0. For the loading option LOP4, the decrease of the loss and validation loss is much gentler.

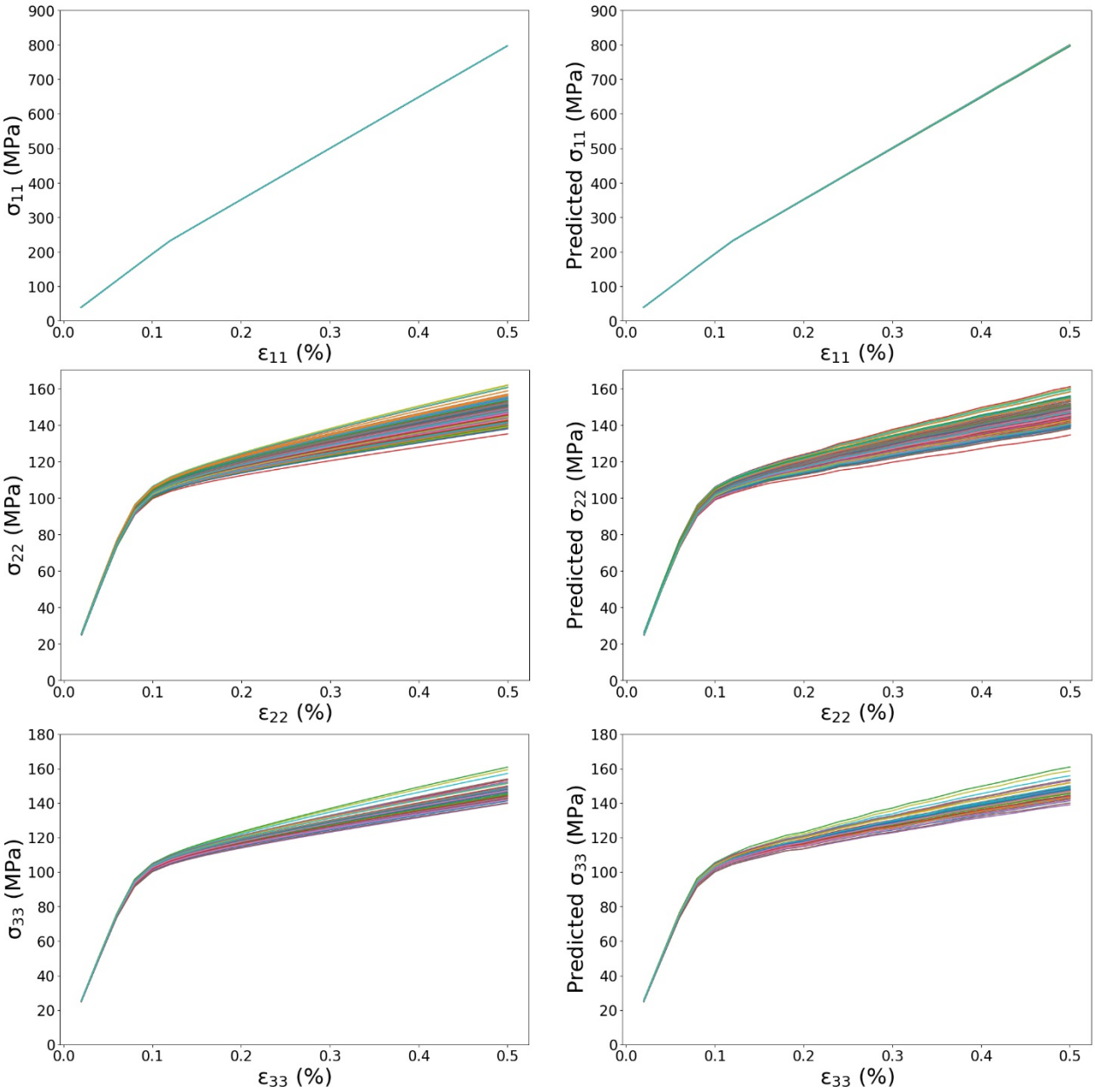



*Figure 5.10. CNN model predication comparison of stress strain curve on LOP1 to LOP3*

Figure 5.10 illustrates comparison between the actual 50 stress-strain curves taken from the verification set, shown on the left hand side with the predicted curves shown on the right hand side, for the first 3 LOPs. The actual curves and the predicted curves look almost identical.

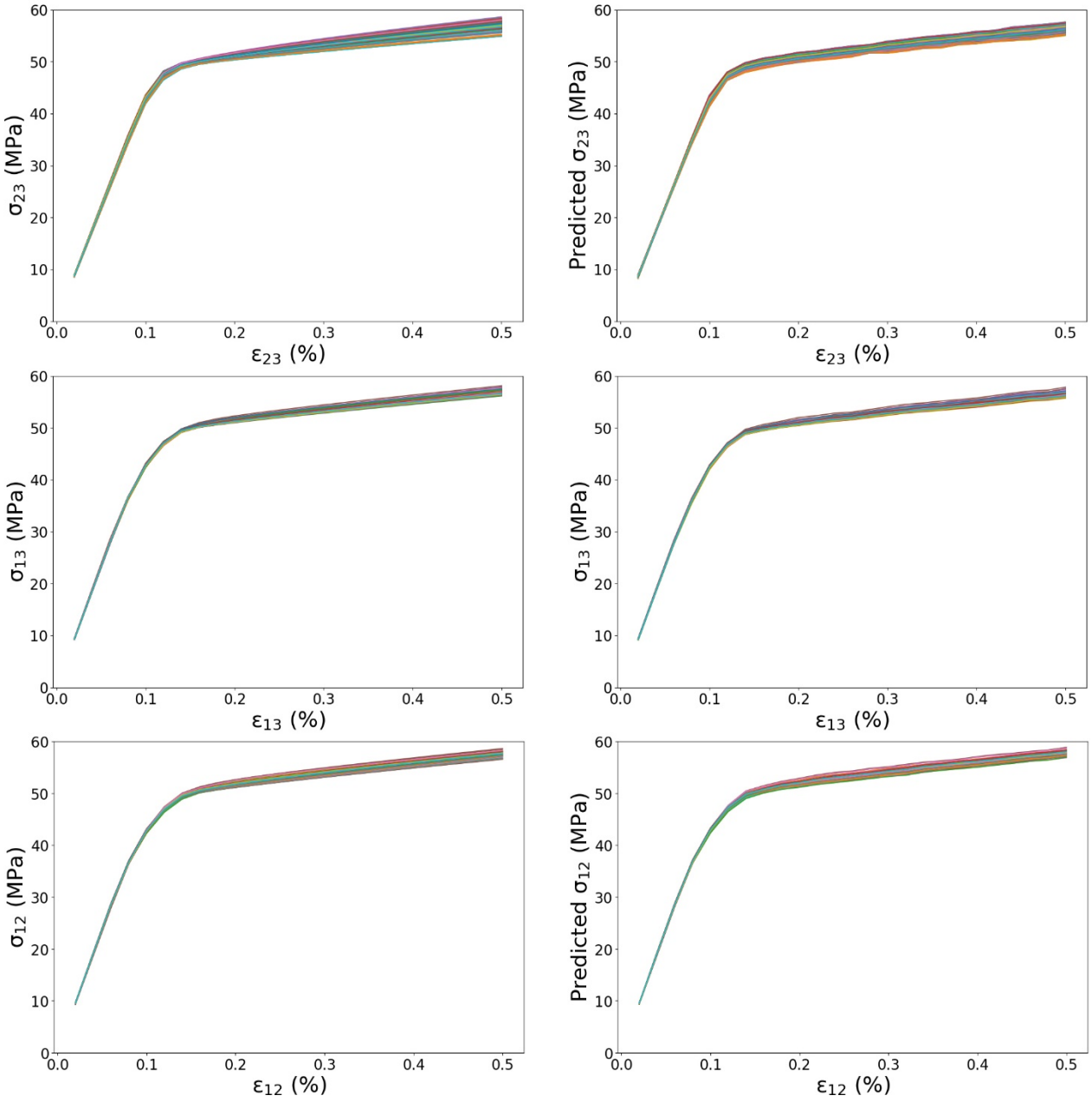



*Figure 5.11. CNN model predication comparison of stress strain curve on LOP4 to LOP6*

Similarly, Fig. 5.11 illustrates the corresponding comparison for the last 3 LOPs. The actual curves and the predicted curves look almost identical. To further check if the actual and predicted curves match, the individual actual and predicted curves were compared one-by-one, and the

comparison revealed that the actual and predicted curves overlapped each other. For brevity, here we select the bottom, intermediate and top curves for this comparison to avoid redundancy.

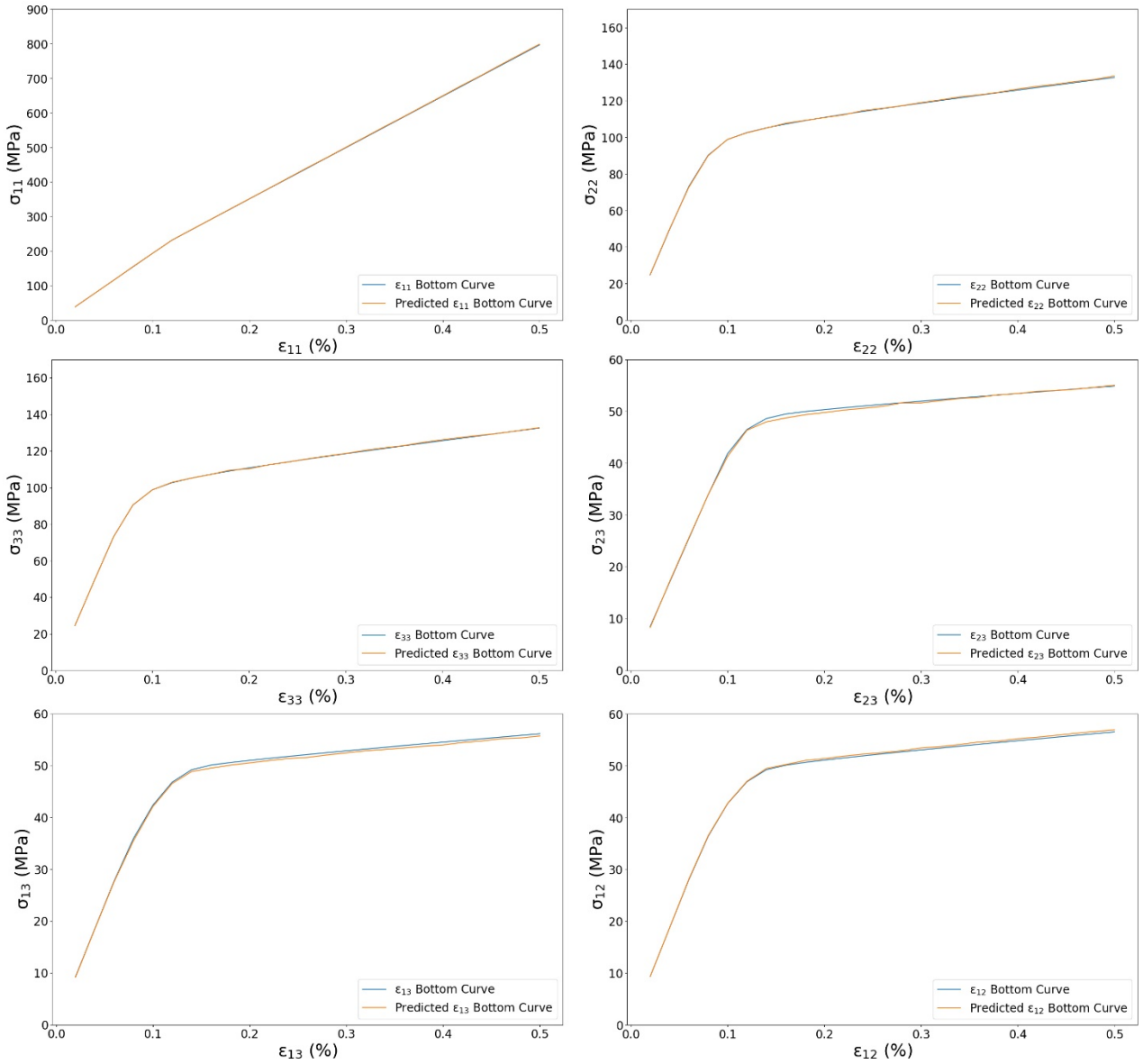



*Figure 5.12. CNN model predication comparison of the top stress strain curve for 6 LOPs*

Figure 5.12 presents the comparison between the actual and predicted bottom stress-strain curves for the 6 LOPs. The actual and predicted curves for the 6 LOPs almost overlap each other, which indicates good performance of the CNN model.

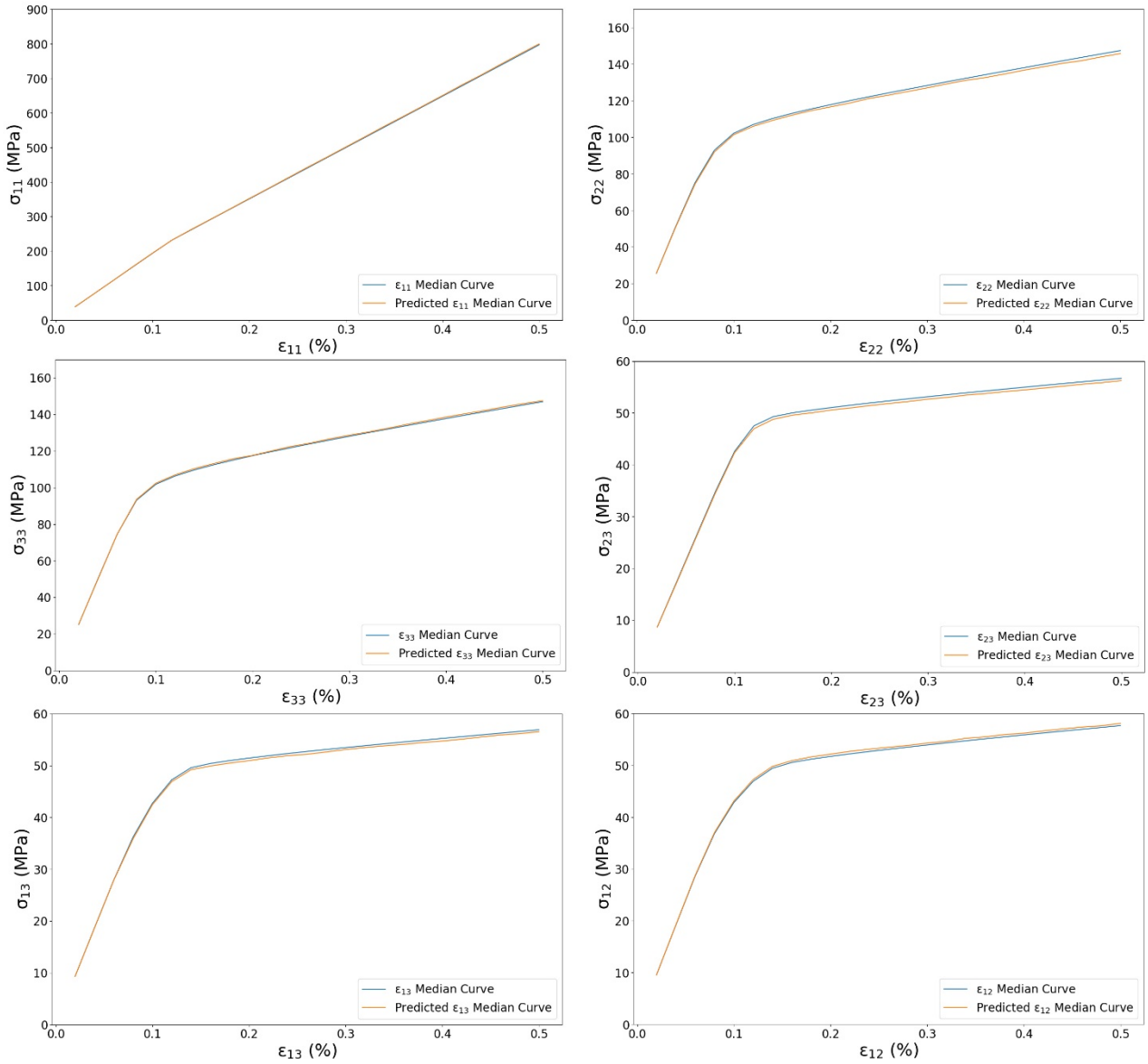



*Figure 5.13. CNN model predication comparison of the median stress strain curve for 6 LOPs*

Similarly, Fig 5.13 presents comparison of the actual and predicted intermediate stress-strain curves for the 6 LOPs. Like the bottom curve comparison for the 6 LOPs, the actual and predicted intermediate curves for the 6 LOPs almost overlap each other.

Finally, Fig 5.14 illustrates the corresponding comparison of the top stress-strain curves for the 6 LOPs, demonstrating near overlap and hance good performance of the developed CNN algorithm.

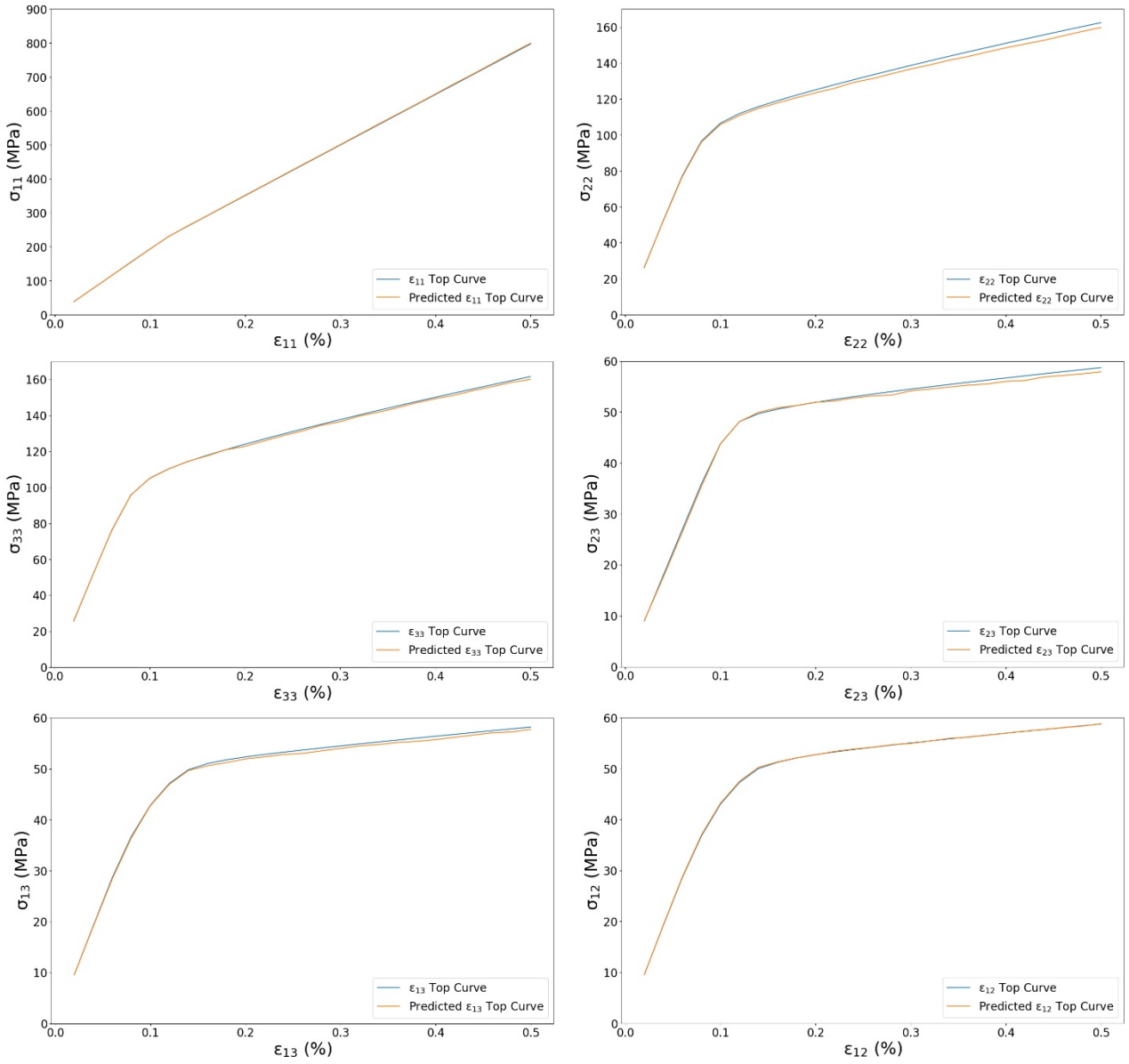



*Figure 5.14. CNN model predication comparison of the top stress strain curve for 6 LOPs*

## 5.3 ANN model performance on the 13 homogenized moduli

Figure 5.15 illustrates the ANN model accuracy rate and validation accuracy rate evolution during the training epochs on the 18,000 data training set. Both the accuracy rate and validation accuracy rate start somewhere around 100% and overlap each other. This might be a sign of overfitting. The loss and validation loss evolution shown on the right-hand side of the figure both start somewhere around $1.864 \times 10^6$ and $1.844 \times 10^6$. The actual performance of the ANN model on the 13 homogenized elastic moduli is assessed in Fig. 5.16.

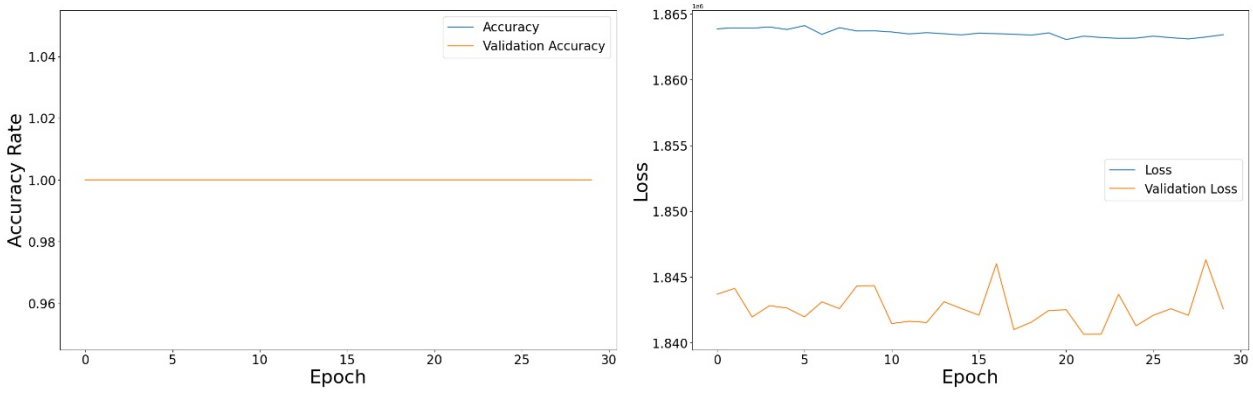



*Figure 5.15. ANN model accuracy and loss for the homogenized moduli*

Figure 5.16 is composed of 13 scatter plots showing the actual homogenized elastic moduli and the corresponding ANN model predictions. As observed in the 13 scatter plots, there is a clear regression pattern around the line $y = x$ for the nine homogenized elastic moduli $C_{11}$, $C_{12}$, $C_{13}$, $C_{22}$, $C_{23}$, $C_{33}$, $C_{44}$, $C_{55}$ and $C_{66}$. These are the moduli describing the response of orthotropic materials in absence of terms indicative of monoclinic behavior the extent of which is given by the four moduli $C_{14}$, $C_{24}$, $C_{34}$ and $C_{56}$. The performance of the ANN algorithm for these four moduli is not successful. This may be due to the fact that the ANN model inputs do not provide enough information or features to extract and make correct predictions. Thus, the training data size might be not enough. Nonetheless, it must be pointed out that these moduli are very small relative to the nine moduli representative of orthotropic materials, likely requiring additional features that supplement the employed fiber centers for accurate prediction.

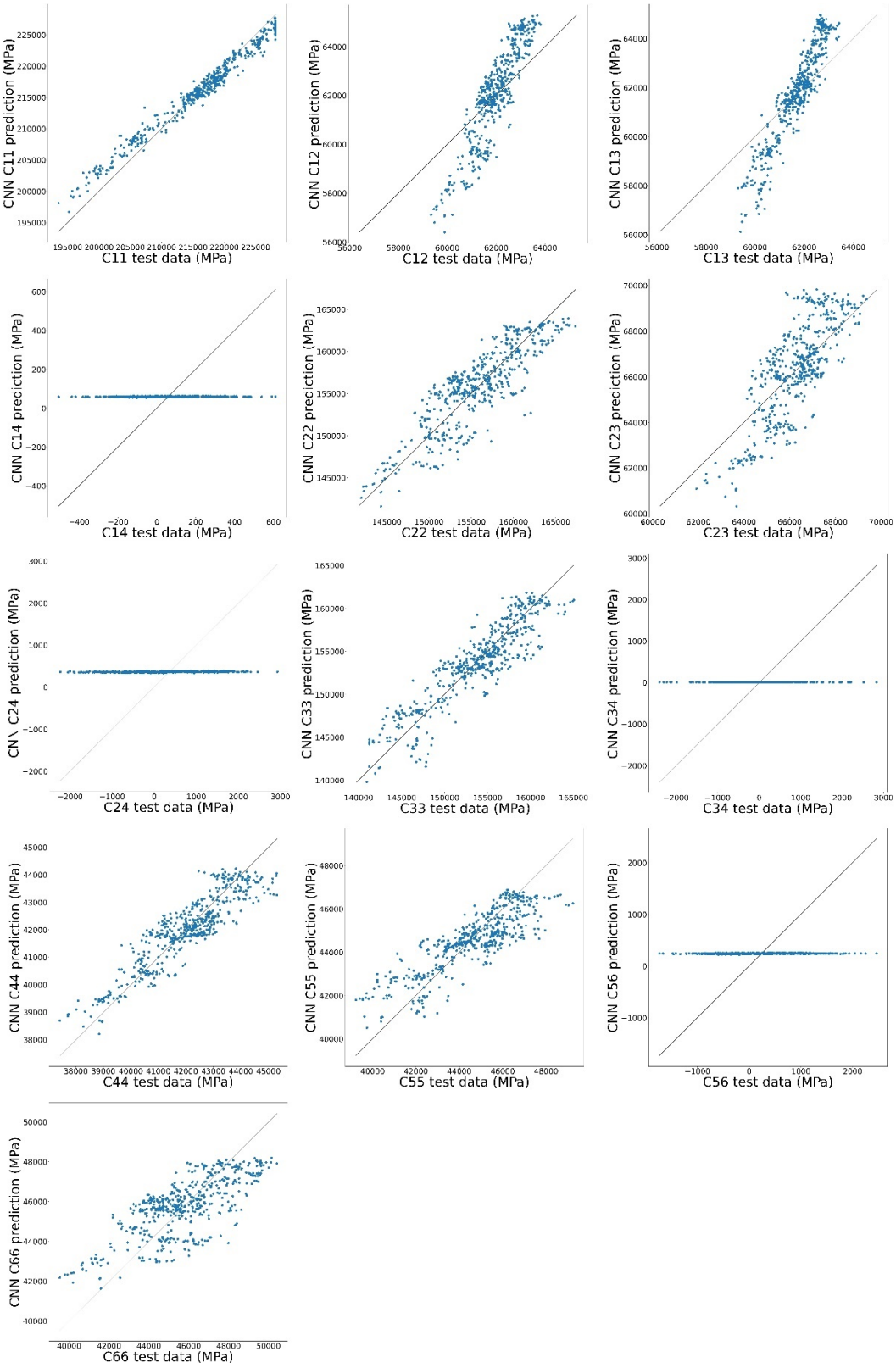



*Figure 5.16. ANN model performance visualization for the homogenized moduli*

## 5.4 CNN model performance on the 13 homogenized moduli

Figure 5.17 illustrates the CNN model accuracy rate and validation accuracy rate evolution during the training epochs on the 18,000 data training set. The accuracy rate starts from 0 and validation accuracy rate starts somewhere around 100%, both finally remain constant at 100% and overlap each other. The loss variation with epoch count on the right-hand side of the figure starts at a very large number (approx. $1.586\times10^{8}$) and eventually decreases to a relatively small, constant level. In contrast, the validation loss variation with epoch count starts at a small number (approx. 20,000) and further decreases to a small number. The actual performance of the CNN model on the 13 homogenized moduli is assessed in Fig. 5.18.

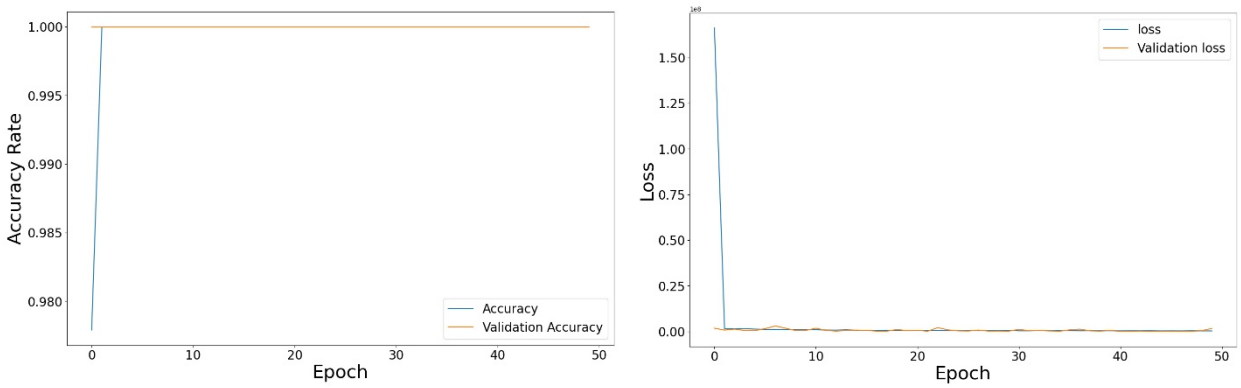



*Figure 5.17. CNN model accuracy and loss for the homogenized moduli*

Figure 5.18 is composed of 13 scatter plots of the actual homogenized moduli and the corresponding CNN model predictions. As observed in the 13 scatter plots, all the data points representing actual and predicted moduli lie on top of the regression line $y = x$. This indicates that the CNN model's performance is capable of accurately predicting not only the nine moduli indicative of orthotropic behavior but also four very small moduli that indicate slight departures from orthotropy, effectively resulting in slightly monoclinic RUC realizations. Moreover, as will be illustrated in the following section, the 13 moduli were generated very quickly relative to the full-scale FVDAM calculations.

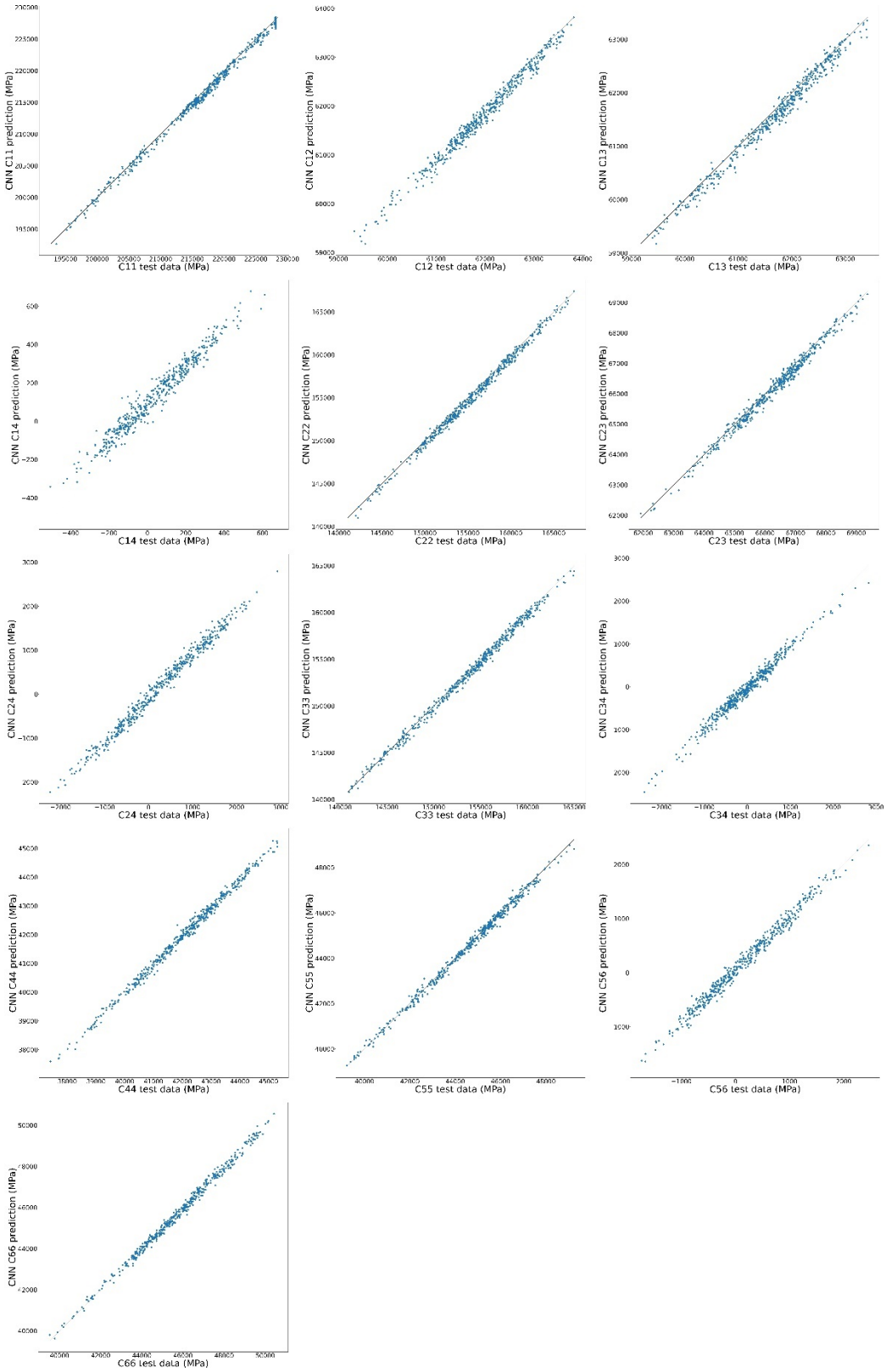



*Figure 5.18. CNN model performance visualization for the homogenized moduli*

## 5.5 Comparison of execution times

The execution times taken to generate the stress-strain responses and the homogenized moduli are dramatically different between the ANN/CNN algorithms and the FVDAM simulations. Overall, the performance of the ANN/CNN algorithms is several orders of magnitude faster than the FVDAM calculations. The latter involve the solution of a complicated unit cell problem that requires incremental and repetitive solution algorithm due to the load-dependent evolution of plasticity inside the RUC, which in turn is impacted by the fiber distributions and loading direction.

### 5.5.1 Homogenized stress-strain response

| Execution time for one stress strain response | ANN | CNN | FVDAM |
|---|---|---|---|
| LOP 1 | 0.001s | 0.001s | 34s |
| LOP 2 | 0.001s | 0.001s | 90s |
| LOP 3 | 0.001s | 0.001s | 81s |
| LOP 4 | 0.001s | 0.001s | 87s |
| LOP 5 | 0.001s | 0.001s | 94s |
| LOP 6 | 0.001s | 0.001s | 91s |

*Table 5.1 Execution time for one stress strain response from ANN/CNN and FVDAM*

Table 5.1 shows that the execution times to predict one stress-strain response for both the ANN and CNN models is approx. 0.001 seconds. By contrast, the execution time of the FVDAM algorithm to calculate one stress-strain response requires 34 seconds for the loading option LOP1, and between 80 and 90 seconds for the remaining LOPs. The difference lies in the speed with which the iterative process that calculates the evolution of plastic strains converges, which does not require many iterations for loading along the fiber direction relative to loading in the transverse directions. The above comparison illustrates that execution time reductions of four orders of magnitude are obtained from the trained ANN/CNN models relative to full-scale FVDAM calculations.

### 5.5.2 Homogenized Moduli

| | ANN | CNN | FVDAM |
|---|---|---|---|
| Execution time for one homogenized moduli response | 0.001s | 0.001s | 32.8 s |

*Table 5.2 Execution time for one homogenized moduli response from ANN/CNN and FVDAM*

Table 5.2 shows that the execution time to predict one set of homogenized moduli for both the ANN and CNN model is 0.001 seconds. By contrast, the execution time of the FVDAM algorithm to calculate the 13 homogenized moduli requires 32.8 seconds. The ANN/CNN models continue to outperform by far the elastic FVDAM calculations which do not require iterations because of the absence of plasticity.

## 5.6 Summary

In this chapter, we discussed the performance of the ANN and CNN models in predicting the stress-strain response and the 13 homogenized moduli of a unidirectional boron/aluminum composite with random fiber distributions. In terms of the stress-strain response, both the ANN and CNN models designed here yield very good predictions and can be employed as reliable tools to calculate the elastic-plastic stress-strain response of unidirectional composite materials. As for the 13 homogenized moduli, the ANN model does not perform well due to the input data type and size of the training data set. By contrast, the CNN model which was trained on RUC images produces very good and precise predictions of both the homogenized moduli and the elastic-plastic stress-strain curves under 6 different types of unidirectional loading. This result is important in developing accurate ML-based computational models for implementation in multi-scale analyses of large-scale composite structures.

# Chapter 6

# Summary and Conclusions

In this thesis, a computational scheme was developed to generate thousands of microstructure realizations of unidirectional composites with random fiber distributions employed by the homogenization theory called FVDAM. Once the microstructure was realized using randomly distributed fiber centers, it was then discretized into equally dimensioned subvolumes, and the material assignment matrix was created for input into FVDAM simulation. Subsequently, the FVDAM homogenization theory was incorporated into a python-driven interface that enabled generation of thousands of elastic-plastic stress strain curves for unidirectional metal matrix composites with random fiber distribution. Using this code, 2,000 stress-strain curves were generated under 6 fundamental loadings based on the microstructure realizations and 20,000 microstructure realizations for the determination of the homogenized stiffness tensor. The 2,000 microstructure realizations and the corresponding stress-strain responses, and the 20,000 microstructure realizations and the related homogenized stiffness tensor, were then employed in ANN and CNN architectures that were designed and optimized for predictive purposes.

First, the variations observed in the homogenized elastic moduli for all the microstructural realizations were small, with negligibly small contributions from the terms suggestive of monoclinic behavior. This indicates that the generated RUC microstructures, despite fiber distribution randomness, produced essentially orthotropic behavior characterized by 9 elastic homogenized moduli, likely due to the relatively large number of fibers contains within the RUCs. Moreover, all the density distributions of the thirteen homogenized stiffness matrix elements for the generated microstructures followed normal distribution, and thus were good enough for training by the ANN and CNN model.

The generated stress-strain curves were then examined for the 6 uniaxial loading directions relative to the RUC microstructural realizations. First, very small graphical differences were observed in the elastic response for the different RUC microstructures, indicating small differences

in the homogenized moduli, as observed in the small variations of the 13 homogenized moduli with microstructural realization. Substantially greater variations were observed in the elastic-plastic region that were dependent on the loading direction. Uniaxial loading by $\sigma_{11}$ only along the fiber direction, LOP 1, produced virtually the same response regardless of the substantial variations in the RUC microstructures due the fiber constraint that limited the variations in the plastic strain distributions. In contrast, substantial deviations in the elastic-plastic stress-strain response were observed under loadings from LOP 2 to LOP 6. For uniaxial loading by transverse $\bar{\sigma}_{22}$ stress only oriented along the horizontal axis, LOP 2, the stiffest response was produced by the RUC with the most ordered microstructure characterized by aligned fiber rows and the softest by RUC microstructure with most disordered fibers. The microstructure with some degree of order produces an intermediate stress-strain response. For uniaxial loading by transverse $\bar{\sigma}_{33}$ stress only oriented along the vertical axis, LOP 3, the corresponding effect of RUC microstructure on the homogenized stress-strain response was similar, with the stiffest response produced by the RUC with the most ordered microstructure characterized by aligned fiber rows and the softest by RUC microstructure with most disordered fibers. The microstructure with some degree of order produces an intermediate stress-strain response. In contrast, under uniaxial transverse shear loading only, LOP 4, it was the most ordered microstructure that produced the most compliant response, with the stiffest response generated by the most disordered RUC. The effect of microstructure on the homogenized response was also less pronounced relative to the two preceding cases involving transverse normal stresses $\bar{\sigma}_{22}$ and $\bar{\sigma}_{33}$ where the greatest scatter in the elastic-plastic region was observed. The effect of RUC microstructures on the homogenized stress-strain responses under uniaxial shear loading by $\bar{\sigma}_{13}$ and $\bar{\sigma}_{12}$ in the $x_1$-$x_3$ and $x_1$-$x_2$ planes respectively, was even smaller despite differences in the RUC microstructures.

Finally, the ANN and CNN models were trained in predicting the stress-strain response and the 13 homogenized moduli. Both the ANN and CNN models designed here yielded very good predictions of the elastic-plastic stress-strain responses under the 6 uniaxial loading directions, and therefore may be employed as good tools to calculate the stress-strain response of unidirectional composite materials based on the microstructures given in terms of either the fiber center locations or RUC images. Only 1,900 RUC realizations were required to achieve such good performance for both models. In contrast, substantially greater number of microstructural realizations than those

generated for the prediction of elastic-plastic stress-were required to achieve good results for the homogenized elastic moduli. The number of microstructural realizations were increased from 2,000 to 10,000, 15,000 and finally 20,000 and subsets of these realizations were used for prediction after training. Whereas the CNN model produced accurate results after being trained and optimized using the 20,000 microstructural realizations, including the very small, moduli indicative of monoclinic behavior, the ANN model based just on the fiber center distributions did not perform nearly as well. It may be concluded that large numbers of microstructural features are required to accurately predict elastic properties of random composite materials with small homogenized moduli differences due to fiber distribution variations. This result is important in developing accurate ML-based computational models for implementation in multi-scale analyses of large-scale composite structures.

Perhaps most importantly, the execution times required to predict the homogenized elastic-plastic response of random fiber composites based on the ANN/CNN algorithms are several orders of magnitude smaller that the full-scale calculations based on the FVDAM homogenization theory. In particular, the execution time required of the ANN/CNN model to predict one stress-strain response, or one homogenized moduli response is 0.001 seconds. By contrast, the FVDAM algorithm takes anywhere from 34 to 94 seconds to predict a single homogenized stress-strain response depending on the orientation of the applied load relative to the fiber direction. Similarly, the calculation of the 13 homogenized elastic moduli requires 32.8 seconds for FVDAM versus just 0.001 seconds for the CNN algorithm. These dramatic reductions in execution times make possible calculation of the elastic-plastic response of large composite structures that experience plasticity during deformation.

In conclusion, machine learning and deep learning algorithms can be good alternative computational methods for multi-scale analysis of composite structures, with the combined advantages of good accuracy and time efficiency.

# References


1. Drago, A., & Pindera, M.J., 2007. Micro-macromechanical analysis of heterogeneous materials: Macroscopically homogeneous vs periodic microstructures. Comp. Sci. Technol., 67(6), 1243-1263.
2. Pindera, M-J., Khatam, H., Drago, A. S., and Bansal, Y., 2009. Micromechanics of spatially uniform heterogeneous media: a critical review and emerging approaches. Composites B, 40(5), 349-378.
3. Charalambakis, N., 2010. Homogenization techniques and micromechanics. A survey and perspectives. Appl. Mech. Rev., 63, 030803:1-10.
4. Charalambakis, N., Chatzigeorgiou, G., Chemisky, Y., Meraghni, F., 2018. Mathematical homogenization of inelastic dissipative materials: a survey and recent progress. Continuum Mech. Thermodyn, 30:1-51.
5. Cavalcante, M.A.A., Pindera, M.-J. and Khatam, H., 2012. Finite-volume micromechanics of periodic materials: past, present and future. Composites, Part B, 43(6), 2521-2543.
6. Drago, A.S., Pindera, M.J., 2008. A locally exact homogenization theory for periodic microstructures with isotropic phases. J. Appl. Mech., 75(5), 051010:1-14.
7. McCulloch, W.S., Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophysics, 5, 115-133.
8. Rumelhart, D.E., Hinton, G.E., Williams, R.J., 1986. Learning representations by back-propagating errors. Nature, 323, 533-536.
9. Ghaboussi, J., Garrett, J.H., Wu, X., 1991. Knowledge-based modeling of material behavior with neural networks. J. Engineering Mech., 117(1), 132-153.
10. Niezgoda, S.R., Kanjarla, A.K., Kalidindi, S.R., 2013. Novel microstructure quantification framework for databasing, visualization, and analysis of microstructure data. Integr. Mater. Manuf. Innov. 2, 54–80.
11. Ronneberger, O., Fischer, P., Brox, T., 2015. U-net: Convolutional networks for biomedical image segmentation. In Med. Image. Comput. Comput. Assist. Interv., 234–241.
12. McDowell, D.L., Lesar, R.A., 2016. The need for microstructure informatics in process structure-property relations. MRS Bull. 41, 587–593.
13. Bereau, T., Andrienko, D., Kremer, K., 2016. Research update: computational materials discovery in soft matter. APL Mater. 4, 053101.
14. Wodo, O., Broderick, S., Rajan, K., 2016. Microstructural informatics for accelerating the discovery of processing-microstructure-property relationships. MRS Bull. 41, 603–609.
15. Erichson, N.B., Mathelin, L., Yao, Z., Brunton, S.L., Mahoney, M.W., Kutz, J.N., 2020. Shallow neural networks for fluid flow reconstruction with limited sensors. Proc. R. Soc. A 476, 20200097:1-25.

16. Jang, D.P., Fazily, P., Yoon, J.W., 2021. Machine learning-based constitutive model for J2-plasticity. Int. J. Plasticity, 138, 102919:1-17.

17. Abueidda, D.A., Koric, S., Sobh, N.A., Sehitoglu, H., 2021. Deep learning for plasticity and thermo-viscoplasticity. Int. J. Plasticity, 136, 102852:1-30.

18. Muhammad, W., Brahme, A.P., Ibragimova, O., Kang, J., Inal, K., 2021. A machine learning framework to predict local strain distribution and the evolution of plastic anisotropy fracture in additively & manufactured alloys. Int. J. Plasticity, 136, 102867:1-29.

19. Mianrood, J.R., Siboni, N.H., Raabe, D., 2021. Teaching solid mechanics to artificial intelligence: a fast solver for heterogeneous solids. Comp. Mater. 99, 1-10.

20. Xu, K., Huang, D., Z., Darve, E., 2021. Learning constitutive relations using symmetric positive definite neural networks. J. Computational Physics, 428(1), 110072:1-35.

21. Masi, F., Stefanou, I., Vannucci, P., Victor Maffi-Berthier, V., 2021. Thermodynamics-based artificial neural networks for constitutive modeling. J. Mech. Phys. Solids, 147, 104277:1-28.

22. Yang, Z., Yu, C-H., Buehler, M.J., 2021. Deep learning model to predict complex stress and strain fields in hierarchical composites. Sci. Adv., 7:eabd7416.

23. Watkins, J., Wu, X., 2021. Machine Learning and its Application on Homogenized Moduli. CE 4955 Final Report. January 2021, University of Virginia.

24. Yin, S., He, Z., Pindera, M-J., 2021 A new hybrid homogenization theory for periodic composites with random fiber distributions. Composite Structures, 269, 113997:1-19.

25. Versteeg, H. K. and Malalasekera, W., 1995. An Introduction to Computational Fluid Dynamics: The Finite Volume Method. Prentice Hall, New York.

26. Cavalcante, M.A.A., Pindera, M.-J. & Khatam, H., 2012. Finite-volume micromechanics of periodic materials: past, present and future. Composites: Part B, 43(6), 2521-2543.

27. Pindera, M-J., Khatam, H., Drago, A. S., and Bansal, Y., 2009. Micromechanics of spatially uniform heterogeneous media: a critical review and emerging approaches. Composites: Part B 40(5), 349-378.

28. Bansal, Y. and Pindera, M-J., 2003. Efficient reformulation of the thermoelastic higher-order theory for fgms. J. Thermal Stresses, 26(11-12), 1055-1092.

29. Bansal, Y. and Pindera, M-J., 2006. Finite-volume direct averaging micromechanics of heterogeneous materials with elastic-plastic phases. Int. J. Plasticity, 22(5), 775-825.

30. Zhong, Y., Bansal, Y. and Pindera, M-J., 2004. Efficient reformulation of the thermal higher-order theory for FGM's with variable thermal conductivity. Int. J. Computational Engineering Science, 5(4), 795-831.

31. Cavalcante, M.A.A., Marques, S.P.C., Pindera, M-J., 2007a. Parametric formulation of the finite-volume theory for functionally graded materials. Part I: Analysis. J. Applied Mechanics, 74(5), 935-945.

32. Gattu, M., Khatam, H., Drago, A.S., Pindera, M-J., 2008. Parametric finite-volume micromechanics of uniaxial, continuously-reinforced periodic materials with elastic phases. J. Eng. Mater. Technol., 130(3), 031015:1-15.

33. Khatam, H., Pindera, M-J., 2009. Parametric finite-volume micromechanics of periodic materials with elastoplastic phases. Int. J. Plasticity, 25(7), 1386-1411.

34. Cardiff, P., and Demirdžić, I., 2018, "Thirty Years of the Finite Volume Method for Solid Mechanics," arXiv preprint, arXiv:1810.02105.

35. Adakroy, R. 2022. Generating Structure Property Maps for Composites. CE 4955 Final Report. May 2022, University of Virginia.

36. Mendelson, A., 1986. Plasticity: Theory and Application. Krieger Publishing Co., Malabar, FL (reprint edition).Mendelson

37. Williams, T. O. and Pindera, M-J., 1997. An analytical model for the inelastic axial shear response of unidirectional metal matrix composites. Int. J. Plasticity, 13(3), 261-289.

38. Pindera, M-J., Bansal, Y., 2007. On the micromechanics-based simulation of metal matrix composite response, J. Engrg. Matr. Techn., 129, 468-482.

39. Lakshmanan, S. How, When, and Why Should You Normalize / Standardize / Rescale Your Data? Towards AI. [Online] May 16, 2019.

40. Ronaghan, S. Deep Learning: Which Loss and Activation Functions should I use? Towards Data Science. [Online] July 26, 2018.

41. Zhou, V. A Simple Explanation of the Softmax Function. [Online] July 22, 2019.

# ANN_1200_1600_1800_2000_LOP6

October 14, 2022

[29]:
```
# pip install pandas
```

[30]:
```
from keras.models import Sequential
from keras.layers import Dense, Conv1D,Conv2D, Flatten
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import keras
import glob
import os
```

[31]:
```
projectDir = r'C:\Users\18810\Desktop\RUC Collection'
print(projectDir)
```

```
C:\Users\18810\Desktop\RUC Collection
```

[32]:
```
extension = 'csv'
all_filenames = glob.glob(os.path.join(projectDir + '\CSV 2000', '*.csv'))
all_filenames[0]
```

[32]:
```
'C:\\Users\\18810\\Desktop\\RUC Collection\\CSV 2000\\ann_input0000.csv'
```

[33]:
```
X = []
for i in range(2000):
    file = open(all_filenames[i])
    numpy_array = pd.read_csv(file, delimiter=',',header=None)
    # numpy_array = np. loadtxt(file, delimiter=',')
    X.append(numpy_array)
print(X[0])
```

```
             0
0   159.000000
1   211.000000
2    21.932944
3    26.432944
```

```
4    26.432944
5    79.298832
6    26.432944
7   132.164719
8    26.432944
9   185.030607
10   26.432944
11   26.432944
12   79.298832
13   79.298832
14   79.298832
15  132.164719
16   79.298832
17  185.030607
18   79.298832
19   26.432944
20  132.164719
21   79.298832
22  132.164719
```

[34]:
```
y = pd.read_csv('LOP_6_SIGMA12_collection.csv',header=None)
print(y.shape)
print(type(y))
y.head(--25)
```

```
(25, 2001)
<class 'pandas.core.frame.DataFrame'>
```

[34]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | \ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.0002 | 9.5686 | 9.5686 | 9.5718 | 9.5762 | 9.5686 | 9.5686 | 9.5786 | |
| 1 | 0.0004 | 19.1360 | 19.1360 | 19.1420 | 19.1510 | 19.1360 | 19.1360 | 19.1560 | |
| 2 | 0.0006 | 28.5980 | 28.5980 | 28.6170 | 28.6210 | 28.5980 | 28.6110 | 28.6370 | |
| 3 | 0.0008 | 36.4790 | 36.4790 | 36.4880 | 36.4780 | 36.4790 | 36.4960 | 36.4780 | |
| 4 | 0.0010 | 42.3200 | 42.3200 | 42.3150 | 42.3080 | 42.3200 | 42.3350 | 42.2870 | |
| 5 | 0.0012 | 46.3910 | 46.3910 | 46.3800 | 46.3810 | 46.3910 | 46.4070 | 46.3580 | |
| 6 | 0.0014 | 48.9660 | 48.9660 | 48.9500 | 48.9600 | 48.9660 | 48.9700 | 48.9300 | |
| 7 | 0.0016 | 50.1150 | 50.1150 | 50.1140 | 50.1200 | 50.1150 | 50.1160 | 50.1170 | |
| 8 | 0.0018 | 50.6550 | 50.6550 | 50.6560 | 50.6620 | 50.6550 | 50.6550 | 50.6620 | |
| 9 | 0.0020 | 51.1120 | 51.1120 | 51.1130 | 51.1190 | 51.1120 | 51.1120 | 51.1200 | |
| 10 | 0.0022 | 51.5370 | 51.5370 | 51.5380 | 51.5450 | 51.5370 | 51.5370 | 51.5460 | |
| 11 | 0.0024 | 51.9450 | 51.9450 | 51.9470 | 51.9540 | 51.9450 | 51.9450 | 51.9550 | |
| 12 | 0.0026 | 52.3410 | 52.3410 | 52.3420 | 52.3520 | 52.3410 | 52.3410 | 52.3520 | |
| 13 | 0.0028 | 52.7270 | 52.7270 | 52.7290 | 52.7390 | 52.7270 | 52.7280 | 52.7400 | |
| 14 | 0.0030 | 53.1080 | 53.1080 | 53.1090 | 53.1200 | 53.1080 | 53.1080 | 53.1200 | |
| 15 | 0.0032 | 53.4810 | 53.4810 | 53.4830 | 53.4950 | 53.4810 | 53.4810 | 53.4960 | |
| 16 | 0.0034 | 53.8500 | 53.8500 | 53.8520 | 53.8650 | 53.8500 | 53.8500 | 53.8660 | |
| 17 | 0.0036 | 54.2150 | 54.2150 | 54.2180 | 54.2310 | 54.2150 | 54.2150 | 54.2320 | |
| 18 | 0.0038 | 54.5770 | 54.5770 | 54.5790 | 54.5940 | 54.5770 | 54.5770 | 54.5940 | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 19 | 0.0040 | 54.9350 | 54.9350 | 54.9380 | 54.9530 | 54.9350 | 54.9350 | 54.9540 |
| 20 | 0.0042 | 55.2900 | 55.2900 | 55.2930 | 55.3090 | 55.2900 | 55.2900 | 55.3100 |
| 21 | 0.0044 | 55.6430 | 55.6430 | 55.6460 | 55.6630 | 55.6430 | 55.6420 | 55.6640 |
| 22 | 0.0046 | 55.9940 | 55.9940 | 55.9970 | 56.0150 | 55.9940 | 55.9930 | 56.0160 |
| 23 | 0.0048 | 56.3420 | 56.3420 | 56.3460 | 56.3650 | 56.3420 | 56.3420 | 56.3650 |
| 24 | 0.0050 | 56.6890 | 56.6890 | 56.6920 | 56.7120 | 56.6890 | 56.6880 | 56.7130 |

| | 8 | 9 | ... | 1991 | 1992 | 1993 | 1994 | 1995 | \ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 9.574 | 9.5794 | ... | 9.3592 | 9.3455 | 9.3448 | 9.3599 | 9.3448 | |
| 1 | 19.147 | 19.1570 | ... | 18.7160 | 18.6890 | 18.6870 | 18.7180 | 18.6870 | |
| 2 | 28.624 | 28.6290 | ... | 27.9810 | 27.9480 | 27.9430 | 27.9890 | 27.9460 | |
| 3 | 36.484 | 36.4960 | ... | 36.3190 | 36.2980 | 36.2830 | 36.3210 | 36.2650 | |
| 4 | 42.311 | 42.3240 | ... | 42.7930 | 42.7910 | 42.7830 | 42.8220 | 42.7810 | |
| 5 | 46.384 | 46.4020 | ... | 47.0830 | 47.0960 | 47.1010 | 47.1350 | 47.1280 | |
| 6 | 48.957 | 48.9610 | ... | 49.5820 | 49.5740 | 49.5920 | 49.6160 | 49.5910 | |
| 7 | 50.115 | 50.1090 | ... | 50.4820 | 50.4630 | 50.4740 | 50.4980 | 50.4520 | |
| 8 | 50.658 | 50.6580 | ... | 51.0360 | 51.0110 | 51.0220 | 51.0530 | 50.9940 | |
| 9 | 51.115 | 51.1160 | ... | 51.5230 | 51.4920 | 51.5050 | 51.5410 | 51.4720 | |
| 10 | 51.541 | 51.5420 | ... | 51.9770 | 51.9430 | 51.9560 | 51.9970 | 51.9190 | |
| 11 | 51.949 | 51.9510 | ... | 52.4120 | 52.3740 | 52.3880 | 52.4330 | 52.3470 | |
| 12 | 52.346 | 52.3470 | ... | 52.8310 | 52.7910 | 52.8060 | 52.8540 | 52.7620 | |
| 13 | 52.733 | 52.7350 | ... | 53.2380 | 53.1970 | 53.2120 | 53.2630 | 53.1650 | |
| 14 | 53.113 | 53.1160 | ... | 53.6280 | 53.5910 | 53.6050 | 53.6590 | 53.5590 | |
| 15 | 53.487 | 53.4900 | ... | 54.0090 | 53.9720 | 53.9860 | 54.0420 | 53.9440 | |
| 16 | 53.857 | 53.8600 | ... | 54.3830 | 54.3450 | 54.3580 | 54.4160 | 54.3210 | |
| 17 | 54.222 | 54.2260 | ... | 54.7500 | 54.7100 | 54.7240 | 54.7840 | 54.6870 | |
| 18 | 54.584 | 54.5880 | ... | 55.1120 | 55.0710 | 55.0840 | 55.1460 | 55.0460 | |
| 19 | 54.943 | 54.9470 | ... | 55.4690 | 55.4260 | 55.4400 | 55.5020 | 55.4010 | |
| 20 | 55.299 | 55.3030 | ... | 55.8210 | 55.7770 | 55.7910 | 55.8550 | 55.7510 | |
| 21 | 55.652 | 55.6560 | ... | 56.1700 | 56.1250 | 56.1390 | 56.2020 | 56.0980 | |
| 22 | 56.003 | 56.0070 | ... | 56.5150 | 56.4690 | 56.4830 | 56.5460 | 56.4410 | |
| 23 | 56.352 | 56.3570 | ... | 56.8560 | 56.8100 | 56.8240 | 56.8850 | 56.7810 | |
| 24 | 56.698 | 56.7040 | ... | 57.1950 | 57.1480 | 57.1620 | 57.2210 | 57.1180 | |

| | 1996 | 1997 | 1998 | 1999 | 2000 |
|---|---|---|---|---|---|
| 0 | 9.3469 | 9.3494 | 9.3492 | 9.3543 | 9.3494 |
| 1 | 18.6920 | 18.6970 | 18.6960 | 18.7070 | 18.6960 |
| 2 | 27.9480 | 27.9600 | 27.9570 | 27.9640 | 27.9330 |
| 3 | 36.2500 | 36.2610 | 36.2410 | 36.2270 | 36.2080 |
| 4 | 42.7670 | 42.7790 | 42.7630 | 42.7350 | 42.7320 |
| 5 | 47.1040 | 47.1140 | 47.1230 | 47.0790 | 47.0940 |
| 6 | 49.5890 | 49.5770 | 49.5820 | 49.5630 | 49.5500 |
| 7 | 50.4550 | 50.4260 | 50.4110 | 50.3980 | 50.3570 |
| 8 | 50.9990 | 50.9660 | 50.9450 | 50.9340 | 50.8820 |
| 9 | 51.4790 | 51.4440 | 51.4190 | 51.4080 | 51.3490 |
| 10 | 51.9280 | 51.8910 | 51.8620 | 51.8520 | 51.7870 |
| 11 | 52.3580 | 52.3180 | 52.2860 | 52.2780 | 52.2070 |

```
12  52.7740  52.7320  52.6980  52.6900  52.6140
13  53.1780  53.1340  53.0980  53.0910  53.0100
14  53.5730  53.5270  53.4890  53.4830  53.3980
15  53.9570  53.9110  53.8720  53.8610  53.7780
16  54.3290  54.2820  54.2480  54.2300  54.1510
17  54.6940  54.6460  54.6130  54.5920  54.5130
18  55.0530  55.0040  54.9710  54.9490  54.8690
19  55.4080  55.3570  55.3230  55.3020  55.2200
20  55.7590  55.7070  55.6720  55.6500  55.5670
21  56.1050  56.0520  56.0170  55.9950  55.9110
22  56.4490  56.3950  56.3590  56.3370  56.2510
23  56.7890  56.7340  56.6970  56.6760  56.5890
24  57.1260  57.0700  57.0330  57.0130  56.9240

[25 rows x 2001 columns]
```

[35]:
```
X = np.array(X)
X = X.reshape(2000, 23)
X_input = X[0:2000]
print(X_input.shape)
print(X_input[1999])
```

```
(2000, 23)
[159.         211.          21.93294387  64.1054526   32.91902306
 115.42983722  25.36049823 165.61869264  76.12923234 181.95859096
  25.82235632  24.38902307  66.00332628  93.28627211  85.11467013
 126.49187173 133.39110318 178.73104273 125.60288691  26.0349274
 120.80692658  80.13866749 132.22169038]
```

[36]:
```
Y = []
for col in y.columns:
    Y.append(y[col])
Y_input = np.array(Y[1:2001])
print(Y_input[1999])
print(Y_input.shape)
```

```
[ 9.3494 18.696  27.933  36.208  42.732  47.094  49.55   50.357  50.882
 51.349  51.787  52.207  52.614  53.01   53.398  53.778  54.151  54.513
 54.869  55.22   55.567  55.911  56.251  56.589  56.924 ]
(2000, 25)
```

[37]:
```
# X_train = X_input[0:600]
# X_test = X_input[600:750]
# X_val = X_input[750:900]
# X_verify = X_input[900:1000]
```

[38]:
```
# y_train = Y_input[0:600]
# y_test = Y_input[600:750]
```

```
# y_val = Y_input[750:900]
# y_verify = Y_input[900:1000]
```

[39]:
```
X_train, X_test, y_train, y_test = train_test_split(X_input, Y_input,␣
↪test_size=0.05, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
↪026, random_state=42)
X_train, X_verify, y_train, y_verify = train_test_split(X_train, y_train,␣
↪test_size=0.027, random_state=42)
print(len(X_train))
print(len(X_test))
print(len(X_val))
print(len(X_verify))
```

```
1800
100
50
50
```

[40]:
```
from tensorflow.keras.models import Sequential

model = keras.Sequential([
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(25)
])
model.build(input_shape=(1, 23))
model.compile(
                loss=keras.losses.MeanSquaredError(),
                optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
                metrics=["accuracy", "mean_squared_error"],
)
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_11 (Dense)            (1, 256)                  6144
```

```
 dense_12 (Dense)            (1, 256)                  65792

 dense_13 (Dense)            (1, 256)                  65792

 dense_14 (Dense)            (1, 256)                  65792

 dense_15 (Dense)            (1, 256)                  65792

 dense_16 (Dense)            (1, 256)                  65792

 dense_17 (Dense)            (1, 256)                  65792

 dense_18 (Dense)            (1, 256)                  65792

 dense_19 (Dense)            (1, 256)                  65792

 dense_20 (Dense)            (1, 256)                  65792

 dense_21 (Dense)            (1, 25)                   6425

=================================================================
Total params: 604,697
Trainable params: 604,697
Non-trainable params: 0
_________________________________________________________________
```

[41]:
```
with tf.device('/gpu:1'):
    model.fit(X_train, y_train, batch_size=20, epochs=100, verbose=2,␣
 ↪validation_data=(X_val, y_val))
```

```
Epoch 1/100
90/90 - 1s - loss: 496.3734 - accuracy: 0.3150 - mean_squared_error: 496.3734 -
val_loss: 2.1514 - val_accuracy: 1.0000 - val_mean_squared_error: 2.1514 -
1s/epoch - 13ms/step
Epoch 2/100
90/90 - 0s - loss: 1.0703 - accuracy: 0.9961 - mean_squared_error: 1.0703 -
val_loss: 0.8460 - val_accuracy: 1.0000 - val_mean_squared_error: 0.8460 -
218ms/epoch - 2ms/step
Epoch 3/100
90/90 - 0s - loss: 0.5651 - accuracy: 1.0000 - mean_squared_error: 0.5651 -
val_loss: 0.5219 - val_accuracy: 1.0000 - val_mean_squared_error: 0.5219 -
206ms/epoch - 2ms/step
Epoch 4/100
90/90 - 0s - loss: 0.4384 - accuracy: 0.9994 - mean_squared_error: 0.4384 -
val_loss: 0.3969 - val_accuracy: 1.0000 - val_mean_squared_error: 0.3969 -
191ms/epoch - 2ms/step
Epoch 5/100
90/90 - 0s - loss: 0.3595 - accuracy: 1.0000 - mean_squared_error: 0.3595 -
```

```
val_loss: 0.3696 - val_accuracy: 1.0000 - val_mean_squared_error: 0.3696 -
213ms/epoch - 2ms/step
Epoch 6/100
90/90 - 0s - loss: 0.3363 - accuracy: 0.9994 - mean_squared_error: 0.3363 -
val_loss: 0.4013 - val_accuracy: 1.0000 - val_mean_squared_error: 0.4013 -
200ms/epoch - 2ms/step
Epoch 7/100
90/90 - 0s - loss: 0.3297 - accuracy: 0.9989 - mean_squared_error: 0.3297 -
val_loss: 0.2890 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2890 -
197ms/epoch - 2ms/step
Epoch 8/100
90/90 - 0s - loss: 0.3120 - accuracy: 1.0000 - mean_squared_error: 0.3120 -
val_loss: 0.2956 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2956 -
200ms/epoch - 2ms/step
Epoch 9/100
90/90 - 0s - loss: 0.2721 - accuracy: 1.0000 - mean_squared_error: 0.2721 -
val_loss: 0.2401 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2401 -
202ms/epoch - 2ms/step
Epoch 10/100
90/90 - 0s - loss: 0.2796 - accuracy: 0.9928 - mean_squared_error: 0.2796 -
val_loss: 0.3151 - val_accuracy: 1.0000 - val_mean_squared_error: 0.3151 -
268ms/epoch - 3ms/step
Epoch 11/100
90/90 - 0s - loss: 0.2513 - accuracy: 1.0000 - mean_squared_error: 0.2513 -
val_loss: 0.2728 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2728 -
250ms/epoch - 3ms/step
Epoch 12/100
90/90 - 0s - loss: 0.2267 - accuracy: 1.0000 - mean_squared_error: 0.2267 -
val_loss: 0.2004 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2004 -
185ms/epoch - 2ms/step
Epoch 13/100
90/90 - 0s - loss: 0.2473 - accuracy: 1.0000 - mean_squared_error: 0.2473 -
val_loss: 0.1862 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1862 -
199ms/epoch - 2ms/step
Epoch 14/100
90/90 - 0s - loss: 0.2017 - accuracy: 1.0000 - mean_squared_error: 0.2017 -
val_loss: 0.2314 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2314 -
193ms/epoch - 2ms/step
Epoch 15/100
90/90 - 0s - loss: 0.2103 - accuracy: 1.0000 - mean_squared_error: 0.2103 -
val_loss: 0.1675 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1675 -
191ms/epoch - 2ms/step
Epoch 16/100
90/90 - 0s - loss: 0.2050 - accuracy: 1.0000 - mean_squared_error: 0.2050 -
val_loss: 0.1554 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1554 -
219ms/epoch - 2ms/step
Epoch 17/100
90/90 - 0s - loss: 0.1931 - accuracy: 0.9994 - mean_squared_error: 0.1931 -
```

```
val_loss: 0.1729 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1729 -
190ms/epoch - 2ms/step
Epoch 18/100
90/90 - 0s - loss: 0.2058 - accuracy: 0.9989 - mean_squared_error: 0.2058 -
val_loss: 0.1558 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1558 -
206ms/epoch - 2ms/step
Epoch 19/100
90/90 - 0s - loss: 0.1807 - accuracy: 1.0000 - mean_squared_error: 0.1807 -
val_loss: 0.1903 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1903 -
194ms/epoch - 2ms/step
Epoch 20/100
90/90 - 0s - loss: 0.1647 - accuracy: 1.0000 - mean_squared_error: 0.1647 -
val_loss: 0.1363 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1363 -
180ms/epoch - 2ms/step
Epoch 21/100
90/90 - 0s - loss: 0.1523 - accuracy: 1.0000 - mean_squared_error: 0.1523 -
val_loss: 0.1286 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1286 -
196ms/epoch - 2ms/step
Epoch 22/100
90/90 - 0s - loss: 0.1376 - accuracy: 1.0000 - mean_squared_error: 0.1376 -
val_loss: 0.1449 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1449 -
193ms/epoch - 2ms/step
Epoch 23/100
90/90 - 0s - loss: 0.1366 - accuracy: 1.0000 - mean_squared_error: 0.1366 -
val_loss: 0.1269 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1269 -
192ms/epoch - 2ms/step
Epoch 24/100
90/90 - 0s - loss: 0.1380 - accuracy: 1.0000 - mean_squared_error: 0.1380 -
val_loss: 0.2275 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2275 -
200ms/epoch - 2ms/step
Epoch 25/100
90/90 - 0s - loss: 0.1599 - accuracy: 1.0000 - mean_squared_error: 0.1599 -
val_loss: 0.1607 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1607 -
196ms/epoch - 2ms/step
Epoch 26/100
90/90 - 0s - loss: 0.1654 - accuracy: 0.9989 - mean_squared_error: 0.1654 -
val_loss: 0.1102 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1102 -
198ms/epoch - 2ms/step
Epoch 27/100
90/90 - 0s - loss: 0.1333 - accuracy: 0.9994 - mean_squared_error: 0.1333 -
val_loss: 0.1089 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1089 -
208ms/epoch - 2ms/step
Epoch 28/100
90/90 - 0s - loss: 0.1346 - accuracy: 1.0000 - mean_squared_error: 0.1346 -
val_loss: 0.1712 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1712 -
216ms/epoch - 2ms/step
Epoch 29/100
90/90 - 0s - loss: 0.1322 - accuracy: 1.0000 - mean_squared_error: 0.1322 -
```

```
val_loss: 0.1121 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1121 -
215ms/epoch - 2ms/step
Epoch 30/100
90/90 - 0s - loss: 0.1345 - accuracy: 0.9983 - mean_squared_error: 0.1345 -
val_loss: 0.2532 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2532 -
230ms/epoch - 3ms/step
Epoch 31/100
90/90 - 0s - loss: 0.1178 - accuracy: 1.0000 - mean_squared_error: 0.1178 -
val_loss: 0.1089 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1089 -
325ms/epoch - 4ms/step
Epoch 32/100
90/90 - 0s - loss: 0.1589 - accuracy: 0.9994 - mean_squared_error: 0.1589 -
val_loss: 0.1319 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1319 -
274ms/epoch - 3ms/step
Epoch 33/100
90/90 - 0s - loss: 0.1261 - accuracy: 1.0000 - mean_squared_error: 0.1261 -
val_loss: 0.1411 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1411 -
265ms/epoch - 3ms/step
Epoch 34/100
90/90 - 0s - loss: 0.1497 - accuracy: 0.9967 - mean_squared_error: 0.1497 -
val_loss: 0.0922 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0922 -
211ms/epoch - 2ms/step
Epoch 35/100
90/90 - 0s - loss: 0.1094 - accuracy: 1.0000 - mean_squared_error: 0.1094 -
val_loss: 0.2863 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2863 -
215ms/epoch - 2ms/step
Epoch 36/100
90/90 - 0s - loss: 0.1170 - accuracy: 0.9994 - mean_squared_error: 0.1170 -
val_loss: 0.1038 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1038 -
208ms/epoch - 2ms/step
Epoch 37/100
90/90 - 0s - loss: 0.1037 - accuracy: 0.9978 - mean_squared_error: 0.1037 -
val_loss: 0.0911 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0911 -
273ms/epoch - 3ms/step
Epoch 38/100
90/90 - 0s - loss: 0.0975 - accuracy: 1.0000 - mean_squared_error: 0.0975 -
val_loss: 0.0820 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0820 -
239ms/epoch - 3ms/step
Epoch 39/100
90/90 - 0s - loss: 0.1183 - accuracy: 1.0000 - mean_squared_error: 0.1183 -
val_loss: 0.0928 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0928 -
224ms/epoch - 2ms/step
Epoch 40/100
90/90 - 0s - loss: 0.1169 - accuracy: 1.0000 - mean_squared_error: 0.1169 -
val_loss: 0.3634 - val_accuracy: 1.0000 - val_mean_squared_error: 0.3634 -
220ms/epoch - 2ms/step
Epoch 41/100
90/90 - 0s - loss: 0.1030 - accuracy: 1.0000 - mean_squared_error: 0.1030 -
```

```
val_loss: 0.1478 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1478 -
228ms/epoch - 3ms/step
Epoch 42/100
90/90 - 0s - loss: 0.0963 - accuracy: 1.0000 - mean_squared_error: 0.0963 -
val_loss: 0.0876 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0876 -
205ms/epoch - 2ms/step
Epoch 43/100
90/90 - 0s - loss: 0.0819 - accuracy: 1.0000 - mean_squared_error: 0.0819 -
val_loss: 0.0736 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0736 -
248ms/epoch - 3ms/step
Epoch 44/100
90/90 - 0s - loss: 0.1188 - accuracy: 0.9961 - mean_squared_error: 0.1188 -
val_loss: 0.2169 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2169 -
233ms/epoch - 3ms/step
Epoch 45/100
90/90 - 0s - loss: 0.1070 - accuracy: 1.0000 - mean_squared_error: 0.1070 -
val_loss: 0.0767 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0767 -
208ms/epoch - 2ms/step
Epoch 46/100
90/90 - 0s - loss: 0.1168 - accuracy: 1.0000 - mean_squared_error: 0.1168 -
val_loss: 0.0772 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0772 -
202ms/epoch - 2ms/step
Epoch 47/100
90/90 - 0s - loss: 0.0758 - accuracy: 1.0000 - mean_squared_error: 0.0758 -
val_loss: 0.0807 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0807 -
192ms/epoch - 2ms/step
Epoch 48/100
90/90 - 0s - loss: 0.1279 - accuracy: 1.0000 - mean_squared_error: 0.1279 -
val_loss: 0.0793 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0793 -
189ms/epoch - 2ms/step
Epoch 49/100
90/90 - 0s - loss: 0.0985 - accuracy: 1.0000 - mean_squared_error: 0.0985 -
val_loss: 0.0621 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0621 -
203ms/epoch - 2ms/step
Epoch 50/100
90/90 - 0s - loss: 0.0758 - accuracy: 1.0000 - mean_squared_error: 0.0758 -
val_loss: 0.0641 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0641 -
195ms/epoch - 2ms/step
Epoch 51/100
90/90 - 0s - loss: 0.0820 - accuracy: 1.0000 - mean_squared_error: 0.0820 -
val_loss: 0.0652 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0652 -
201ms/epoch - 2ms/step
Epoch 52/100
90/90 - 0s - loss: 0.0951 - accuracy: 1.0000 - mean_squared_error: 0.0951 -
val_loss: 0.1917 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1917 -
197ms/epoch - 2ms/step
Epoch 53/100
90/90 - 0s - loss: 0.1195 - accuracy: 1.0000 - mean_squared_error: 0.1195 -
```

```
val_loss: 0.0913 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0913 -
224ms/epoch - 2ms/step
Epoch 54/100
90/90 - 0s - loss: 0.0970 - accuracy: 0.9989 - mean_squared_error: 0.0970 -
val_loss: 0.0777 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0777 -
211ms/epoch - 2ms/step
Epoch 55/100
90/90 - 0s - loss: 0.1007 - accuracy: 1.0000 - mean_squared_error: 0.1007 -
val_loss: 0.1136 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1136 -
232ms/epoch - 3ms/step
Epoch 56/100
90/90 - 0s - loss: 0.0947 - accuracy: 1.0000 - mean_squared_error: 0.0947 -
val_loss: 0.0921 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0921 -
205ms/epoch - 2ms/step
Epoch 57/100
90/90 - 0s - loss: 0.1231 - accuracy: 1.0000 - mean_squared_error: 0.1231 -
val_loss: 0.0599 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0599 -
212ms/epoch - 2ms/step
Epoch 58/100
90/90 - 0s - loss: 0.0845 - accuracy: 1.0000 - mean_squared_error: 0.0845 -
val_loss: 0.0736 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0736 -
263ms/epoch - 3ms/step
Epoch 59/100
90/90 - 0s - loss: 0.0746 - accuracy: 1.0000 - mean_squared_error: 0.0746 -
val_loss: 0.0509 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0509 -
258ms/epoch - 3ms/step
Epoch 60/100
90/90 - 0s - loss: 0.0841 - accuracy: 1.0000 - mean_squared_error: 0.0841 -
val_loss: 0.1110 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1110 -
255ms/epoch - 3ms/step
Epoch 61/100
90/90 - 0s - loss: 0.0937 - accuracy: 1.0000 - mean_squared_error: 0.0937 -
val_loss: 0.0522 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0522 -
208ms/epoch - 2ms/step
Epoch 62/100
90/90 - 0s - loss: 0.0627 - accuracy: 1.0000 - mean_squared_error: 0.0627 -
val_loss: 0.0504 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0504 -
193ms/epoch - 2ms/step
Epoch 63/100
90/90 - 0s - loss: 0.0851 - accuracy: 1.0000 - mean_squared_error: 0.0851 -
val_loss: 0.2709 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2709 -
204ms/epoch - 2ms/step
Epoch 64/100
90/90 - 0s - loss: 0.0664 - accuracy: 1.0000 - mean_squared_error: 0.0664 -
val_loss: 0.0674 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0674 -
179ms/epoch - 2ms/step
Epoch 65/100
90/90 - 0s - loss: 0.0618 - accuracy: 1.0000 - mean_squared_error: 0.0618 -
```

```
val_loss: 0.0478 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0478 -
239ms/epoch - 3ms/step
Epoch 66/100
90/90 - 0s - loss: 0.0600 - accuracy: 1.0000 - mean_squared_error: 0.0600 -
val_loss: 0.0450 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0450 -
257ms/epoch - 3ms/step
Epoch 67/100
90/90 - 0s - loss: 0.0660 - accuracy: 1.0000 - mean_squared_error: 0.0660 -
val_loss: 0.0443 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0443 -
242ms/epoch - 3ms/step
Epoch 68/100
90/90 - 0s - loss: 0.0839 - accuracy: 1.0000 - mean_squared_error: 0.0839 -
val_loss: 0.1833 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1833 -
178ms/epoch - 2ms/step
Epoch 69/100
90/90 - 0s - loss: 0.0855 - accuracy: 1.0000 - mean_squared_error: 0.0855 -
val_loss: 0.0531 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0531 -
174ms/epoch - 2ms/step
Epoch 70/100
90/90 - 0s - loss: 0.0673 - accuracy: 1.0000 - mean_squared_error: 0.0673 -
val_loss: 0.1020 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1020 -
183ms/epoch - 2ms/step
Epoch 71/100
90/90 - 0s - loss: 0.0631 - accuracy: 1.0000 - mean_squared_error: 0.0631 -
val_loss: 0.0397 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0397 -
181ms/epoch - 2ms/step
Epoch 72/100
90/90 - 0s - loss: 0.0835 - accuracy: 0.9994 - mean_squared_error: 0.0835 -
val_loss: 0.0532 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0532 -
176ms/epoch - 2ms/step
Epoch 73/100
90/90 - 0s - loss: 0.1140 - accuracy: 1.0000 - mean_squared_error: 0.1140 -
val_loss: 0.0958 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0958 -
174ms/epoch - 2ms/step
Epoch 74/100
90/90 - 0s - loss: 0.0904 - accuracy: 1.0000 - mean_squared_error: 0.0904 -
val_loss: 0.0963 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0963 -
194ms/epoch - 2ms/step
Epoch 75/100
90/90 - 0s - loss: 0.0869 - accuracy: 0.9972 - mean_squared_error: 0.0869 -
val_loss: 0.1130 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1130 -
207ms/epoch - 2ms/step
Epoch 76/100
90/90 - 0s - loss: 0.0988 - accuracy: 1.0000 - mean_squared_error: 0.0988 -
val_loss: 0.0766 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0766 -
238ms/epoch - 3ms/step
Epoch 77/100
90/90 - 0s - loss: 0.0591 - accuracy: 1.0000 - mean_squared_error: 0.0591 -
```

```
val_loss: 0.0412 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0412 -
201ms/epoch - 2ms/step
Epoch 78/100
90/90 - 0s - loss: 0.0690 - accuracy: 1.0000 - mean_squared_error: 0.0690 -
val_loss: 0.0365 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0365 -
177ms/epoch - 2ms/step
Epoch 79/100
90/90 - 0s - loss: 0.0632 - accuracy: 1.0000 - mean_squared_error: 0.0632 -
val_loss: 0.0941 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0941 -
175ms/epoch - 2ms/step
Epoch 80/100
90/90 - 0s - loss: 0.1030 - accuracy: 1.0000 - mean_squared_error: 0.1030 -
val_loss: 0.0395 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0395 -
172ms/epoch - 2ms/step
Epoch 81/100
90/90 - 0s - loss: 0.0568 - accuracy: 1.0000 - mean_squared_error: 0.0568 -
val_loss: 0.0298 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0298 -
201ms/epoch - 2ms/step
Epoch 82/100
90/90 - 0s - loss: 0.0785 - accuracy: 1.0000 - mean_squared_error: 0.0785 -
val_loss: 0.1530 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1530 -
173ms/epoch - 2ms/step
Epoch 83/100
90/90 - 0s - loss: 0.0637 - accuracy: 1.0000 - mean_squared_error: 0.0637 -
val_loss: 0.0687 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0687 -
175ms/epoch - 2ms/step
Epoch 84/100
90/90 - 0s - loss: 0.0705 - accuracy: 1.0000 - mean_squared_error: 0.0705 -
val_loss: 0.0839 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0839 -
175ms/epoch - 2ms/step
Epoch 85/100
90/90 - 0s - loss: 0.0643 - accuracy: 1.0000 - mean_squared_error: 0.0643 -
val_loss: 0.0516 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0516 -
179ms/epoch - 2ms/step
Epoch 86/100
90/90 - 0s - loss: 0.0511 - accuracy: 0.9994 - mean_squared_error: 0.0511 -
val_loss: 0.0321 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0321 -
184ms/epoch - 2ms/step
Epoch 87/100
90/90 - 0s - loss: 0.0597 - accuracy: 1.0000 - mean_squared_error: 0.0597 -
val_loss: 0.0340 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0340 -
188ms/epoch - 2ms/step
Epoch 88/100
90/90 - 0s - loss: 0.0442 - accuracy: 1.0000 - mean_squared_error: 0.0442 -
val_loss: 0.0332 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0332 -
184ms/epoch - 2ms/step
Epoch 89/100
90/90 - 0s - loss: 0.0586 - accuracy: 1.0000 - mean_squared_error: 0.0586 -
```

```
val_loss: 0.0373 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0373 -
202ms/epoch - 2ms/step
Epoch 90/100
90/90 - 0s - loss: 0.0666 - accuracy: 0.9972 - mean_squared_error: 0.0666 -
val_loss: 0.0343 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0343 -
201ms/epoch - 2ms/step
Epoch 91/100
90/90 - 0s - loss: 0.0597 - accuracy: 1.0000 - mean_squared_error: 0.0597 -
val_loss: 0.0386 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0386 -
185ms/epoch - 2ms/step
Epoch 92/100
90/90 - 0s - loss: 0.0685 - accuracy: 1.0000 - mean_squared_error: 0.0685 -
val_loss: 0.0344 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0344 -
183ms/epoch - 2ms/step
Epoch 93/100
90/90 - 0s - loss: 0.0653 - accuracy: 1.0000 - mean_squared_error: 0.0653 -
val_loss: 0.0322 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0322 -
199ms/epoch - 2ms/step
Epoch 94/100
90/90 - 0s - loss: 0.0670 - accuracy: 1.0000 - mean_squared_error: 0.0670 -
val_loss: 0.0506 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0506 -
192ms/epoch - 2ms/step
Epoch 95/100
90/90 - 0s - loss: 0.0490 - accuracy: 1.0000 - mean_squared_error: 0.0490 -
val_loss: 0.0417 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0417 -
183ms/epoch - 2ms/step
Epoch 96/100
90/90 - 0s - loss: 0.0592 - accuracy: 0.9994 - mean_squared_error: 0.0592 -
val_loss: 0.0356 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0356 -
195ms/epoch - 2ms/step
Epoch 97/100
90/90 - 0s - loss: 0.0668 - accuracy: 0.9989 - mean_squared_error: 0.0668 -
val_loss: 0.0400 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0400 -
183ms/epoch - 2ms/step
Epoch 98/100
90/90 - 0s - loss: 0.0535 - accuracy: 1.0000 - mean_squared_error: 0.0535 -
val_loss: 0.0422 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0422 -
197ms/epoch - 2ms/step
Epoch 99/100
90/90 - 0s - loss: 0.0590 - accuracy: 1.0000 - mean_squared_error: 0.0590 -
val_loss: 0.0321 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0321 -
175ms/epoch - 2ms/step
Epoch 100/100
90/90 - 0s - loss: 0.0626 - accuracy: 1.0000 - mean_squared_error: 0.0626 -
val_loss: 0.0383 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0383 -
177ms/epoch - 2ms/step
```

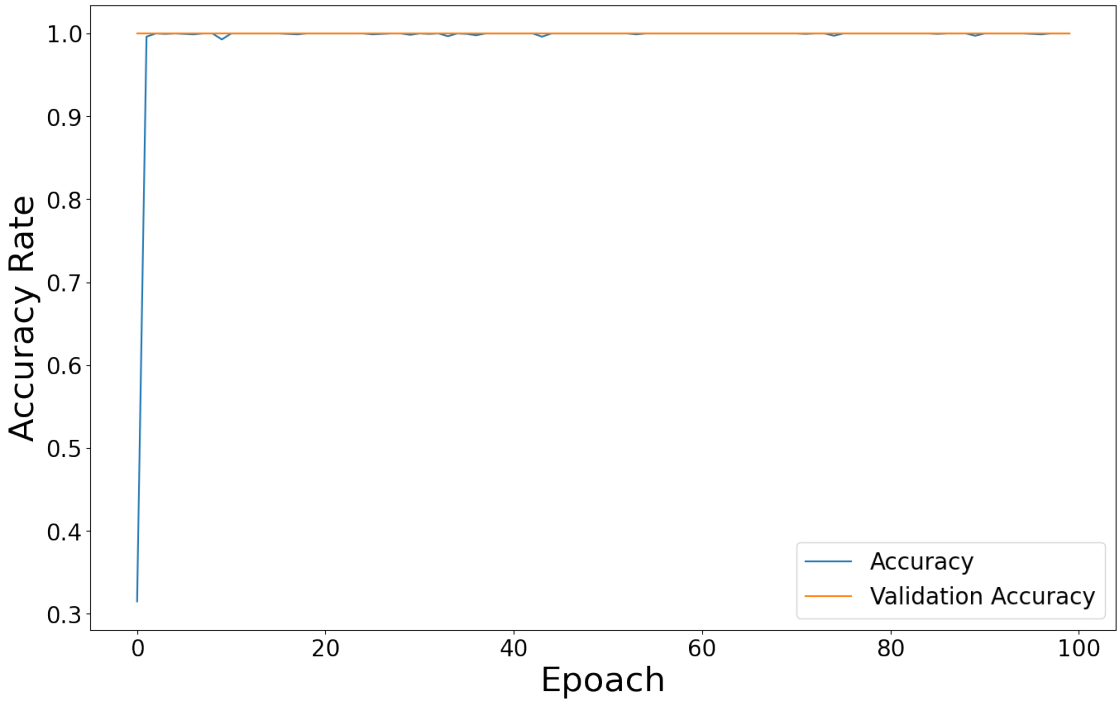
[42]:
```
losses = pd.DataFrame(model.history.history)
```

[43]:
```
losses.tail(1)
```

[43]:
```
        loss  accuracy  mean_squared_error  val_loss  val_accuracy  \
99  0.062633       1.0            0.062633  0.038328           1.0

    val_mean_squared_error
99                0.038328
```

[44]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(losses['accuracy'],label='Accuracy')
plt.plot(losses['val_accuracy'],label='Validation Accuracy')
plt.xlabel('Epoach', fontsize=30)
plt.ylabel('Accuracy Rate', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20)
```

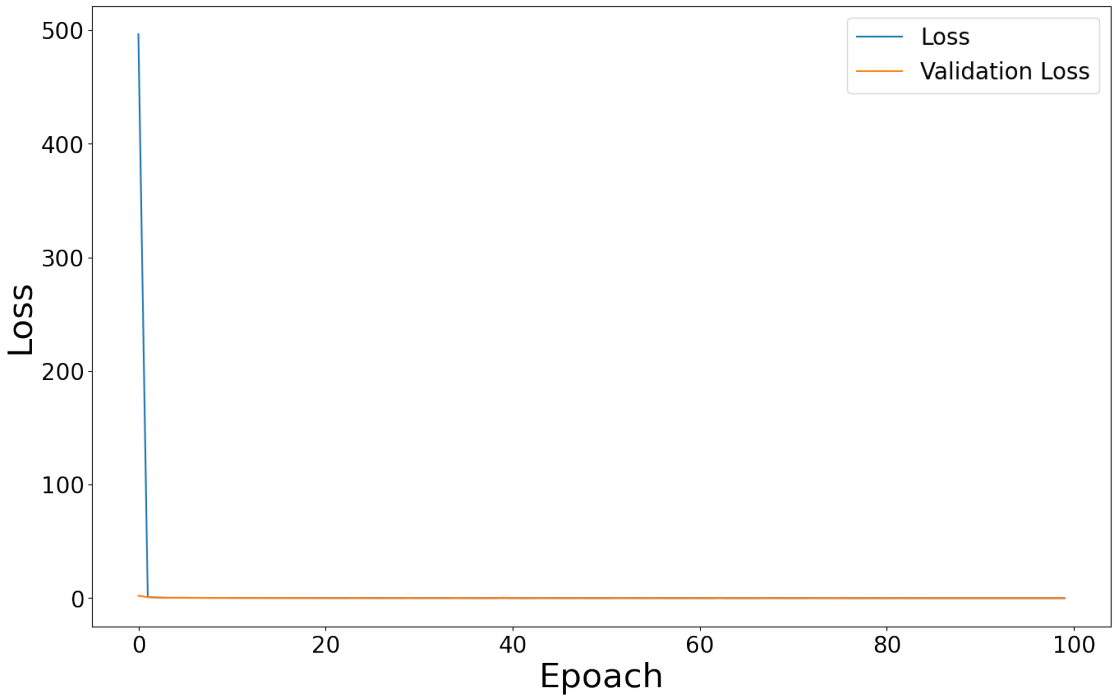
[44]:
```
<matplotlib.legend.Legend at 0x1e5cf401040>
```




[45]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(losses['loss'],label='Loss')
plt.plot(losses['val_loss'],label='Validation Loss')
```

```
plt.xlabel('Epoach', fontsize=30)
plt.ylabel('Loss', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20)
```

[45]: <matplotlib.legend.Legend at 0x1e50c1a3070>




[46]:
```
model.metrics_names
```

[46]: ['loss', 'accuracy', 'mean_squared_error']

[47]:
```
model.evaluate(X_test,y_test)
```

```
4/4 [==============================] - 0s 2ms/step - loss: 0.0387 - accuracy:
1.0000 - mean_squared_error: 0.0387
```

[47]: [0.03873598575592041, 1.0, 0.03873598575592041]

[48]:
```
print(y_verify)
len(y_verify)
```

```
[[ 9.6114 19.222  28.717  ... 57.052  57.413  57.77  ]
 [ 9.5189 19.037  28.472  ... 57.353  57.693  58.031 ]
 [ 9.5293 19.058  28.473  ... 56.518  56.872  57.223 ]
 ...
```

```
 [ 9.5378 19.069  28.376  ... 56.557  56.913  57.266 ]
 [ 9.5156 19.03   28.442  ... 57.081  57.426  57.768 ]
 [ 9.5596 19.118  28.594  ... 56.964  57.316  57.665 ]]
```

[48]:
```
50
```

[49]:
```
predictions = model.predict(X_verify)
predictions
```

```
2/2 [==============================] - 0s 2ms/step
```

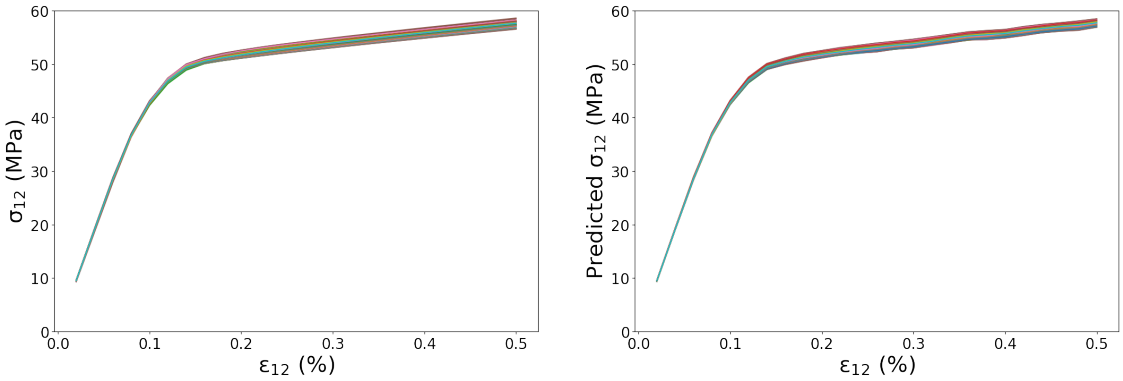
[49]:
```
array([[ 9.519237, 19.191616, 28.694174, ..., 57.131355, 57.379692,
         57.876114],
       [ 9.535583, 19.28199 , 28.766981, ..., 57.575928, 57.849625,
         58.274326],
       [ 9.481114, 19.167328, 28.602041, ..., 56.363445, 56.533577,
         57.112988],
       ...,
       [ 9.436176, 19.043472, 28.430151, ..., 56.55014 , 56.787376,
         57.285316],
       [ 9.455863, 19.109962, 28.487492, ..., 57.123844, 57.431587,
         57.922913],
       [ 9.502147, 19.127972, 28.563267, ..., 57.005848, 57.266273,
         57.728424]], dtype=float32)
```

[50]:
```
x_axis = np.linspace(0.02,0.5,num=25)
y_axis = np.linspace(0,200,10)
print(x_axis)
```

```
[0.02 0.04 0.06 0.08 0.1  0.12 0.14 0.16 0.18 0.2  0.22 0.24 0.26 0.28
 0.3  0.32 0.34 0.36 0.38 0.4  0.42 0.44 0.46 0.48 0.5 ]
```

[51]:
```
plt.figure(figsize=(40,8), dpi=100)
plt.subplot(131)
for i in range(len(y_verify)):
    plt.plot(x_axis, y_verify[i])
plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
plt.ylabel('\u03C3$_{12}$ (MPa)', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.ylim([0, 60])


plt.subplot(132)
for i in range(len(predictions)):
    plt.plot(x_axis, predictions[i])
plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
plt.ylabel('Predicted \u03C3$_{12}$ (MPa)', fontsize=30)
plt.xticks(fontsize=20)
```

```
plt.yticks(fontsize=20)
plt.ylim([0, 60])

plt.show()
```

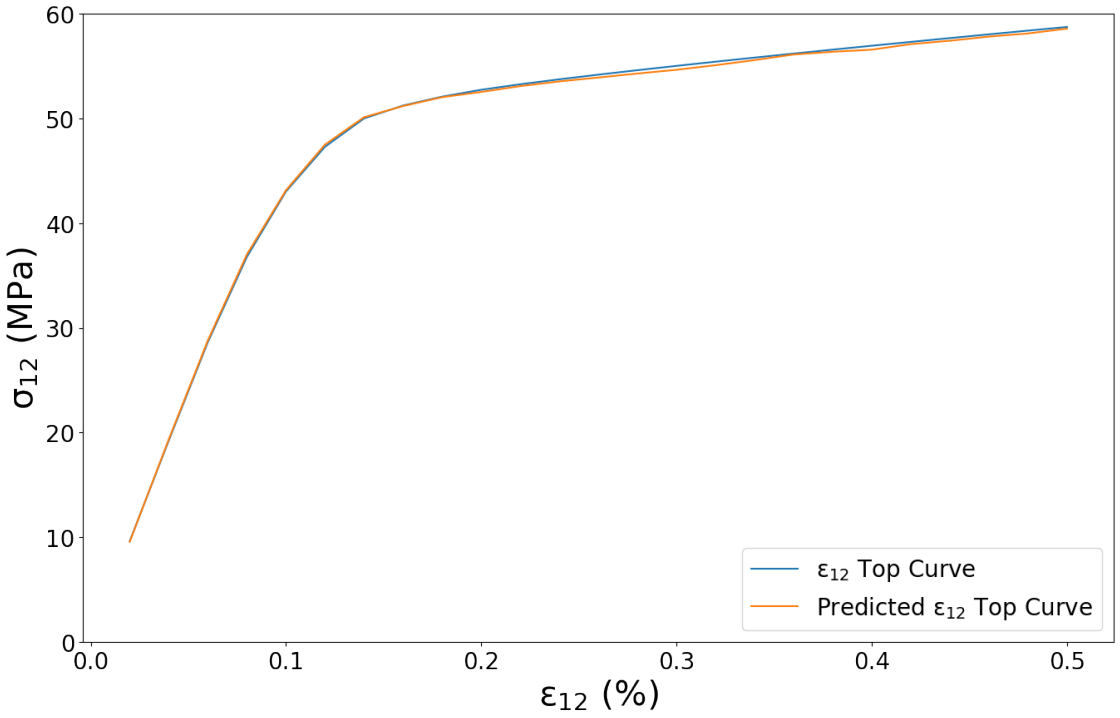



[52]:
```
# for i in range(5):
#     plt.figure(figsize=(16,12), dpi=100)
#
#     plt.plot(x_axis, y_verify[i], label='True Stress Strain')
#     plt.plot(x_axis, predictions[i], label='Predicted Stress Strain')
#     plt.legend(fontsize=20)
#     plt.title("True Curve versus Predicted Curve Plot")
#     plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
#     plt.ylabel('\u03C3$_{12}$ (MPa)', fontsize=30)
#     plt.xticks(fontsize=20)
#     plt.yticks(fontsize=20)
#     plt.ylim([0, 60])
#     plt.show()
```

[53]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(x_axis, Y_input[1379], label= "\u03B5$_{12}$ Top Curve")
plt.plot(x_axis, model.predict(X_input)[1379], label= "Predicted \u03B5$_{12}$␣
↪Top Curve")
plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
plt.ylabel('\u03C3$_{12}$ (MPa)', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20, loc = 'lower right')
plt.ylim([0, 60])
```

```
63/63 [==============================] - 0s 1ms/step
```

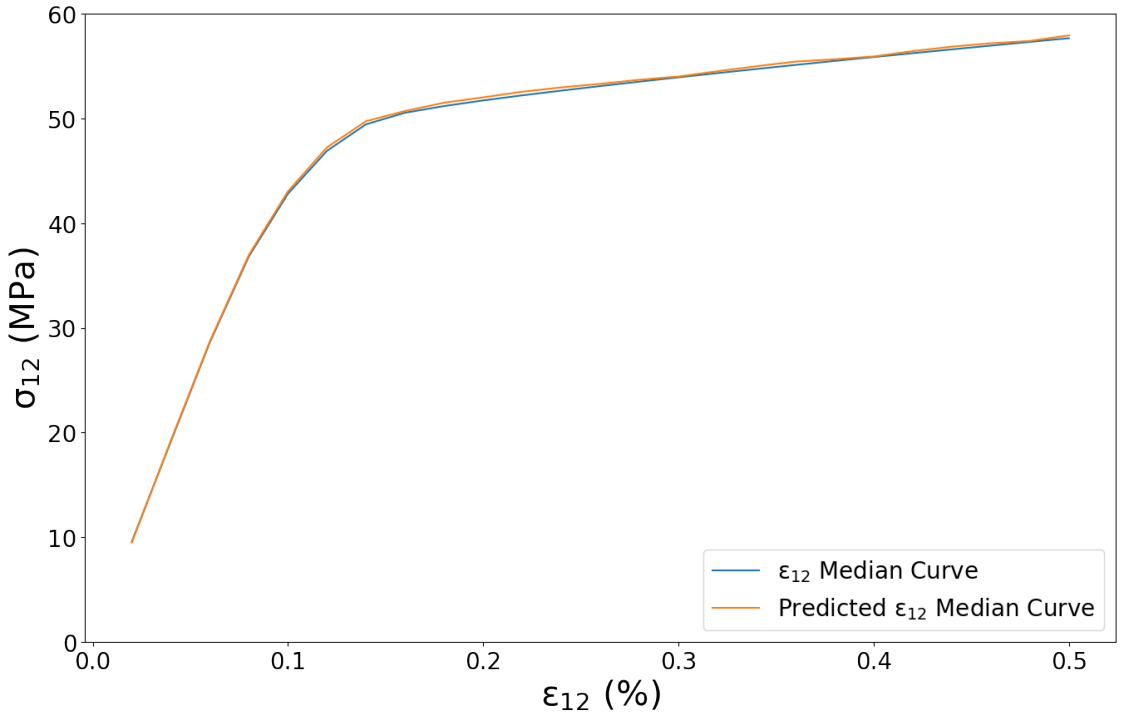
[53]: (0.0, 60.0)




[54]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(x_axis, Y_input[458], label= "\u03B5$_{12}$ Median Curve")
plt.plot(x_axis, model.predict(X_input)[458], label= "Predicted \u03B5$_{12}$␣
 ↪Median Curve")
plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
plt.ylabel('\u03C3$_{12}$ (MPa)', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20, loc = 'lower right')
plt.ylim([0, 60])
```

```
63/63 [==============================] - 0s 1ms/step
```

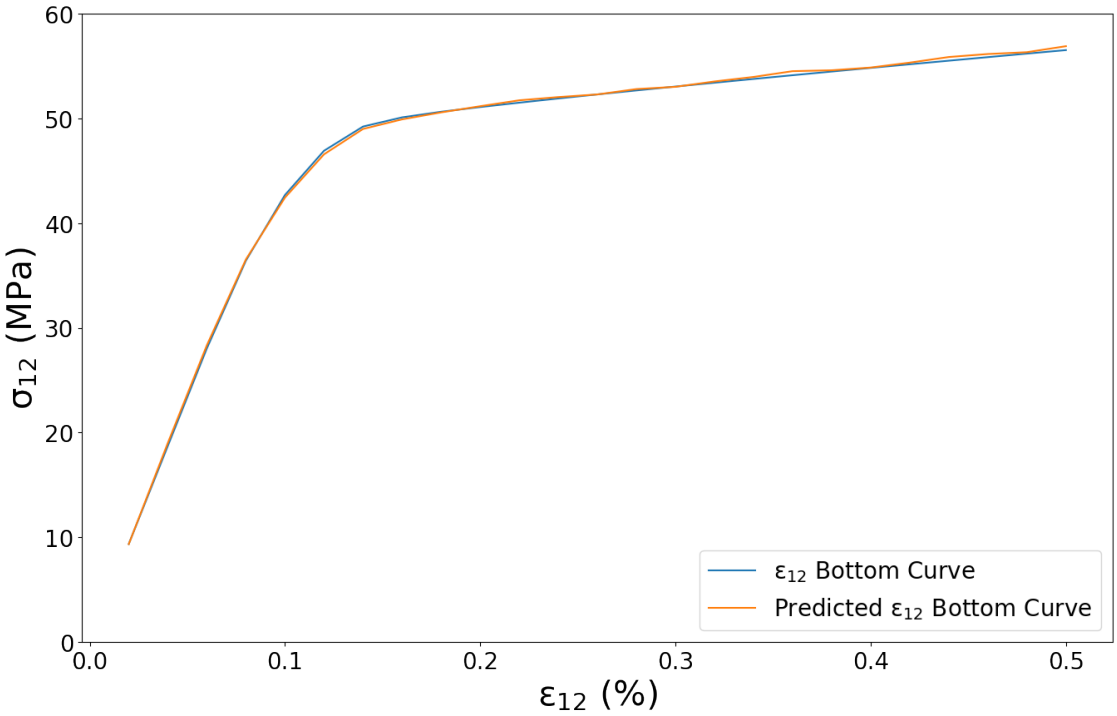
[54]: (0.0, 60.0)




[55]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(x_axis, Y_input[331], label= "\u03B5$_{12}$ Bottom Curve")
plt.plot(x_axis, model.predict(X_input)[331], label= "Predicted \u03B5$_{12}$␣
 ↪Bottom Curve")
plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
plt.ylabel('\u03C3$_{12}$ (MPa)', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20, loc = 'lower right')
plt.ylim([0, 60])
```

```
63/63 [==============================] - 0s 1ms/step
```

[55]: (0.0, 60.0)


60
50
40
30
20
10
0
σ12 (MPa)
0.0
0.1
0.2
0.3
0.4
0.5
ε12 (%)
ε12 Bottom Curve
Predicted ε12 Bottom Curve

# ANN_1200_1600_1800_2000_Homogenized Moduli 10000

October 14, 2022

[8]:
```
# pip install pandas
```

[9]:
```
from keras.models import Sequential
from sklearn import preprocessing
from keras.layers import Dense, Conv1D,Conv2D, Flatten
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import keras
import glob
import os
```

[10]:
```
projectDir = r'C:\Users\18810\Desktop\RUC Collection'
print(projectDir)
```

```
C:\Users\18810\Desktop\RUC Collection
```

[11]:
```
extension = 'csv'
x_filenames = glob.glob(os.path.join(projectDir + '\CSV 20000', '*.csv'))
x_filenames[0]
# im = imageio.imread(x_filenames[0])
```

[11]:
```
'C:\\Users\\18810\\Desktop\\RUC Collection\\CSV 20000\\ann_input0000.csv'
```

[12]:
```
# csv_file = pd.read_csv(r'C:/Users/18810/Desktop/RUC Collection/CSV 20000/
 ↪ann_input'+'0000'+'.csv')
# csv_file = csv_file.drop(csv_file.columns[0], axis=1)
# print(csv_file)
```

[13]:
```
# path = r'C:/Users/18810/Desktop/RUC Collection 0/20000/RUC 20000'
# for fid in range(20000):
#     fileid = f'{fid:04}' # str(fid)
#     # print(fileid)
```

```
#     csv_file = pd.read_csv(r'C:/Users/18810/Desktop/RUC Collection/CSV 20000/
 ↪ann_input'+fileid+'.csv',)
#     X.append(csv_file)
```

[14]:
```
X = []
for i in range(10000):
    file = open(x_filenames[i])
    numpy_array = pd.read_csv(file, delimiter=',',header=None)
    # numpy_array = np. loadtxt(file, delimiter=',')
    X.append(numpy_array)
print(X[0])
```

```
             0
0   159.000000
1   211.000000
2    21.932944
3    26.432944
4    26.432944
5    79.298832
6    26.432944
7   132.164719
8    26.432944
9   185.030607
10   26.432944
11   26.432944
12   79.298832
13   79.298832
14   79.298832
15  132.164719
16   79.298832
17  185.030607
18   79.298832
19   26.432944
20  132.164719
21   79.298832
22  132.164719
```

[15]:
```
X = np.array(X)
X = X.reshape(10000, 23)
X_input = X[0:10000]
print(X_input.shape)
print(X_input[9999])
print(X_input[9999].shape)
X_input = preprocessing.normalize(X_input)
```

```
(10000, 23)
[159.         211.          21.93294387  80.78057355  45.4167103
 138.49454301  92.15068688 114.4322726   94.05319012  77.85554871
```

```
 104.8646826  108.69900865 104.56296643  98.53351403  99.93515595
 138.98474594  35.74237752 175.01704403 124.92134033  42.25672585
  84.57396682  29.24735329 131.75940689]
(23,)
```

[16]:
```
# Homogenized_filenames = glob.glob(os.path.join(projectDir+"\Effective␣
 ↪Stiffness 20000", '*.csv'))
# Homogenized_filenames[0]
```

[17]:
```
Y = []
Homogenized_filenames = glob.glob(os.path.join(projectDir+"\Effective Stiffness␣
 ↪20000", '*.csv'))
for i in range(10000):
    file = open(Homogenized_filenames[i])
    numpy_array = pd.read_csv(file, delimiter=',',header=None, skiprows=1)
    # numpy_array = np.array(numpy_array)
    # numpy_array.reshape(36,1)
    Y.append(numpy_array)
print(Y[0])
```

```
              0            1             2            3           4  \
0  228111.10000   63139.0300   62727.84000    -42.19899      0.0000
1   63139.03000  164169.1000   65857.88000   -230.46350      0.0000
2   62727.84000   65857.8800  161378.80000    -55.89012      0.0000
3     -42.19899    -230.4635     -55.89012  42461.71000      0.0000
4       0.00000       0.0000       0.00000      0.00000  46540.2500
5       0.00000       0.0000       0.00000      0.00000   -251.1407

             5
0       0.0000
1       0.0000
2       0.0000
3       0.0000
4    -251.1407
5   47844.4600
```

[18]:
```
y_input = np.array(Y)
print(type(y_input))
print(Y[0])
# print("*****************************************************************")
```

```
<class 'numpy.ndarray'>
              0            1             2            3           4  \
0  228111.10000   63139.0300   62727.84000    -42.19899      0.0000
1   63139.03000  164169.1000   65857.88000   -230.46350      0.0000
2   62727.84000   65857.8800  161378.80000    -55.89012      0.0000
3     -42.19899    -230.4635     -55.89012  42461.71000      0.0000
4       0.00000       0.0000       0.00000      0.00000  46540.2500
5       0.00000       0.0000       0.00000      0.00000   -251.1407
```

```
            5
0      0.0000
1      0.0000
2      0.0000
3      0.0000
4   -251.1407
5  47844.4600
```

[19]:
```
# index = [0,1,2,3,6,7,8,9,12,13,14,15,18,19,20,21,28,29,34,35]
y_input=y_input.reshape(10000, 36)
print(y_input.shape)
print(y_input[0])
```

```
(10000, 36)
[ 2.281111e+05  6.313903e+04  6.272784e+04 -4.219899e+01  0.000000e+00
  0.000000e+00  6.313903e+04  1.641691e+05  6.585788e+04 -2.304635e+02
  0.000000e+00  0.000000e+00  6.272784e+04  6.585788e+04  1.613788e+05
 -5.589012e+01  0.000000e+00  0.000000e+00 -4.219899e+01 -2.304635e+02
 -5.589012e+01  4.246171e+04  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00  0.000000e+00  4.654025e+04 -2.511407e+02
  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00 -2.511407e+02
  4.784446e+04]
```

[20]:
```
# new = []
# for i in index:
#     new.append(Y_new[0][i])
# print(new)
index = -1
y_clean_index=[]
for i in range(6):
  for j in range(6):
    index += 1
    if i == 0 and j == 0:
      continue
    if i in [0,1,2,3] and j in [4,5]:
      continue
    if j in [0,1,2,3] and i in [4,5]:
      continue
    y_clean_index.append(index)
print(y_clean_index)
y_clean_index = [0, 1, 2, 3, 7, 8, 9, 14, 15, 21, 28, 29, 35]
print(y_clean_index)
```

```
[1, 2, 3, 6, 7, 8, 9, 12, 13, 14, 15, 18, 19, 20, 21, 28, 29, 34, 35]
[0, 1, 2, 3, 7, 8, 9, 14, 15, 21, 28, 29, 35]
```

[21]:
```
# y_input = []
# for i in range(2000):
#     temp = []
#     for j in index:
#         temp.append(Y_new[i][j])
#     y_input.append(temp)
# y_input = np.array(y_input)
# print(y_input[0])
#␣
↪print("**********************************************************************")
# print(y_input[1999])
y_clean=[]

for i in range(10000):
    y_clean.append(y_input[i][y_clean_index])

y_clean=np.array(y_clean)
print(y_clean.shape)
print(y_clean[0])
```

```
(10000, 13)
[ 2.281111e+05  6.313903e+04  6.272784e+04 -4.219899e+01  1.641691e+05
  6.585788e+04 -2.304635e+02  1.613788e+05 -5.589012e+01  4.246171e+04
  4.654025e+04 -2.511407e+02  4.784446e+04]
```

[22]:
```
# names=[[],[],[],[],[],[],[],[],[],[],[],[],[]]
# for i in range(13):
#     names[i]=[]
#     for j in range(2000):
#         names[i].append(y_clean[j][i])
#     names[i]=np.array(names[i])
```

[23]:
```
y_input= y_clean
print(y_input[0])
# print(names[0])
# print(names[0].shape)
```

```
[ 2.281111e+05  6.313903e+04  6.272784e+04 -4.219899e+01  1.641691e+05
  6.585788e+04 -2.304635e+02  1.613788e+05 -5.589012e+01  4.246171e+04
  4.654025e+04 -2.511407e+02  4.784446e+04]
```

[24]:
```
# X_train, X_test, y_train, y_test = train_test_split(X_input, y_input,␣
↪test_size=0.2, random_state=42)
# X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,␣
↪test_size=0.125, random_state=42)
# X_train, X_verify, y_train, y_verify = train_test_split(X_train, y_train,␣
↪test_size=0.1425, random_state=42)
# print(len(X_train))
```

```
# print(len(X_test))
# print(len(X_val))
# print(len(X_verify))
```

[25]:
```
# moduli = ["C11", "C12", "C13", "C14", "C22", "C23", "C24", "C33", "C34",␣
 ↪"C44", "C55", "C56", "C66"]
#
X_train, X_test, y_train, y_test = train_test_split(X_input, y_input,␣
 ↪test_size=0.025, random_state=4)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
 ↪02515, random_state=4)
# with tf.device('/gpu:1'):
#     print("********************Training "+moduli[i]+"********************")
#     model.fit(X_train, y_train, batch_size=20, epochs=100, verbose=2,␣
 ↪validation_data=(X_val, y_val))
```

[28]:
```
model = keras.Sequential([
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(13, activation="relu")
])
# model.add(tf.keras.layers.Reshape((6, 6)))
model.build(input_shape=(1, 23))
model.compile(
                loss=keras.losses.MeanSquaredError(),
                optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
                metrics=["accuracy", "mean_squared_error"],
)
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_12 (Dense)            (1, 256)                  6144

 dense_13 (Dense)            (1, 256)                  65792

 dense_14 (Dense)            (1, 256)                  65792
```

```
 dense_15 (Dense)            (1, 256)                  65792

 dense_16 (Dense)            (1, 256)                  65792

 dense_17 (Dense)            (1, 512)                  131584

 dense_18 (Dense)            (1, 256)                  131328

 dense_19 (Dense)            (1, 256)                  65792

 dense_20 (Dense)            (1, 256)                  65792

 dense_21 (Dense)            (1, 256)                  65792

 dense_22 (Dense)            (1, 256)                  65792

 dense_23 (Dense)            (1, 13)                   3341

=================================================================
Total params: 798,733
Trainable params: 798,733
Non-trainable params: 0
_________________________________________________________________
```

[29]:
```
with tf.device('/gpu:1'):
    model.fit(X_train, y_train, batch_size=10, epochs=25, verbose=2,␣
 ↪validation_data=(X_val, y_val))
```

```
Epoch 1/25
951/951 - 4s - loss: 1319880320.0000 - accuracy: 0.8159 - mean_squared_error:
1319880448.0000 - val_loss: 143188800.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 143188800.0000 - 4s/epoch - 4ms/step
Epoch 2/25
951/951 - 2s - loss: 141736640.0000 - accuracy: 1.0000 - mean_squared_error:
141736624.0000 - val_loss: 141880112.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141880112.0000 - 2s/epoch - 2ms/step
Epoch 3/25
951/951 - 2s - loss: 141458896.0000 - accuracy: 1.0000 - mean_squared_error:
141458896.0000 - val_loss: 142267056.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 142267056.0000 - 2s/epoch - 2ms/step
Epoch 4/25
951/951 - 2s - loss: 141328912.0000 - accuracy: 1.0000 - mean_squared_error:
141328912.0000 - val_loss: 141719408.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141719424.0000 - 2s/epoch - 2ms/step
Epoch 5/25
951/951 - 2s - loss: 141155808.0000 - accuracy: 1.0000 - mean_squared_error:
141155808.0000 - val_loss: 141595568.0000 - val_accuracy: 1.0000 -
```

```
val_mean_squared_error: 141595568.0000 - 2s/epoch - 2ms/step
Epoch 6/25
951/951 - 2s - loss: 141128016.0000 - accuracy: 1.0000 - mean_squared_error:
141128032.0000 - val_loss: 141282112.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141282112.0000 - 2s/epoch - 2ms/step
Epoch 7/25
951/951 - 2s - loss: 141017760.0000 - accuracy: 1.0000 - mean_squared_error:
141017760.0000 - val_loss: 141720864.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141720864.0000 - 2s/epoch - 3ms/step
Epoch 8/25
951/951 - 2s - loss: 140985280.0000 - accuracy: 1.0000 - mean_squared_error:
140985280.0000 - val_loss: 141639088.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141639088.0000 - 2s/epoch - 3ms/step
Epoch 9/25
951/951 - 2s - loss: 140905328.0000 - accuracy: 1.0000 - mean_squared_error:
140905328.0000 - val_loss: 141248048.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141248048.0000 - 2s/epoch - 3ms/step
Epoch 10/25
951/951 - 3s - loss: 140963888.0000 - accuracy: 1.0000 - mean_squared_error:
140963888.0000 - val_loss: 141299248.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141299232.0000 - 3s/epoch - 3ms/step
Epoch 11/25
951/951 - 3s - loss: 140903952.0000 - accuracy: 1.0000 - mean_squared_error:
140903952.0000 - val_loss: 141808448.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141808448.0000 - 3s/epoch - 3ms/step
Epoch 12/25
951/951 - 2s - loss: 140841136.0000 - accuracy: 1.0000 - mean_squared_error:
140841136.0000 - val_loss: 141358272.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141358272.0000 - 2s/epoch - 3ms/step
Epoch 13/25
951/951 - 2s - loss: 140815424.0000 - accuracy: 1.0000 - mean_squared_error:
140815424.0000 - val_loss: 141152624.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141152624.0000 - 2s/epoch - 2ms/step
Epoch 14/25
951/951 - 2s - loss: 140761552.0000 - accuracy: 1.0000 - mean_squared_error:
140761552.0000 - val_loss: 141507248.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141507248.0000 - 2s/epoch - 2ms/step
Epoch 15/25
951/951 - 2s - loss: 140842880.0000 - accuracy: 1.0000 - mean_squared_error:
140842864.0000 - val_loss: 141457776.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141457776.0000 - 2s/epoch - 2ms/step
Epoch 16/25
951/951 - 2s - loss: 140757664.0000 - accuracy: 1.0000 - mean_squared_error:
140757664.0000 - val_loss: 140959808.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 140959808.0000 - 2s/epoch - 2ms/step
Epoch 17/25
951/951 - 2s - loss: 140761040.0000 - accuracy: 1.0000 - mean_squared_error:
140761056.0000 - val_loss: 141585168.0000 - val_accuracy: 1.0000 -
```

```
val_mean_squared_error: 141585168.0000 - 2s/epoch - 2ms/step
Epoch 18/25
951/951 - 2s - loss: 140745648.0000 - accuracy: 1.0000 - mean_squared_error:
140745648.0000 - val_loss: 142254272.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 142254272.0000 - 2s/epoch - 2ms/step
Epoch 19/25
951/951 - 2s - loss: 140833728.0000 - accuracy: 1.0000 - mean_squared_error:
140833728.0000 - val_loss: 140941968.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 140941968.0000 - 2s/epoch - 2ms/step
Epoch 20/25
951/951 - 2s - loss: 140687088.0000 - accuracy: 1.0000 - mean_squared_error:
140687088.0000 - val_loss: 140927008.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 140927008.0000 - 2s/epoch - 2ms/step
Epoch 21/25
951/951 - 2s - loss: 140739440.0000 - accuracy: 1.0000 - mean_squared_error:
140739440.0000 - val_loss: 140981376.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 140981376.0000 - 2s/epoch - 2ms/step
Epoch 22/25
951/951 - 2s - loss: 140650256.0000 - accuracy: 1.0000 - mean_squared_error:
140650240.0000 - val_loss: 141024288.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141024288.0000 - 2s/epoch - 2ms/step
Epoch 23/25
951/951 - 2s - loss: 140633008.0000 - accuracy: 1.0000 - mean_squared_error:
140633008.0000 - val_loss: 141080512.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141080512.0000 - 2s/epoch - 2ms/step
Epoch 24/25
951/951 - 2s - loss: 140692336.0000 - accuracy: 1.0000 - mean_squared_error:
140692336.0000 - val_loss: 140933472.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 140933472.0000 - 2s/epoch - 2ms/step
Epoch 25/25
951/951 - 2s - loss: 140608160.0000 - accuracy: 1.0000 - mean_squared_error:
140608160.0000 - val_loss: 141622640.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 141622640.0000 - 2s/epoch - 2ms/step
```

[30]:
```
losses = pd.DataFrame(model.history.history)
```

[31]:
```
losses.tail(1)
```

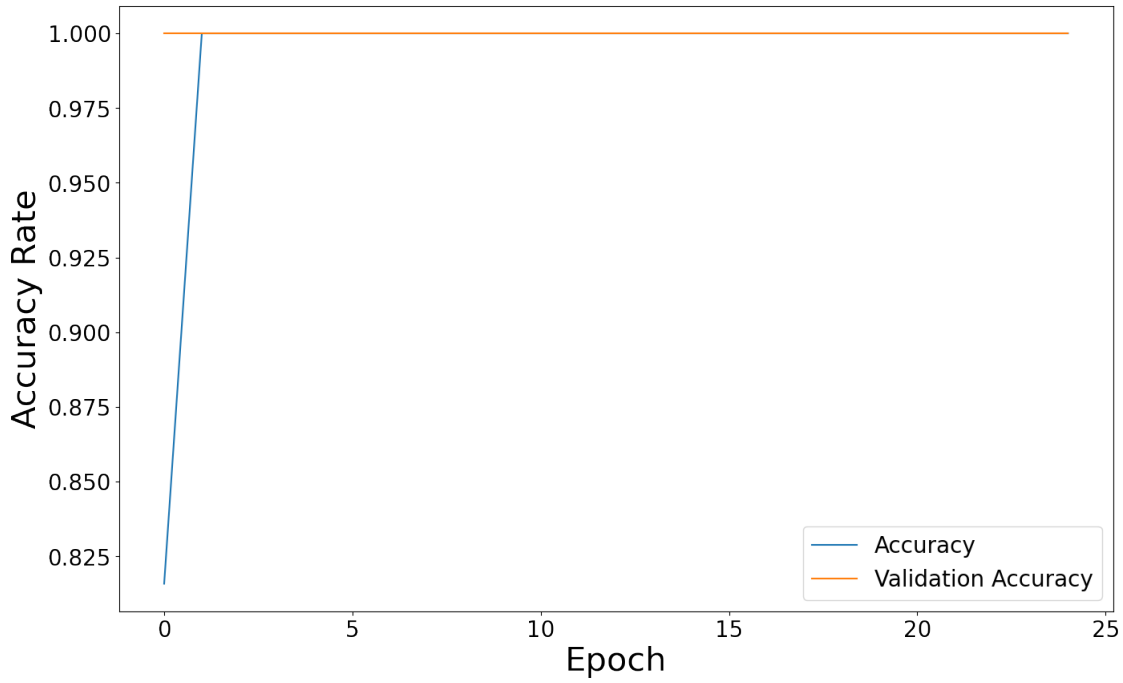
[31]:

| | loss | accuracy | mean_squared_error | val_loss | val_accuracy | val_mean_squared_error |
|---|---|---|---|---|---|---|
| 24 | 140608160.0 | 1.0 | 140608160.0 | 141622640.0 | 1.0 | 141622640.0 |

[32]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(losses['accuracy'],label='Accuracy')
plt.plot(losses['val_accuracy'],label='Validation Accuracy')
plt.xlabel('Epoch', fontsize=30)
```

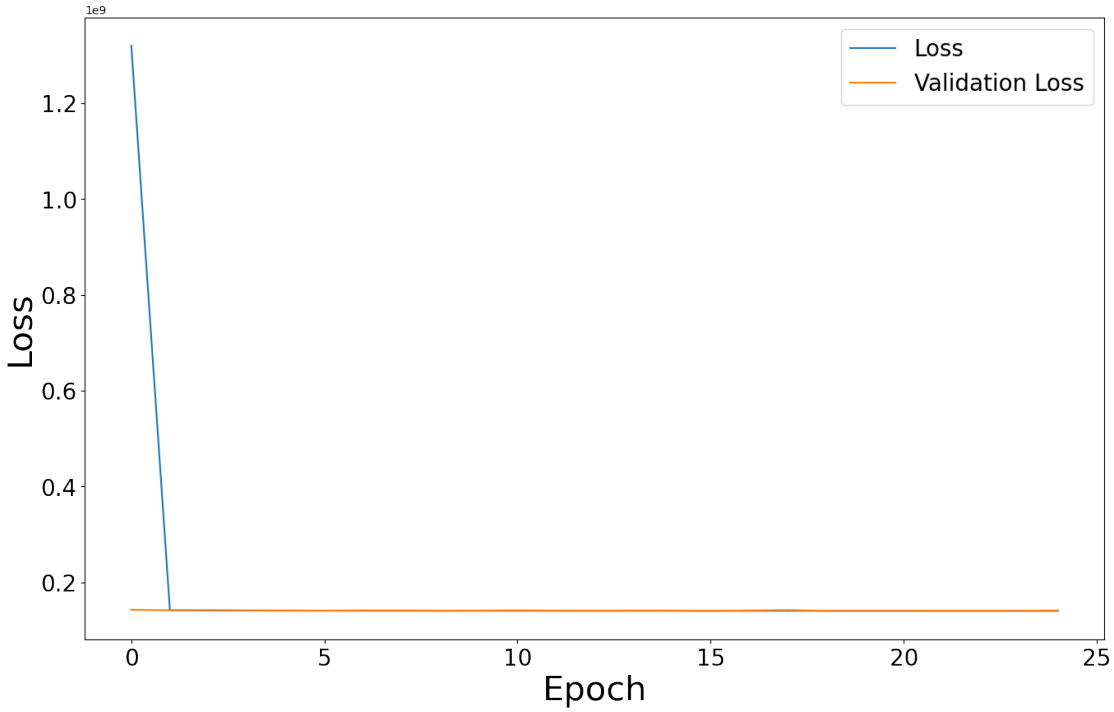
```
plt.ylabel('Accuracy Rate', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20)
```

[32]: <matplotlib.legend.Legend at 0x243bd483ee0>




[33]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(losses['loss'],label='Loss')
plt.plot(losses['val_loss'],label='Validation Loss')
plt.xlabel('Epoch', fontsize=30)
plt.ylabel('Loss', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20)
```

[33]: <matplotlib.legend.Legend at 0x243bd4f5a30>




[34]:
```
model.metrics_names
```

[34]:
```
['loss', 'accuracy', 'mean_squared_error']
```

[35]:
```
model.evaluate(X_test,y_test)
```

```
8/8 [==============================] - 0s 2ms/step - loss: 142058688.0000 -
accuracy: 1.0000 - mean_squared_error: 142058688.0000
```

[35]:
```
[142058688.0, 1.0, 142058688.0]
```

[36]:
```
# print(y_verify[0])
# len(y_verify)
```

[37]:
```
ypred = model.predict(X_test)
ypred.shape
```

```
8/8 [==============================] - 0s 0s/step
```

[37]:
```
(250, 13)
```

[38]:
```
# y_test_prediction = model.predict(X_test)
# print(y_test_prediction.shape)
print(ypred[0][:])
```

```
[216245.06   61269.875  61074.97       0.     155453.36   65635.664
      0.     152842.02       0.          0.      44163.72       0.
  46177.805]
```

[47]:
```
y_test.shape
```

[47]:
```
(250, 13)
```

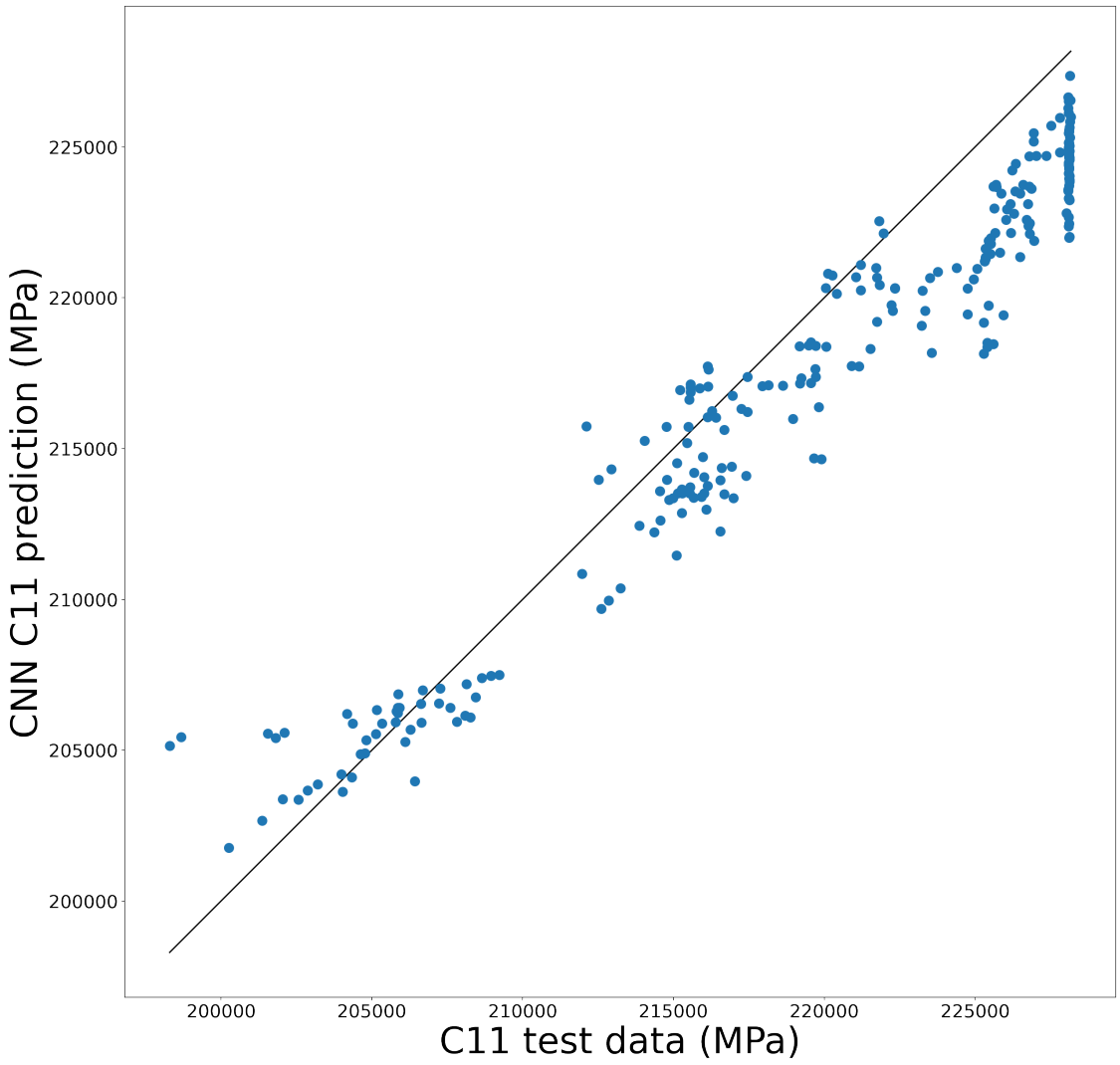
[49]:
```
C_test = [[], [], [], [], [], [], [], [], [], [], [], [], []]
for i in range(13):
    for j in range(y_test.shape[0]):
        C_test[i].append(y_test[j][i])

C_pred = [[], [], [], [], [], [], [], [], [], [], [], [], []]
for i in range(13):
    for j in range(ypred.shape[0]):
        C_pred[i].append(ypred[j][i])

names = ["C11", "C12", "C13", "C14", "C22", "C23", "C24", "C33", "C34", "C44",␣
 ↪"C55", "C56", "C66"]

# for i in range(13):
#     # plt.figure(figsize=(10, 10), dpi=100)
#     # plt.scatter(C_test[i], C_pred[i], s=200, marker=".", linewidths=0.1,␣
 ↪alpha=0.8)
#     fig, ax = plt.subplots(figsize=(20, 20), dpi=150)
#     ax.scatter(C_test[i], C_pred[i], s=100, cmap=plt.cm.coolwarm, zorder=10)
#     lims = [
#     np.min([ax.get_xlim(), ax.get_ylim()]),  # min of both axes
#     np.max([ax.get_xlim(), ax.get_ylim()])]  # max of both axes
#     ax.plot(lims, lims, 'k-', alpha=1, zorder=0)
#     plt.xlabel(names[i]+' test data (MPa)',fontsize=40)
#     plt.ylabel("ANN "+names[i]+' prediction (MPa)',fontsize=40)
#     ax.set_aspect('equal')
#     ax.set_xlim(lims)
#     ax.set_ylim(lims)
#     plt.xticks(fontsize=20)
#     plt.yticks(fontsize=20)
#     # plt.title(names[i],fontsize=30)
```

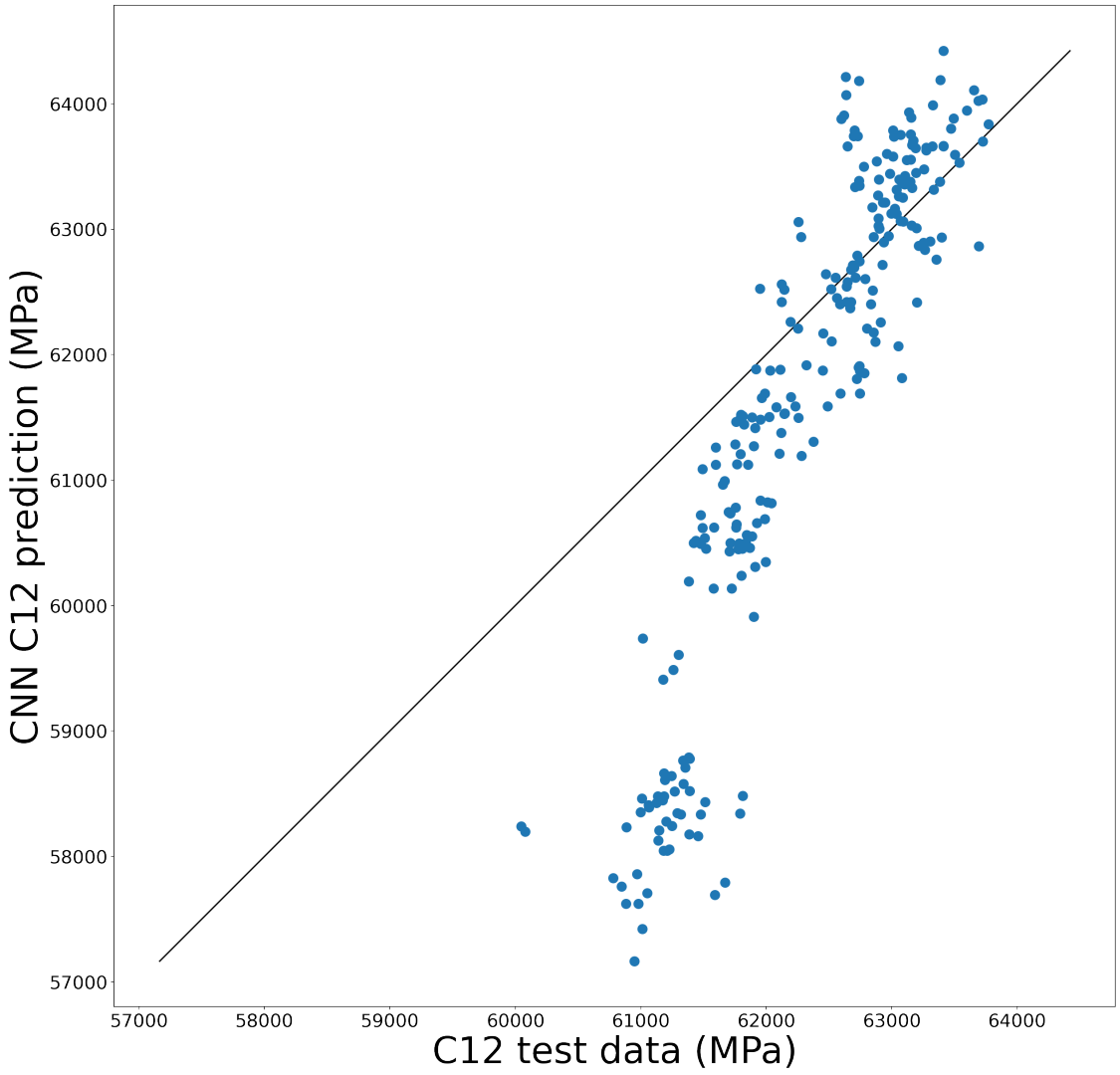
[44]:
```
len(C_test[12])
```

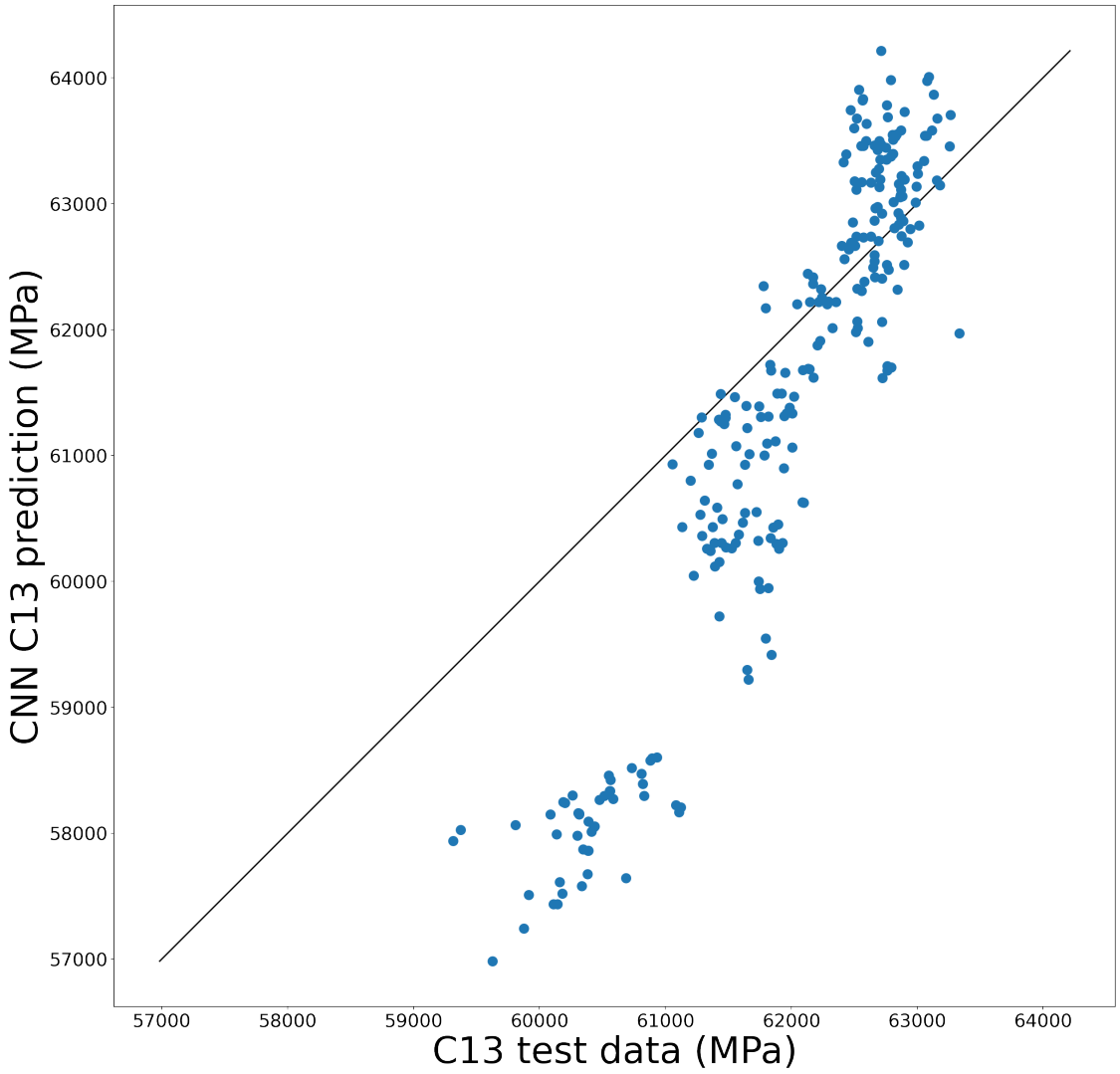
[44]:
```
13
```

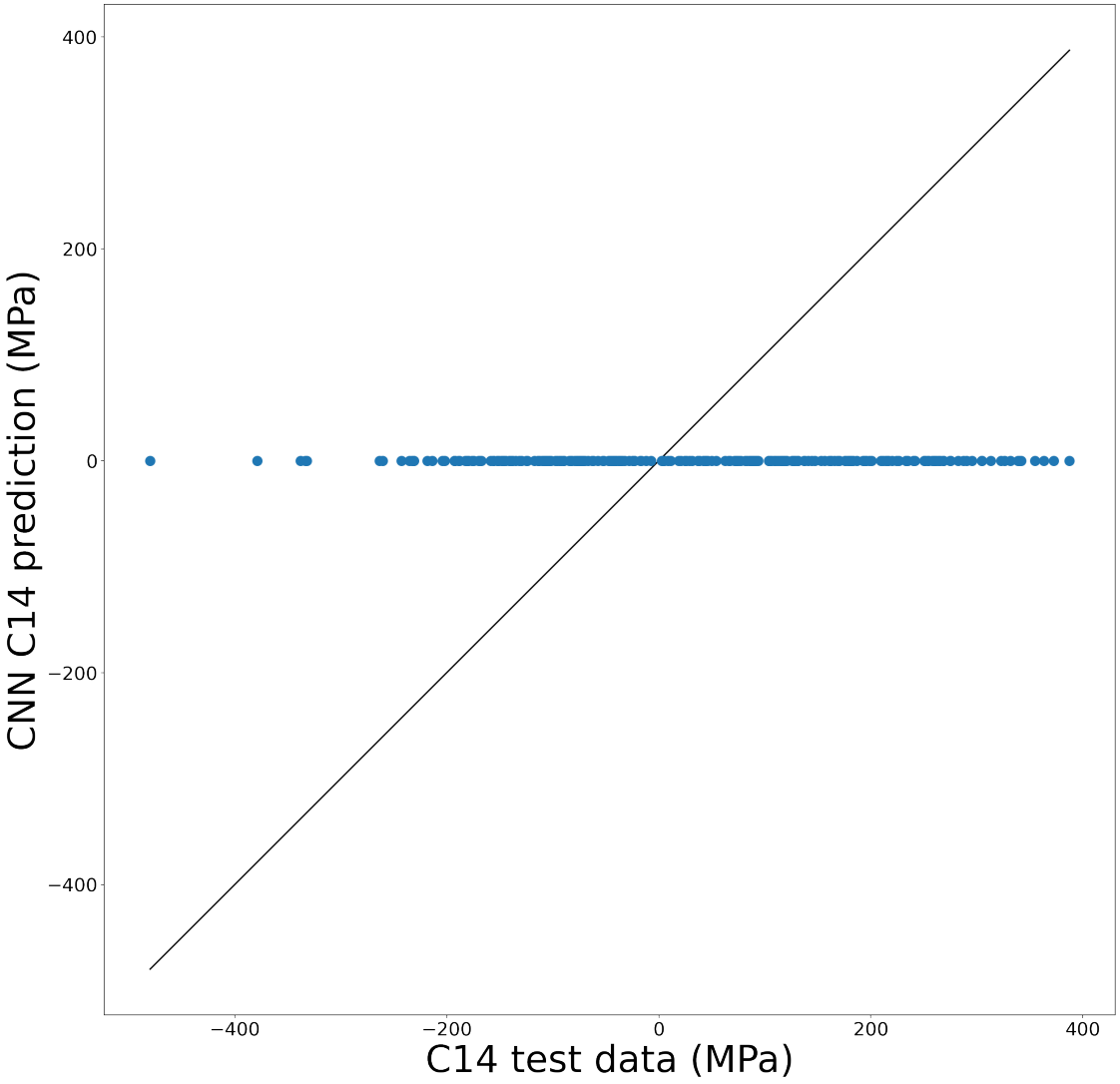
[50]:
```
for idx in range(13):

    fig, ax = plt.subplots(figsize=(20, 20), dpi=150)
    ax.scatter(C_test[idx], C_pred[idx], s=100, cmap=plt.cm.coolwarm, zorder=10)
```

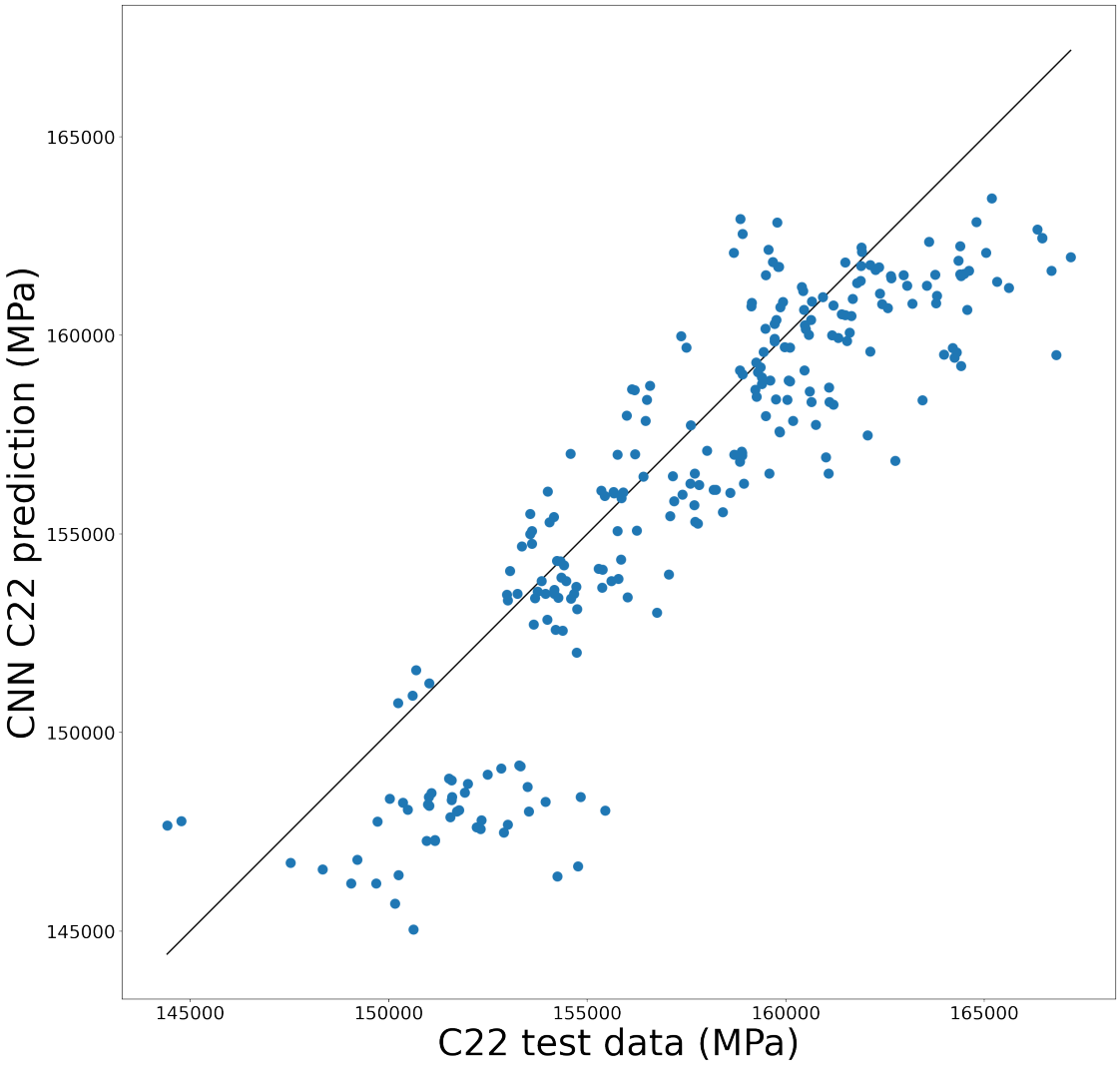
```
    lims = [
        np.min([np.min(C_test[idx]), np.min(C_pred[idx])]),  # min of both axes
        np.max([np.max(C_test[idx]), np.max(C_pred[idx])])]  # max of both axes
    ax.plot(lims, lims, 'k-', alpha=1, zorder=0)
    plt.xlabel(names[idx]+' test data (MPa)',fontsize=40)
    plt.ylabel("CNN "+names[idx]+' prediction (MPa)',fontsize=40)
    ax.set_aspect('equal')
    # plt.xlim([226000, 233000])
    # plt.ylim([226000, 233000])
    # ax.set_xlim(lims)
    # ax.set_ylim(lims)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    # plt.title(names[i], fontsize=30)
    plt.show()
```

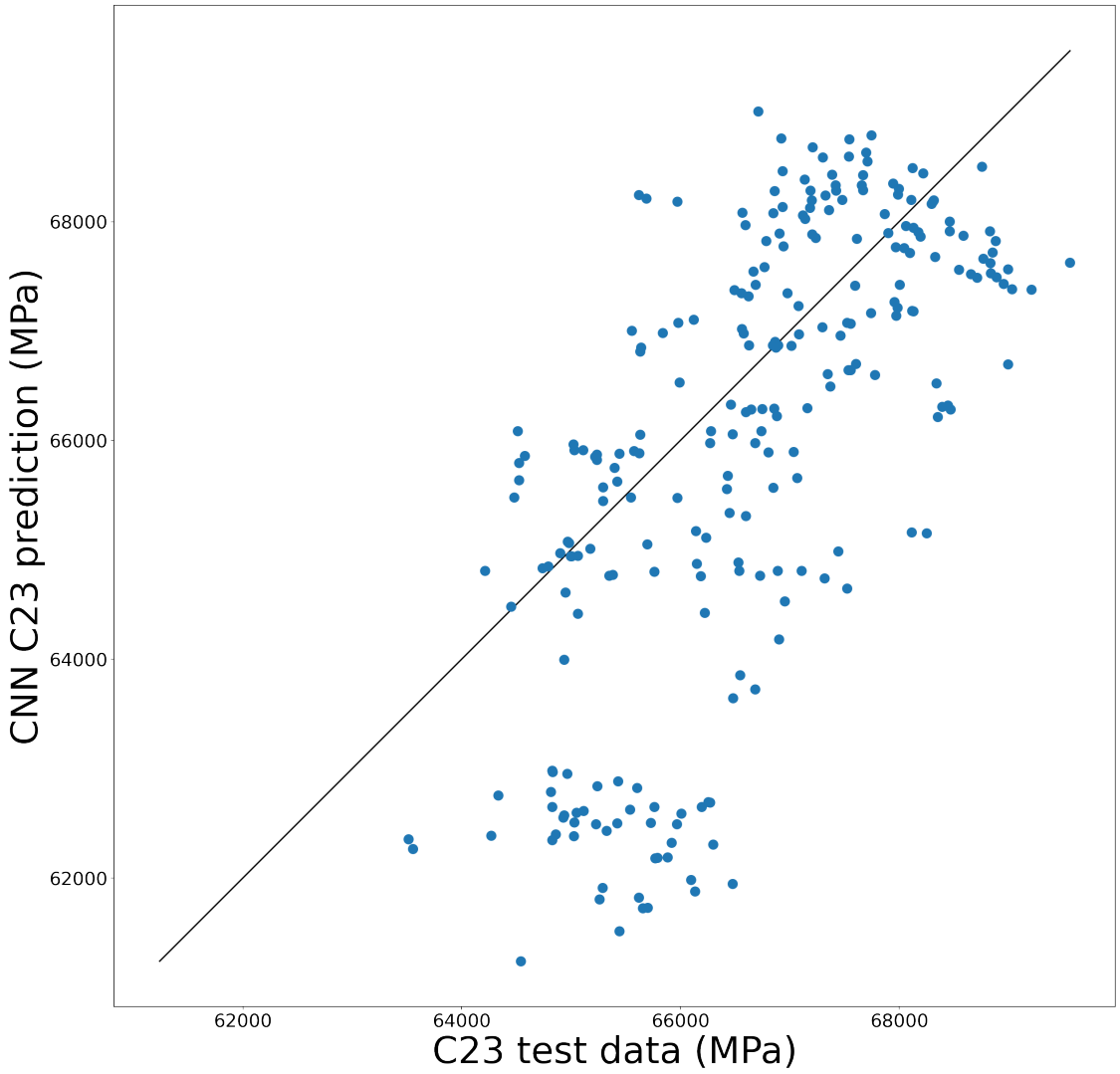


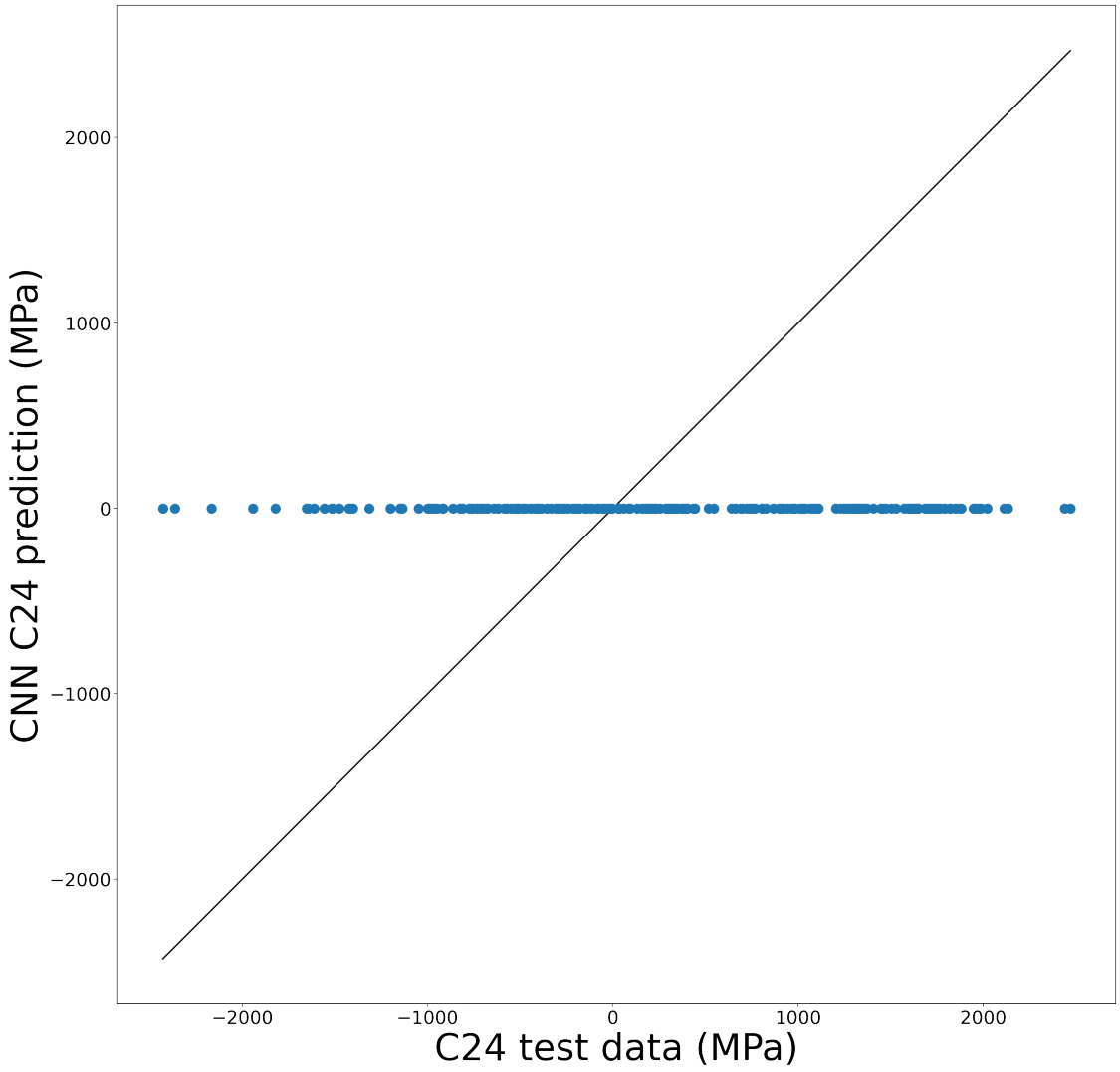

CNN C12 prediction (MPa)
C12 test data (MPa)
64000
63000
62000
61000
60000
59000
58000
57000
57000
58000
59000
60000
61000
62000
63000
64000

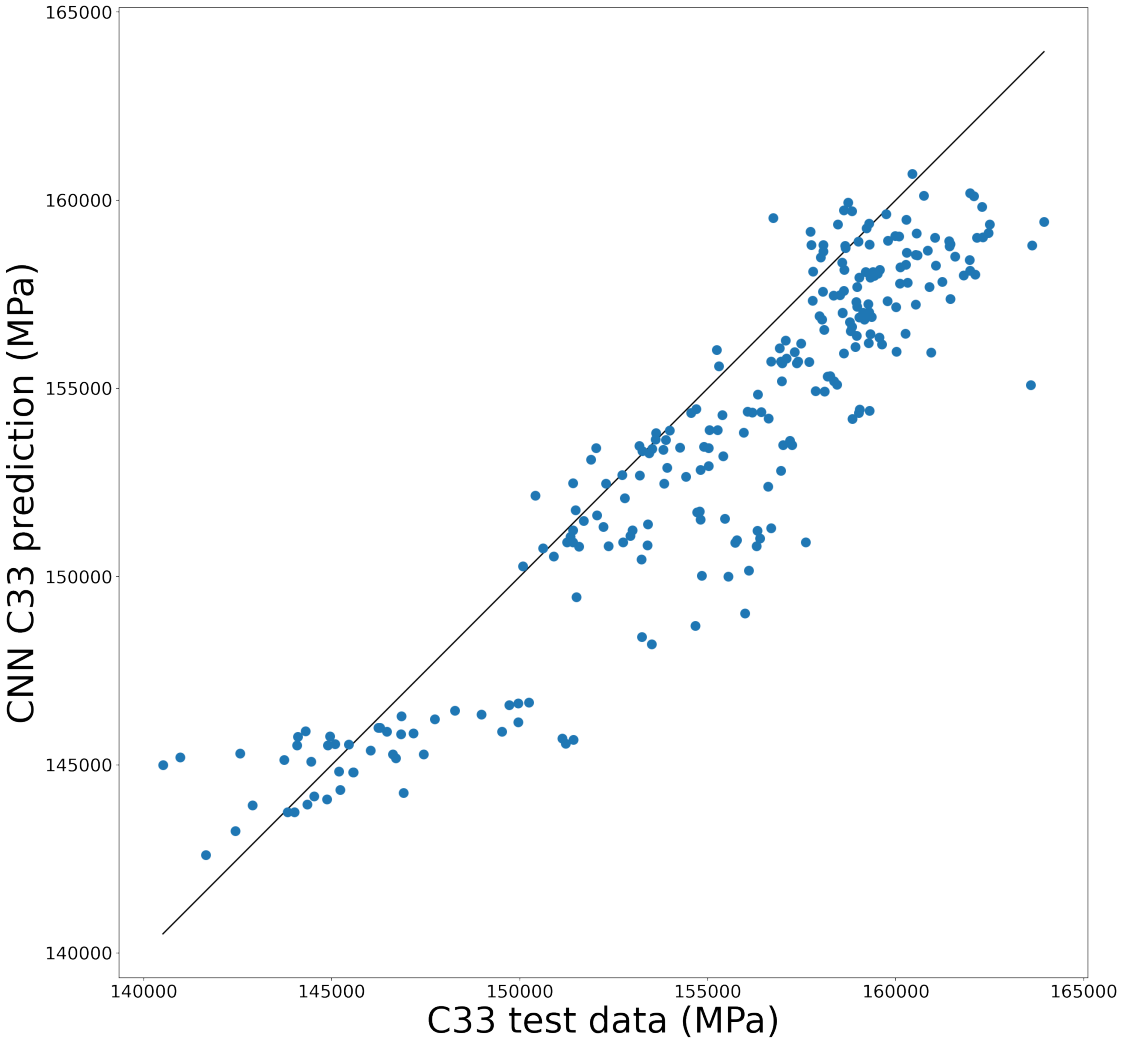

CNN C13 prediction (MPa)
C13 test data (MPa)
64000
63000
62000
61000
60000
59000
58000
57000
57000
58000
59000
60000
61000
62000
63000
64000

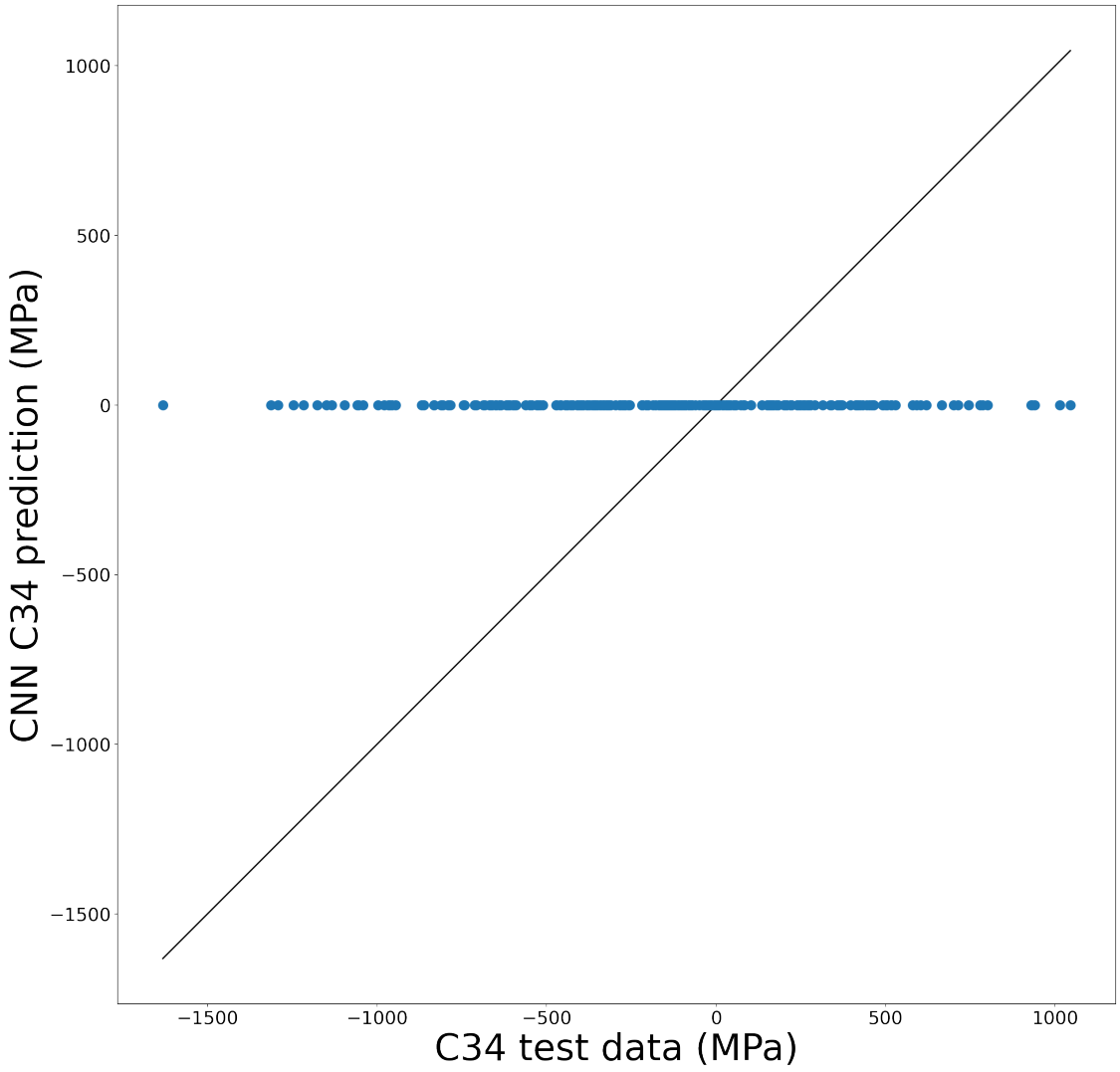

CNN C14 prediction (MPa)
C14 test data (MPa)
400
200
0
−200
−400
−400
−200
0
200
400

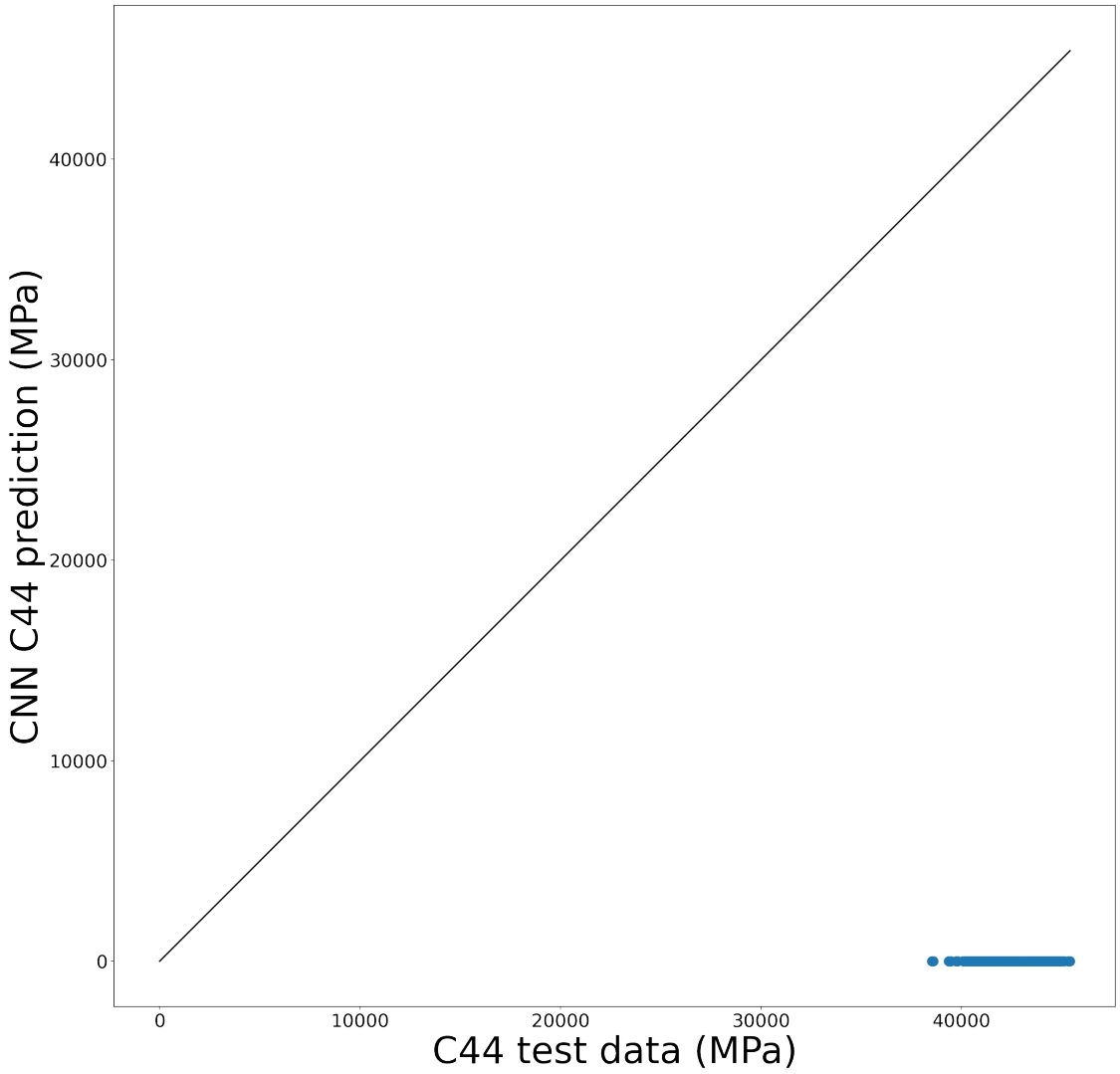

CNN C22 prediction (MPa)
165000
160000
155000
150000
145000
145000
150000
155000
160000
165000
C22 test data (MPa)

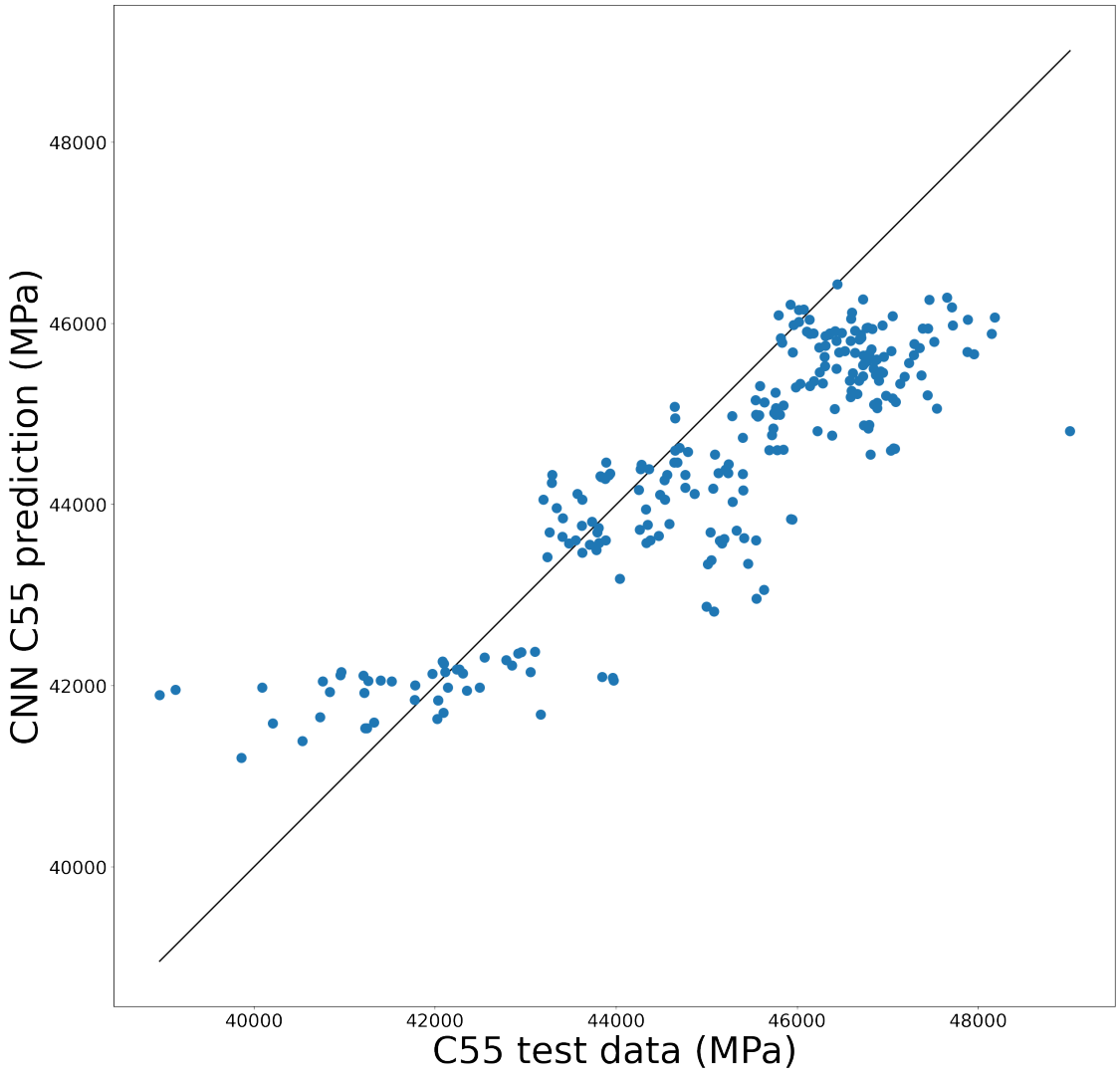

CNN C23 prediction (MPa)
68000
66000
64000
62000
62000
64000
66000
68000
C23 test data (MPa)

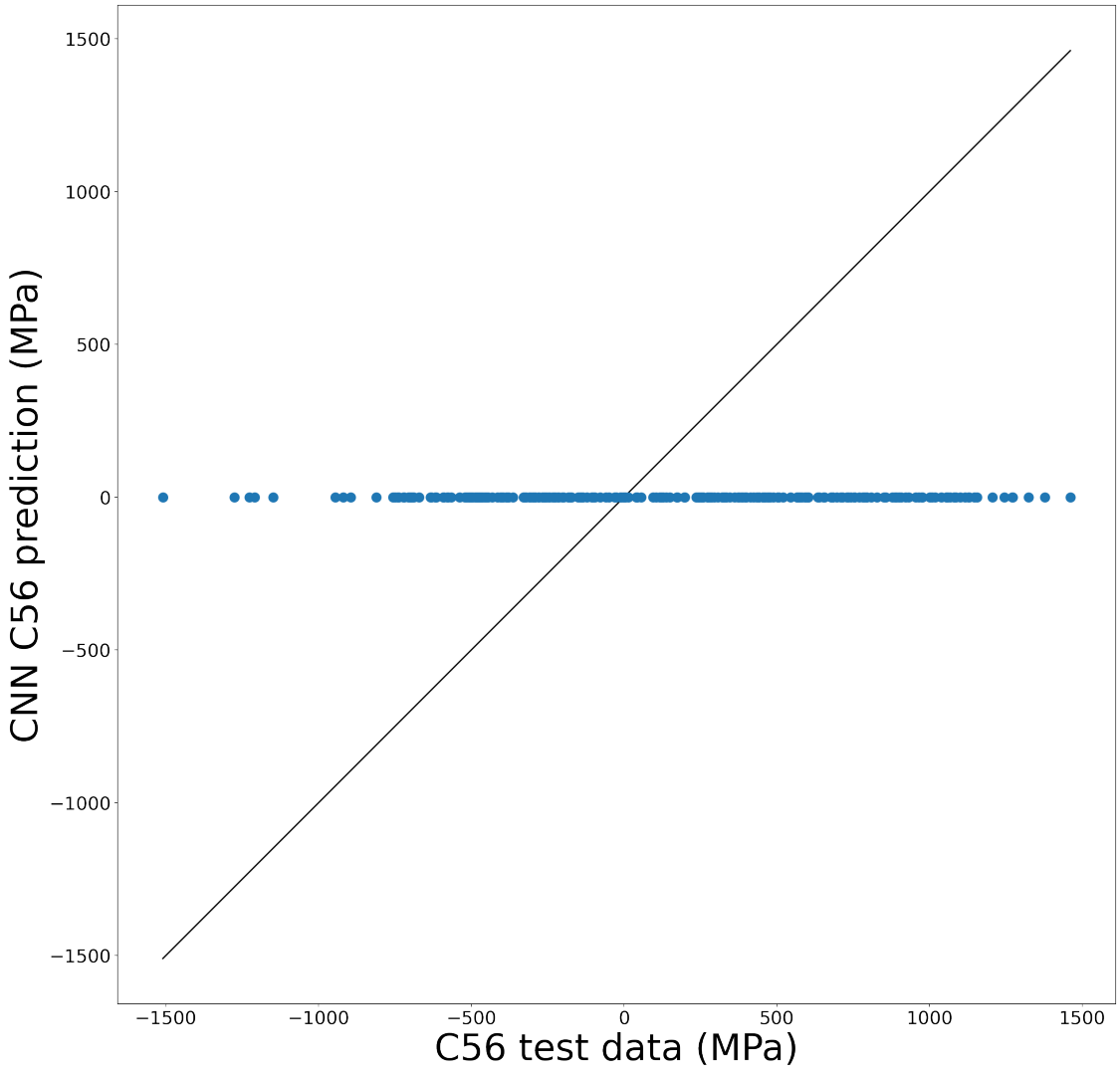

CNN C24 prediction (MPa)
2000
1000
0
−1000
−2000
−2000
−1000
0
1000
2000
C24 test data (MPa)

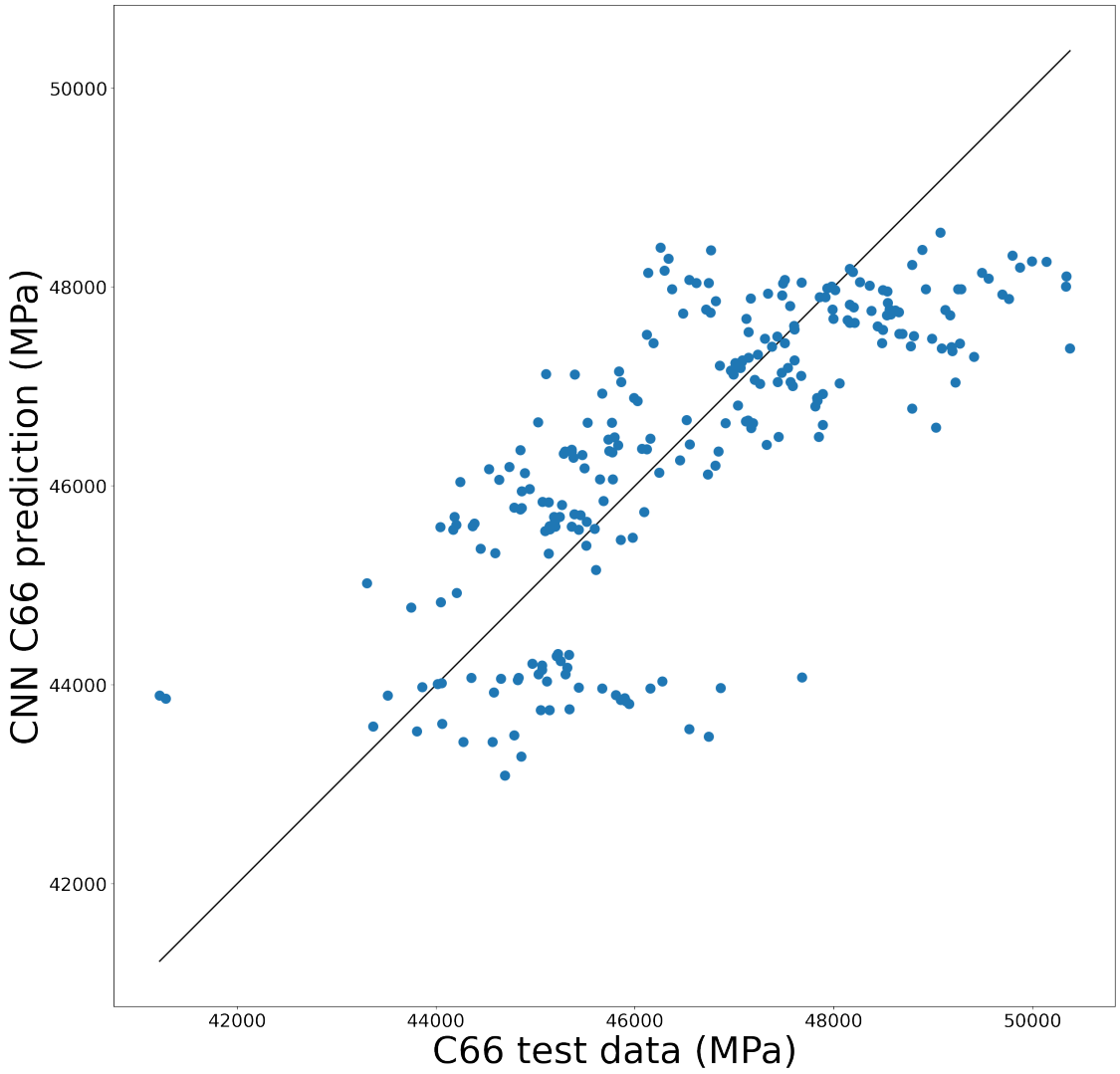

CNN C33 prediction (MPa)
C33 test data (MPa)
165000
160000
155000
150000
145000
140000
140000
145000
150000
155000
160000
165000


CNN C34 prediction (MPa)
C34 test data (MPa)
1000
500
0
−500
−1000
−1500
−1500
−1000
−500
0
500
1000


CNN C44 prediction (MPa)
C44 test data (MPa)
0
10000
20000
30000
40000


CNN C55 prediction (MPa)
48000
46000
44000
42000
40000
40000
42000
44000
46000
48000
C55 test data (MPa)


CNN C56 prediction (MPa)
C56 test data (MPa)
1500
1000
500
0
−500
−1000
−1500
−1500
−1000
−500
0
500
1000
1500


CNN C66 prediction (MPa)
C66 test data (MPa)
50000
48000
46000
44000
42000
42000
44000
46000
48000
50000

[ ]:

# CNN_1200_1600_1800_2000_LOP6

October 14, 2022

[1]:
```
# pip install imageio
```

[2]:
```
from keras.models import Sequential
from keras.layers import Dense, Conv1D,Conv2D, Flatten
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
from PIL import Image
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy.io as sio
import imageio
import keras
import os
import glob
```

[3]:
```
projectDir = r'C:\Users\18810\Desktop\RUC Collection'
print(projectDir)
```

```
C:\Users\18810\Desktop\RUC Collection
```

[4]:
```
extension = 'png'
x_filenames = glob.glob(os.path.join(projectDir+"\RUC 2000", '*.png'))
x_filenames[0]
im = imageio.imread(x_filenames[0])
print(im.shape)
```

```
(45, 60, 4)
```

[5]:
```
X = []
for i in range(2000):
# file = open(all_filenames[i])
    numpy_array = imageio.imread(x_filenames[i])
```

```
    X.append(numpy_array)
X = np.array(X)
print(X.shape)
```

```
(2000, 45, 60, 4)
```

[6]:
```
X = X.reshape(2000, 45, 60, 4)
X_input = X[0:2108]
print(X_input.shape)
# print(X_input[0])
```

```
(2000, 45, 60, 4)
```

[7]:
```
y = pd.read_csv('LOP_6_SIGMA12_collection.csv', header=None)
print(y.shape)
print(type(y))
# y = np.array(y)
y.head()
```

```
(25, 2001)
<class 'pandas.core.frame.DataFrame'>
```

[7]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | \ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.0002 | 9.5686 | 9.5686 | 9.5718 | 9.5762 | 9.5686 | 9.5686 | 9.5786 | |
| 1 | 0.0004 | 19.1360 | 19.1360 | 19.1420 | 19.1510 | 19.1360 | 19.1360 | 19.1560 | |
| 2 | 0.0006 | 28.5980 | 28.5980 | 28.6170 | 28.6210 | 28.5980 | 28.6110 | 28.6370 | |
| 3 | 0.0008 | 36.4790 | 36.4790 | 36.4880 | 36.4780 | 36.4790 | 36.4960 | 36.4780 | |
| 4 | 0.0010 | 42.3200 | 42.3200 | 42.3150 | 42.3080 | 42.3200 | 42.3350 | 42.2870 | |

| | 8 | 9 | ... | 1991 | 1992 | 1993 | 1994 | 1995 | 1996 | \ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 9.574 | 9.5794 | ... | 9.3592 | 9.3455 | 9.3448 | 9.3599 | 9.3448 | 9.3469 | |
| 1 | 19.147 | 19.1570 | ... | 18.7160 | 18.6890 | 18.6870 | 18.7180 | 18.6870 | 18.6920 | |
| 2 | 28.624 | 28.6290 | ... | 27.9810 | 27.9480 | 27.9430 | 27.9890 | 27.9460 | 27.9480 | |
| 3 | 36.484 | 36.4960 | ... | 36.3190 | 36.2980 | 36.2830 | 36.3210 | 36.2650 | 36.2500 | |
| 4 | 42.311 | 42.3240 | ... | 42.7930 | 42.7910 | 42.7830 | 42.8220 | 42.7810 | 42.7670 | |

| | 1997 | 1998 | 1999 | 2000 |
|---|---|---|---|---|
| 0 | 9.3494 | 9.3492 | 9.3543 | 9.3494 |
| 1 | 18.6970 | 18.6960 | 18.7070 | 18.6960 |
| 2 | 27.9600 | 27.9570 | 27.9640 | 27.9330 |
| 3 | 36.2610 | 36.2410 | 36.2270 | 36.2080 |
| 4 | 42.7790 | 42.7630 | 42.7350 | 42.7320 |

[5 rows x 2001 columns]

[8]:
```
Y = []
for col in y.columns:
    Y.append(y[col])
    Y_input = np.array(Y[1:2001])
```

```
print(Y_input[1999])
print(Y_input.shape)
```

```
[ 9.3494 18.696  27.933  36.208  42.732  47.094  49.55   50.357  50.882
 51.349  51.787  52.207  52.614  53.01   53.398  53.778  54.151  54.513
 54.869  55.22   55.567  55.911  56.251  56.589  56.924 ]
(2000, 25)
```

[9]:
```
X_train, X_test, y_train, y_test = train_test_split(X_input, Y_input,␣
↪test_size=0.05, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
↪026, random_state=42)
X_train, X_verify, y_train, y_verify = train_test_split(X_train, y_train,␣
↪test_size=0.027, random_state=42)
print(len(X_train))
print(len(X_test))
print(len(X_val))
print(len(X_verify))
```

```
1800
100
50
50
```

[10]:
```
model = keras.Sequential([
    # keras.layers.Conv2D(32, (2,2), padding="same",␣
↪activation="elu",input_shape=(60, 45, 4)),
    keras.layers.Conv2D(256, (2,2), padding="same",␣
↪activation="elu",input_shape=(45, 60, 4)),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="elu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="elu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="elu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="elu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="elu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="elu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="elu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="elu"),
    keras.layers.Flatten(),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
    keras.layers.Dense(256, activation="relu"),
```

```
    keras.layers.Dense(25)
])
model.compile(loss=keras.losses.MeanSquaredError(), optimizer=tf.keras.
 ↪optimizers.Adam(learning_rate=0.0001), metrics=["accuracy",␣
 ↪"mean_squared_error"])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 45, 60, 256)       4352

 conv2d_1 (Conv2D)           (None, 45, 60, 256)       262400

 conv2d_2 (Conv2D)           (None, 45, 60, 256)       262400

 conv2d_3 (Conv2D)           (None, 45, 60, 256)       262400

 conv2d_4 (Conv2D)           (None, 45, 60, 256)       262400

 conv2d_5 (Conv2D)           (None, 45, 60, 256)       262400

 conv2d_6 (Conv2D)           (None, 45, 60, 256)       262400

 conv2d_7 (Conv2D)           (None, 45, 60, 256)       262400

 conv2d_8 (Conv2D)           (None, 45, 60, 256)       262400

 flatten (Flatten)           (None, 691200)            0

 dense (Dense)               (None, 256)               176947456

 dense_1 (Dense)             (None, 256)               65792

 dense_2 (Dense)             (None, 256)               65792

 dense_3 (Dense)             (None, 256)               65792

 dense_4 (Dense)             (None, 256)               65792

 dense_5 (Dense)             (None, 256)               65792

 dense_6 (Dense)             (None, 256)               65792

 dense_7 (Dense)             (None, 256)               65792

 dense_8 (Dense)             (None, 256)               65792
```

```
 dense_9 (Dense)             (None, 256)               65792

 dense_10 (Dense)            (None, 25)                6425

=================================================================
Total params: 179,649,561
Trainable params: 179,649,561
Non-trainable params: 0
_________________________________________________________________
```

[11]:
```
tf.__version__
```

[11]:
```
'2.9.1'
```

[12]:
```
# pip install --upgrade tensorflow-gpu --user
```

[13]:
```
print("Num GPUs Available: ", len(tf.config.experimental.
↪list_physical_devices('GPU')))
```

```
Num GPUs Available:  1
```

[14]:
```
with tf.device('/gpu:1'):
    model.fit(X_train, y_train, batch_size=20, epochs=25, verbose=2,␣
↪validation_data=(X_val, y_val))
```

```
Epoch 1/50
90/90 - 11s - loss: 152.2352 - accuracy: 0.4850 - mean_squared_error: 152.2352 -
val_loss: 0.2969 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2969 -
11s/epoch - 125ms/step
Epoch 2/50
90/90 - 5s - loss: 0.3930 - accuracy: 1.0000 - mean_squared_error: 0.3930 -
val_loss: 0.7076 - val_accuracy: 1.0000 - val_mean_squared_error: 0.7076 -
5s/epoch - 61ms/step
Epoch 3/50
90/90 - 6s - loss: 0.2867 - accuracy: 1.0000 - mean_squared_error: 0.2867 -
val_loss: 0.1140 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1140 -
6s/epoch - 65ms/step
Epoch 4/50
90/90 - 6s - loss: 0.2693 - accuracy: 1.0000 - mean_squared_error: 0.2693 -
val_loss: 0.3617 - val_accuracy: 1.0000 - val_mean_squared_error: 0.3617 -
6s/epoch - 66ms/step
Epoch 5/50
90/90 - 6s - loss: 0.4747 - accuracy: 1.0000 - mean_squared_error: 0.4747 -
val_loss: 0.4706 - val_accuracy: 1.0000 - val_mean_squared_error: 0.4706 -
6s/epoch - 65ms/step
Epoch 6/50
90/90 - 6s - loss: 0.2666 - accuracy: 1.0000 - mean_squared_error: 0.2666 -
val_loss: 0.1075 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1075 -
```

```
6s/epoch - 62ms/step
Epoch 7/50
90/90 - 6s - loss: 0.2429 - accuracy: 1.0000 - mean_squared_error: 0.2429 -
val_loss: 0.2244 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2244 -
6s/epoch - 64ms/step
Epoch 8/50
90/90 - 6s - loss: 0.1152 - accuracy: 1.0000 - mean_squared_error: 0.1152 -
val_loss: 0.0440 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0440 -
6s/epoch - 66ms/step
Epoch 9/50
90/90 - 6s - loss: 0.2255 - accuracy: 1.0000 - mean_squared_error: 0.2255 -
val_loss: 0.2783 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2783 -
6s/epoch - 67ms/step
Epoch 10/50
90/90 - 6s - loss: 0.1116 - accuracy: 1.0000 - mean_squared_error: 0.1116 -
val_loss: 0.5063 - val_accuracy: 1.0000 - val_mean_squared_error: 0.5063 -
6s/epoch - 65ms/step
Epoch 11/50
90/90 - 6s - loss: 0.1804 - accuracy: 1.0000 - mean_squared_error: 0.1804 -
val_loss: 0.4115 - val_accuracy: 1.0000 - val_mean_squared_error: 0.4115 -
6s/epoch - 66ms/step
Epoch 12/50
90/90 - 6s - loss: 0.3975 - accuracy: 0.9989 - mean_squared_error: 0.3975 -
val_loss: 0.0787 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0787 -
6s/epoch - 65ms/step
Epoch 13/50
90/90 - 6s - loss: 0.1815 - accuracy: 0.9994 - mean_squared_error: 0.1815 -
val_loss: 0.1002 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1002 -
6s/epoch - 65ms/step
Epoch 14/50
90/90 - 6s - loss: 0.1966 - accuracy: 1.0000 - mean_squared_error: 0.1966 -
val_loss: 0.0817 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0817 -
6s/epoch - 68ms/step
Epoch 15/50
90/90 - 6s - loss: 0.0686 - accuracy: 1.0000 - mean_squared_error: 0.0686 -
val_loss: 0.0326 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0326 -
6s/epoch - 63ms/step
Epoch 16/50
90/90 - 6s - loss: 0.0709 - accuracy: 1.0000 - mean_squared_error: 0.0709 -
val_loss: 0.1227 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1227 -
6s/epoch - 63ms/step
Epoch 17/50
90/90 - 6s - loss: 0.2320 - accuracy: 1.0000 - mean_squared_error: 0.2320 -
val_loss: 0.1712 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1712 -
6s/epoch - 63ms/step
Epoch 18/50
90/90 - 6s - loss: 0.2219 - accuracy: 0.9994 - mean_squared_error: 0.2219 -
val_loss: 0.1067 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1067 -
```

```
6s/epoch - 63ms/step
Epoch 19/50
90/90 - 6s - loss: 0.1854 - accuracy: 1.0000 - mean_squared_error: 0.1854 -
val_loss: 0.1151 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1151 -
6s/epoch - 63ms/step
Epoch 20/50
90/90 - 6s - loss: 0.1504 - accuracy: 0.9994 - mean_squared_error: 0.1504 -
val_loss: 0.0232 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0232 -
6s/epoch - 66ms/step
Epoch 21/50
90/90 - 6s - loss: 0.0945 - accuracy: 1.0000 - mean_squared_error: 0.0945 -
val_loss: 0.2166 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2166 -
6s/epoch - 69ms/step
Epoch 22/50
90/90 - 6s - loss: 0.1725 - accuracy: 0.9994 - mean_squared_error: 0.1725 -
val_loss: 0.0348 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0348 -
6s/epoch - 63ms/step
Epoch 23/50
90/90 - 6s - loss: 0.2128 - accuracy: 1.0000 - mean_squared_error: 0.2128 -
val_loss: 0.2323 - val_accuracy: 1.0000 - val_mean_squared_error: 0.2323 -
6s/epoch - 62ms/step
Epoch 24/50
90/90 - 6s - loss: 0.0935 - accuracy: 1.0000 - mean_squared_error: 0.0935 -
val_loss: 0.0371 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0371 -
6s/epoch - 64ms/step
Epoch 25/50
90/90 - 6s - loss: 0.0613 - accuracy: 1.0000 - mean_squared_error: 0.0613 -
val_loss: 0.0818 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0818 -
6s/epoch - 64ms/step
Epoch 26/50
90/90 - 6s - loss: 0.1490 - accuracy: 1.0000 - mean_squared_error: 0.1490 -
val_loss: 0.0646 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0646 -
6s/epoch - 64ms/step
Epoch 27/50
90/90 - 6s - loss: 0.0956 - accuracy: 1.0000 - mean_squared_error: 0.0956 -
val_loss: 0.0661 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0661 -
6s/epoch - 63ms/step
Epoch 28/50
90/90 - 6s - loss: 0.1620 - accuracy: 1.0000 - mean_squared_error: 0.1620 -
val_loss: 0.0166 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0166 -
6s/epoch - 63ms/step
Epoch 29/50
90/90 - 6s - loss: 0.1682 - accuracy: 0.9889 - mean_squared_error: 0.1682 -
val_loss: 0.1244 - val_accuracy: 0.8800 - val_mean_squared_error: 0.1244 -
6s/epoch - 63ms/step
Epoch 30/50
90/90 - 6s - loss: 0.0743 - accuracy: 0.9961 - mean_squared_error: 0.0743 -
val_loss: 0.0460 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0460 -
```

```
6s/epoch - 63ms/step
Epoch 31/50
90/90 - 6s - loss: 0.0841 - accuracy: 1.0000 - mean_squared_error: 0.0841 -
val_loss: 0.0203 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0203 -
6s/epoch - 63ms/step
Epoch 32/50
90/90 - 6s - loss: 0.0972 - accuracy: 0.9833 - mean_squared_error: 0.0972 -
val_loss: 0.0618 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0618 -
6s/epoch - 63ms/step
Epoch 33/50
90/90 - 6s - loss: 0.1569 - accuracy: 0.9417 - mean_squared_error: 0.1569 -
val_loss: 0.0215 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0215 -
6s/epoch - 62ms/step
Epoch 34/50
90/90 - 6s - loss: 0.1024 - accuracy: 1.0000 - mean_squared_error: 0.1024 -
val_loss: 0.0206 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0206 -
6s/epoch - 63ms/step
Epoch 35/50
90/90 - 6s - loss: 0.2149 - accuracy: 0.9989 - mean_squared_error: 0.2149 -
val_loss: 0.1772 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1772 -
6s/epoch - 63ms/step
Epoch 36/50
90/90 - 6s - loss: 0.0731 - accuracy: 0.9983 - mean_squared_error: 0.0731 -
val_loss: 0.0216 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0216 -
6s/epoch - 66ms/step
Epoch 37/50
90/90 - 6s - loss: 0.1524 - accuracy: 0.9994 - mean_squared_error: 0.1524 -
val_loss: 0.0643 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0643 -
6s/epoch - 64ms/step
Epoch 38/50
90/90 - 6s - loss: 0.1211 - accuracy: 1.0000 - mean_squared_error: 0.1211 -
val_loss: 0.1927 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1927 -
6s/epoch - 63ms/step
Epoch 39/50
90/90 - 6s - loss: 0.1002 - accuracy: 1.0000 - mean_squared_error: 0.1002 -
val_loss: 0.0324 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0324 -
6s/epoch - 67ms/step
Epoch 40/50
90/90 - 6s - loss: 0.1240 - accuracy: 1.0000 - mean_squared_error: 0.1240 -
val_loss: 0.0817 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0817 -
6s/epoch - 66ms/step
Epoch 41/50
90/90 - 6s - loss: 0.1299 - accuracy: 0.9617 - mean_squared_error: 0.1299 -
val_loss: 0.0534 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0534 -
6s/epoch - 66ms/step
Epoch 42/50
90/90 - 6s - loss: 0.0937 - accuracy: 0.9456 - mean_squared_error: 0.0937 -
val_loss: 0.0561 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0561 -
```

```
6s/epoch - 66ms/step
Epoch 43/50
90/90 - 6s - loss: 0.1111 - accuracy: 1.0000 - mean_squared_error: 0.1111 -
val_loss: 0.1045 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1045 -
6s/epoch - 72ms/step
Epoch 44/50
90/90 - 6s - loss: 0.0776 - accuracy: 0.9983 - mean_squared_error: 0.0776 -
val_loss: 0.0641 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0641 -
6s/epoch - 66ms/step
Epoch 45/50
90/90 - 6s - loss: 0.0529 - accuracy: 1.0000 - mean_squared_error: 0.0529 -
val_loss: 0.0197 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0197 -
6s/epoch - 65ms/step
Epoch 46/50
90/90 - 6s - loss: 0.1392 - accuracy: 0.9700 - mean_squared_error: 0.1392 -
val_loss: 0.1685 - val_accuracy: 0.0000e+00 - val_mean_squared_error: 0.1685 -
6s/epoch - 64ms/step
Epoch 47/50
90/90 - 6s - loss: 0.3849 - accuracy: 0.9344 - mean_squared_error: 0.3849 -
val_loss: 0.1060 - val_accuracy: 1.0000 - val_mean_squared_error: 0.1060 -
6s/epoch - 67ms/step
Epoch 48/50
90/90 - 6s - loss: 0.1695 - accuracy: 0.9267 - mean_squared_error: 0.1695 -
val_loss: 0.0171 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0171 -
6s/epoch - 70ms/step
Epoch 49/50
90/90 - 6s - loss: 0.0505 - accuracy: 0.9994 - mean_squared_error: 0.0505 -
val_loss: 0.0117 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0117 -
6s/epoch - 66ms/step
Epoch 50/50
90/90 - 6s - loss: 0.0465 - accuracy: 1.0000 - mean_squared_error: 0.0465 -
val_loss: 0.0460 - val_accuracy: 1.0000 - val_mean_squared_error: 0.0460 -
6s/epoch - 72ms/step
```

[15]:
```
losses = pd.DataFrame(model.history.history)
losses.tail(1)
```

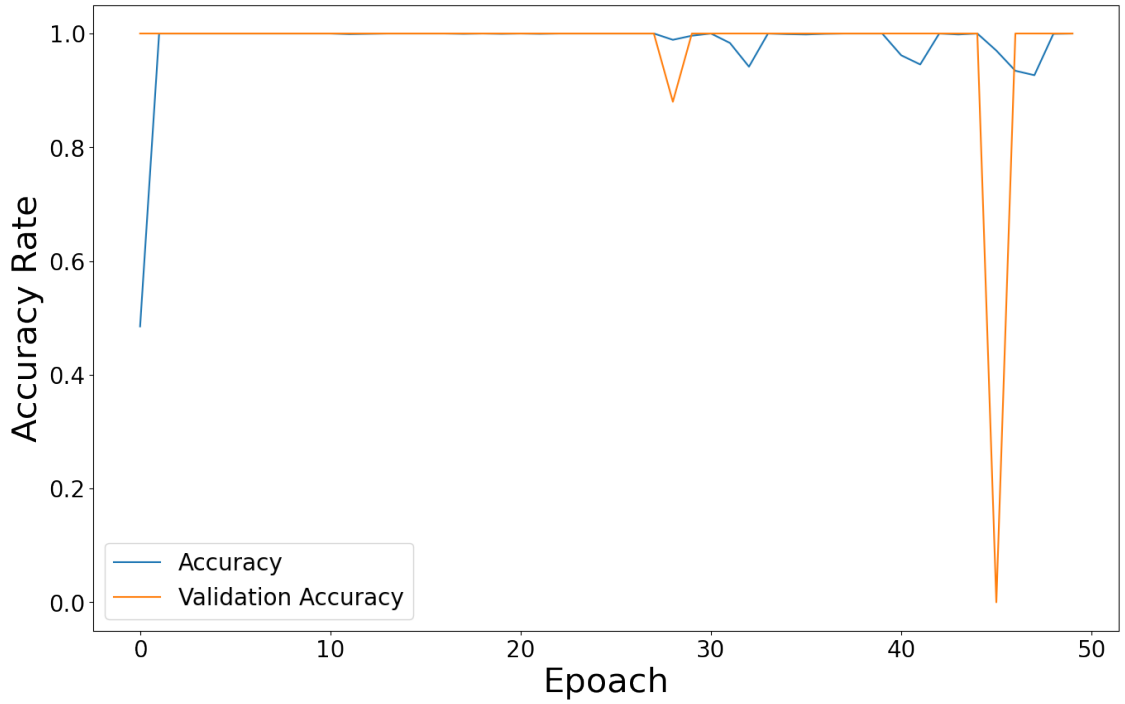
[15]:

| | loss | accuracy | mean_squared_error | val_loss | val_accuracy | val_mean_squared_error |
|---|---|---|---|---|---|---|
| 49 | 0.046501 | 1.0 | 0.046501 | 0.045956 | 1.0 | 0.045956 |

[16]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(losses['accuracy'],label='Accuracy')
plt.plot(losses['val_accuracy'],label='Validation Accuracy')
plt.xlabel('Epoach', fontsize=30)
plt.ylabel('Accuracy Rate', fontsize=30)
```

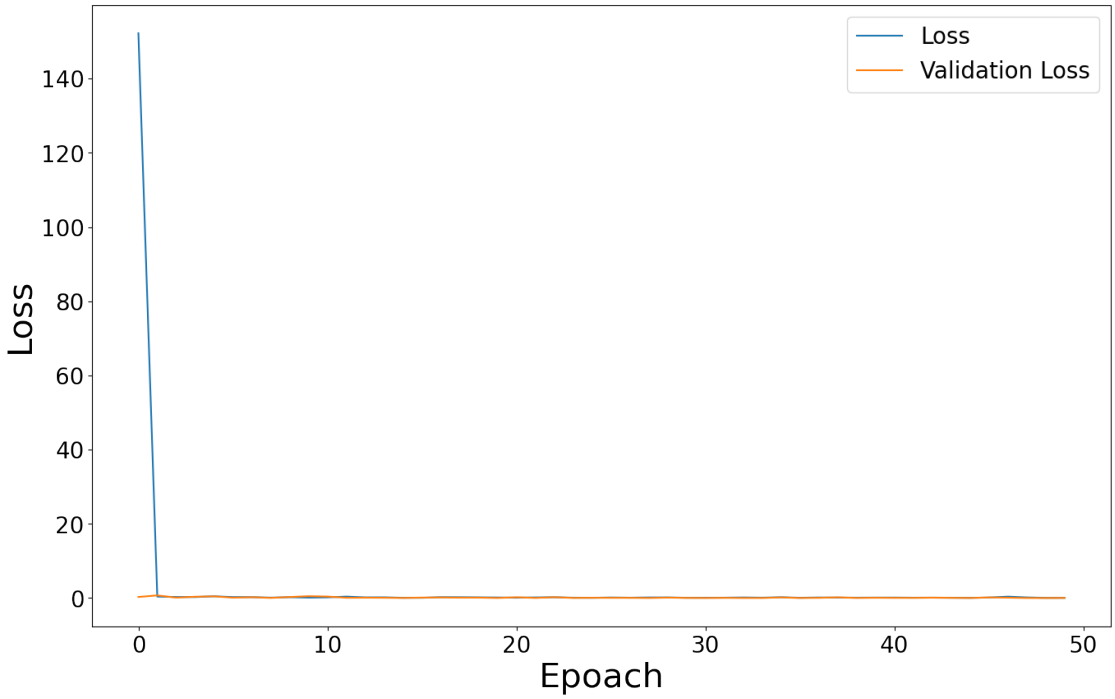
```
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20)
```

[16]: <matplotlib.legend.Legend at 0x244dcf828e0>




[17]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(losses['loss'],label='Loss')
plt.plot(losses['val_loss'],label='Validation Loss')
plt.xlabel('Epoach', fontsize=30)
plt.ylabel('Loss', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20)
```

[17]: <matplotlib.legend.Legend at 0x244dd0342b0>




[18]:
```
model.metrics_names
```

[18]:
```
['loss', 'accuracy', 'mean_squared_error']
```

[19]:
```
print(y_verify)
len(y_verify)
```

```
[[ 9.6114 19.222  28.717  ... 57.052  57.413  57.77  ]
 [ 9.5189 19.037  28.472  ... 57.353  57.693  58.031 ]
 [ 9.5293 19.058  28.473  ... 56.518  56.872  57.223 ]
 ...
 [ 9.5378 19.069  28.376  ... 56.557  56.913  57.266 ]
 [ 9.5156 19.03   28.442  ... 57.081  57.426  57.768 ]
 [ 9.5596 19.118  28.594  ... 56.964  57.316  57.665 ]]
```

[19]:
```
50
```

[20]:
```
predictions = model.predict(X_verify)
predictions
```

```
2/2 [==============================] - 0s 109ms/step
```

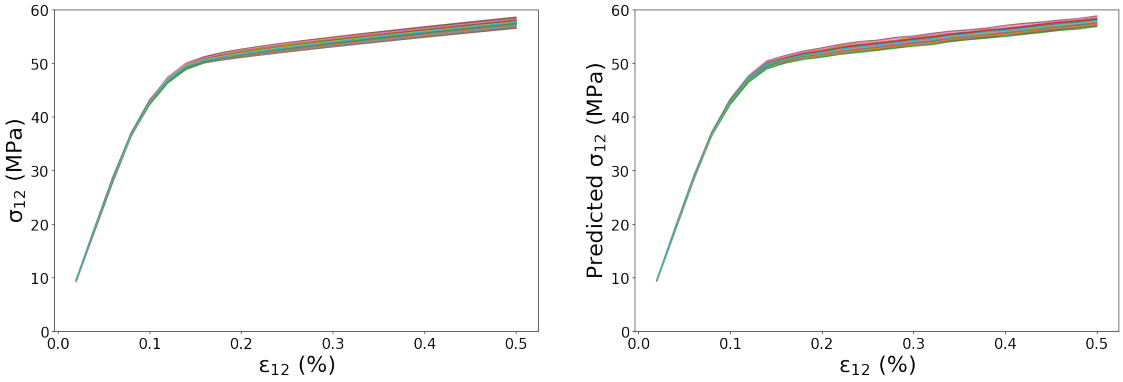
[20]:
```
array([[ 9.5347805, 19.25354  , 28.70302  , ..., 57.328552 , 57.553894 ,
         58.009266 ],
       [ 9.48955  , 19.090202 , 28.57742  , ..., 57.669415 , 58.02167  ,
         58.36301  ],
```

```
        [ 9.509289 , 19.055532 , 28.554155 , ..., 56.627483 , 56.949406 ,
         57.378407 ],
        ...,
        [ 9.442978 , 19.045794 , 28.42243  , ..., 56.85839  , 57.188145 ,
         57.540554 ],
        [ 9.434324 , 19.06074  , 28.459711 , ..., 57.189808 , 57.495354 ,
         57.89616  ],
        [ 9.535934 , 19.17113  , 28.62891  , ..., 57.29061  , 57.591164 ,
         57.985977 ]], dtype=float32)
```

[21]:
```
x_axis = np.linspace(0.02,0.5,num=25)
y_axis = np.linspace(0,200,10)
print(x_axis)
```

```
[0.02 0.04 0.06 0.08 0.1  0.12 0.14 0.16 0.18 0.2  0.22 0.24 0.26 0.28
 0.3  0.32 0.34 0.36 0.38 0.4  0.42 0.44 0.46 0.48 0.5 ]
```

[22]:
```
plt.figure(figsize=(40,8), dpi=100)
plt.subplot(131)
for i in range(len(y_verify)):
    plt.plot(x_axis, y_verify[i])
plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
plt.ylabel('\u03C3$_{12}$ (MPa)', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.ylim([0, 60])


plt.subplot(132)
for i in range(len(predictions)):
    plt.plot(x_axis, predictions[i])
plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
plt.ylabel('Predicted \u03C3$_{12}$ (MPa)', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.ylim([0, 60])

plt.show()
```

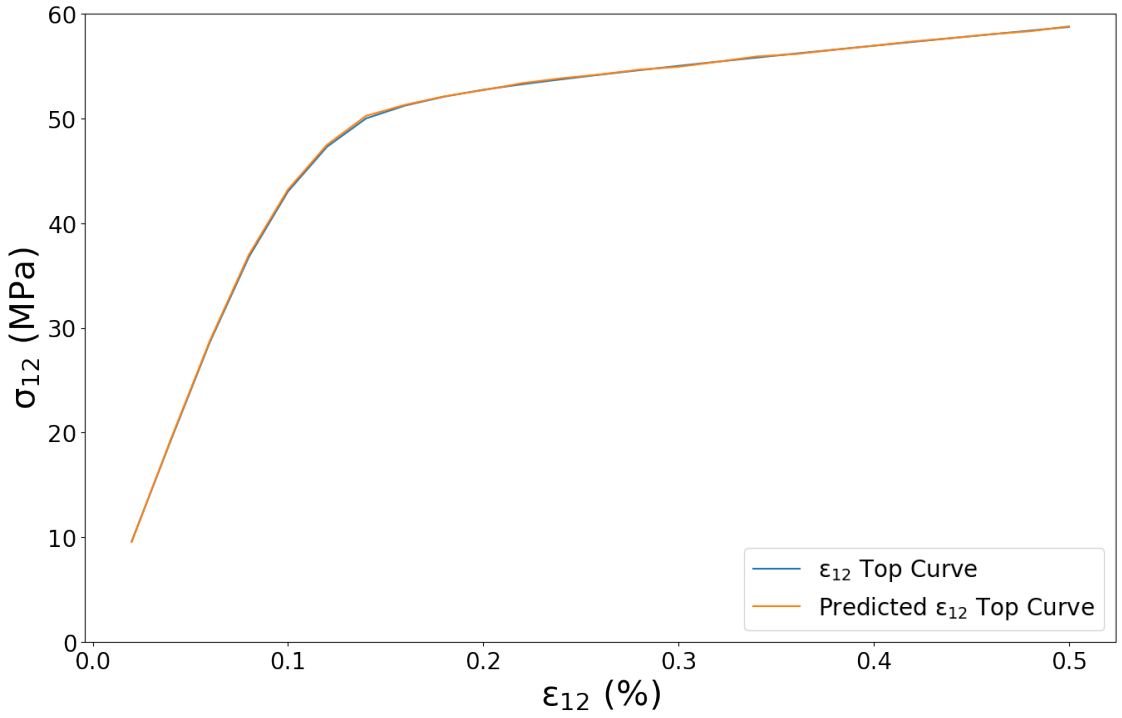



[23]:
```
# for i in range(5):
#     plt.figure(figsize=(16,12), dpi=100)
#
#     plt.plot(x_axis, y_verify[i], label='True Stress Strain')
#     plt.plot(x_axis, predictions[i], label='Predicted Stress Strain')
#     plt.legend(fontsize=20)
#     plt.title("True Curve versus Predicted Curve Plot")
#     plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
#     plt.ylabel('\u03C3$_{12}$ (MPa)', fontsize=30)
#     plt.xticks(fontsize=20)
#     plt.yticks(fontsize=20)
#     plt.ylim([0, 60])
#     plt.show()
```

[24]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(x_axis, Y_input[1379], label= "\u03B5$_{12}$ Top Curve")
plt.plot(x_axis, model.predict(X_input)[1379], label= "Predicted \u03B5$_{12}$␣
↪Top Curve")
plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
plt.ylabel('\u03C3$_{12}$ (MPa)', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20, loc = 'lower right')
plt.ylim([0, 60])
```

```
63/63 [==============================] - 2s 33ms/step
```

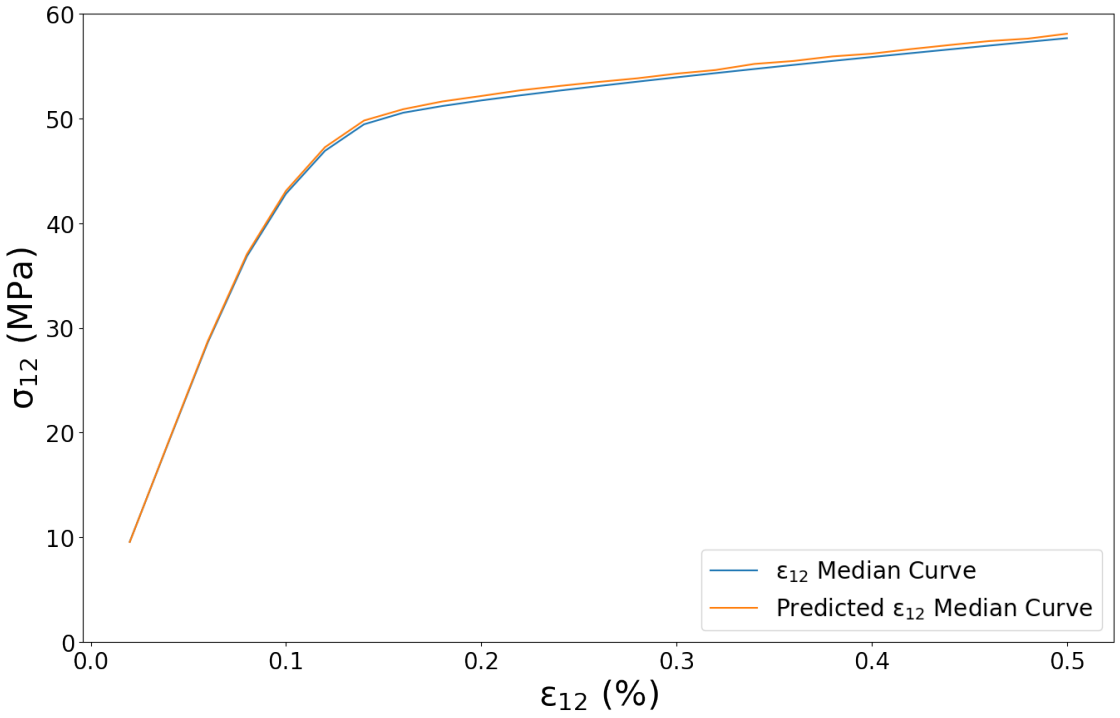
[24]: (0.0, 60.0)




[25]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(x_axis, Y_input[458], label= "\u03B5$_{12}$ Median Curve")
plt.plot(x_axis, model.predict(X_input)[458], label= "Predicted \u03B5$_{12}$␣
 ↪Median Curve")
plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
plt.ylabel('\u03C3$_{12}$ (MPa)', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20, loc = 'lower right')
plt.ylim([0, 60])
```

```
63/63 [==============================] - 2s 32ms/step
```

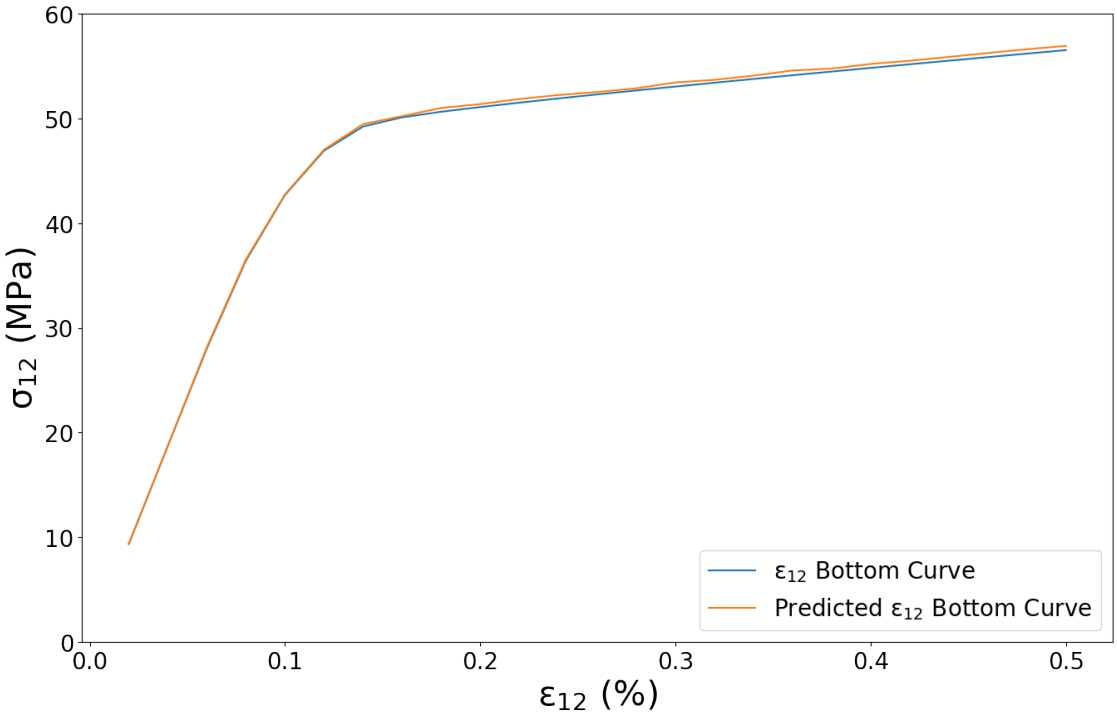
[25]: (0.0, 60.0)




[26]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(x_axis, Y_input[331], label= "\u03B5$_{12}$ Bottom Curve")
plt.plot(x_axis, model.predict(X_input)[331], label= "Predicted \u03B5$_{12}$␣
 ↪Bottom Curve")
plt.xlabel('\u03B5$_{12}$ (%)', fontsize=30)
plt.ylabel('\u03C3$_{12}$ (MPa)', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20, loc = 'lower right')
plt.ylim([0, 60])
```

```
63/63 [==============================] - 2s 32ms/step
```

[26]: (0.0, 60.0)


60
50
40
30
20
10
0
σ12 (MPa)
0.0
0.1
0.2
0.3
0.4
0.5
ε12 (%)
ε12 Bottom Curve
Predicted ε12 Bottom Curve

# CNN_1200_1600_1800_2000_Homogenized Moduli 20000

October 14, 2022

[1]:
```
# pip install tensorflow
```

[2]:
```
from keras.models import Sequential
from keras.layers import Dense, Conv1D,Conv2D, Flatten
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
from PIL import Image
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy.io as sio
import imageio as imageio
import keras
import os
import glob
```

[3]:
```
projectDir = r'C:\Users\18810\Desktop\RUC Collection'
```

[4]:
```
projectDir+"\Effective Stiffness"
```

[4]:
```
'C:\\Users\\18810\\Desktop\\RUC Collection\\Effective Stiffness'
```

[5]:
```
extension = 'png'
x_filenames = glob.glob(os.path.join(projectDir+"\RUC 20000", '*.png'))
x_filenames[0]
im = imageio.imread(x_filenames[0])
print(im.shape)
# plt.imshow(x_filenames[0])
```

```
(45, 60, 4)
```

[6]:
```
X = []
for i in range(20000):
# file = open(all_filenames[i])
    numpy_array = imageio.imread(x_filenames[i])
    X.append(numpy_array)
X = np.array(X)
print(X.shape)
```

```
(20000, 45, 60, 4)
```

[7]:
```
X = X.reshape(20000, 45, 60, 4)
X_input = X[0:20000]
print(X_input.shape)
print(X_input[0])
```

```
(20000, 45, 60, 4)
[[[0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]
  ...
  [0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]]

 [[0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]
  ...
  [0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]]

 [[0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]
  ...
  [0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]]

 ...

 [[0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]
  ...
  [0 0 0 0]
  [0 0 0 0]
```

```
 [0 0 0 0]]

[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 ...
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]

[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 ...
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]]
```

[8]:
```
Homogenized_filenames = glob.glob(os.path.join(projectDir+"\Effective Stiffness␣
↪20000", '*.csv'))
Homogenized_filenames[0]
```

[8]:
```
'C:\\Users\\18810\\Desktop\\RUC Collection\\Effective Stiffness
20000\\Effective_Stiffness0000.csv'
```

[9]:
```
Y = []
Homogenized_filenames = glob.glob(os.path.join(projectDir+"\Effective Stiffness␣
↪20000", '*.csv'))
for i in range(20000):
    file = open(Homogenized_filenames[i])
    numpy_array = pd.read_csv(file, delimiter=',',header=None, skiprows=1)
    # numpy_array = np.array(numpy_array)
    # numpy_array.reshape(36,1)
    Y.append(numpy_array)
print(type(Y))
print(Y[0])
```

```
<class 'list'>
              0           1            2           3          4  \
0  228111.10000   63139.0300   62727.84000    -42.19899     0.0000
1   63139.03000  164169.1000   65857.88000   -230.46350     0.0000
2   62727.84000   65857.8800  161378.80000    -55.89012     0.0000
3     -42.19899    -230.4635     -55.89012  42461.71000     0.0000
4       0.00000       0.0000       0.00000      0.00000  46540.2500
5       0.00000       0.0000       0.00000      0.00000   -251.1407

            5
0      0.0000
```

```
1      0.0000
2      0.0000
3      0.0000
4   -251.1407
5  47844.4600
```

[10]:
```
y_input = np.array(Y)
print(type(y_input))
print(Y[0])
# print("****************************************************************")
```

```
<class 'numpy.ndarray'>
              0            1             2            3           4  \
0  228111.10000   63139.0300   62727.84000    -42.19899      0.0000
1   63139.03000  164169.1000   65857.88000   -230.46350      0.0000
2   62727.84000   65857.8800  161378.80000    -55.89012      0.0000
3     -42.19899    -230.4635     -55.89012  42461.71000      0.0000
4       0.00000       0.0000       0.00000      0.00000  46540.2500
5       0.00000       0.0000       0.00000      0.00000   -251.1407

            5
0      0.0000
1      0.0000
2      0.0000
3      0.0000
4   -251.1407
5  47844.4600
```

[11]:
```
y_input=y_input.reshape(20000, 36)
print(y_input.shape)
print(y_input[0])
```

```
(20000, 36)
[ 2.281111e+05  6.313903e+04  6.272784e+04 -4.219899e+01  0.000000e+00
  0.000000e+00  6.313903e+04  1.641691e+05  6.585788e+04 -2.304635e+02
  0.000000e+00  0.000000e+00  6.272784e+04  6.585788e+04  1.613788e+05
 -5.589012e+01  0.000000e+00  0.000000e+00 -4.219899e+01 -2.304635e+02
 -5.589012e+01  4.246171e+04  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00  0.000000e+00  4.654025e+04 -2.511407e+02
  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00 -2.511407e+02
  4.784446e+04]
```

[12]:
```
index = -1
y_clean_index=[]
for i in range(6):
  for j in range(6):
    index += 1
    if i == 0 and j == 0:
```

```
      continue
    if i in [0,1,2,3] and j in [4,5]:
      continue
    if j in [0,1,2,3] and i in [4,5]:
      continue
    y_clean_index.append(index)
print(y_clean_index)
y_clean_index = [0, 1, 2, 3, 7, 8, 9, 14, 15, 21, 28, 29, 35]
print(y_clean_index)
```

```
[1, 2, 3, 6, 7, 8, 9, 12, 13, 14, 15, 18, 19, 20, 21, 28, 29, 34, 35]
[0, 1, 2, 3, 7, 8, 9, 14, 15, 21, 28, 29, 35]
```

[13]:
```
y_clean=[]

for i in range(20000):
    y_clean.append(y_input[i][y_clean_index])

y_clean=np.array(y_clean)
print(y_clean.shape)
print(y_clean[0])
```

```
(20000, 13)
[ 2.281111e+05  6.313903e+04  6.272784e+04 -4.219899e+01  1.641691e+05
  6.585788e+04 -2.304635e+02  1.613788e+05 -5.589012e+01  4.246171e+04
  4.654025e+04 -2.511407e+02  4.784446e+04]
```

[14]:
```
y_input= y_clean
print(y_input[0])
# print(names[0])
# print(names[0].shape)
```

```
[ 2.281111e+05  6.313903e+04  6.272784e+04 -4.219899e+01  1.641691e+05
  6.585788e+04 -2.304635e+02  1.613788e+05 -5.589012e+01  4.246171e+04
  4.654025e+04 -2.511407e+02  4.784446e+04]
```

[15]:
```
# moduli = ["C11", "C12", "C13", "C14", "C22", "C23", "C24", "C33", "C34",␣
 ↪"C44", "C55", "C56", "C66"]

X_train, X_test, y_train, y_test = train_test_split(X_input, y_input,␣
 ↪test_size=0.025, random_state=4)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
 ↪0256, random_state=4)
print(len(X_train))
print(len(X_test))
print(len(X_val))
# with tf.device('/gpu:1'):
#     print("********************Training "+moduli[i]+"********************")
```

```
#     model.fit(X_train, y_train, batch_size=20, epochs=100, verbose=2,␣
 ↪validation_data=(X_val, y_val))
```

```
19000
500
500
```

[16]:
```
# X_train, X_test, y_train, y_test = train_test_split(X_input, y_input,␣
 ↪test_size=0.2, random_state=42)
# X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,␣
 ↪test_size=0.125, random_state=42)
# X_train, X_verify, y_train, y_verify = train_test_split(X_train, y_train,␣
 ↪test_size=0.1425, random_state=42)
# print(len(X_train))
# print(len(X_test))
# print(len(X_val))
# print(len(X_verify))
# # print(X_verify[56][45])
```

[17]:
```
model = keras.Sequential([
    keras.layers.Conv2D(256, (2,2), padding="same",␣
 ↪activation="relu",input_shape=(45, 60, 4)),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="relu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="relu"),
    keras.layers.Conv2D(512, (2,2), padding="same", activation="relu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="relu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="relu"),
    keras.layers.Conv2D(256, (2,2), padding="same", activation="relu"),
    # keras.layers.MaxPooling2D((2, 2)),
    # keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(256, activation="elu"),
    keras.layers.Dense(256, activation="elu"),
    keras.layers.Dense(256, activation="elu"),
    keras.layers.Dense(512, activation="elu"),
    keras.layers.Dense(256, activation="elu"),
    keras.layers.Dense(256, activation="elu"),
    keras.layers.Dense(256, activation="elu"),
    keras.layers.Dense(13)
])

# model.add(tf.keras.layers.Reshape((6, 6)))
model.compile(
                loss=keras.losses.MeanSquaredError(),
                optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
                metrics=["accuracy", "mean_squared_error"],
)
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 45, 60, 256)       4352

 conv2d_1 (Conv2D)           (None, 45, 60, 256)       262400

 conv2d_2 (Conv2D)           (None, 45, 60, 256)       262400

 conv2d_3 (Conv2D)           (None, 45, 60, 512)       524800

 conv2d_4 (Conv2D)           (None, 45, 60, 256)       524544

 conv2d_5 (Conv2D)           (None, 45, 60, 256)       262400

 conv2d_6 (Conv2D)           (None, 45, 60, 256)       262400

 flatten (Flatten)           (None, 691200)            0

 dense (Dense)               (None, 256)               176947456

 dense_1 (Dense)             (None, 256)               65792

 dense_2 (Dense)             (None, 256)               65792

 dense_3 (Dense)             (None, 512)               131584

 dense_4 (Dense)             (None, 256)               131328

 dense_5 (Dense)             (None, 256)               65792

 dense_6 (Dense)             (None, 256)               65792

 dense_7 (Dense)             (None, 13)                3341

=================================================================
Total params: 179,580,173
Trainable params: 179,580,173
Non-trainable params: 0
_________________________________________________________________
```

[18]:
```
tf.__version__
```

[18]: '2.9.1'

[19]:
```
# pip install --upgrade tensorflow-gpu --user
```

[20]:
```
print("Num GPUs Available: ", len(tf.config.experimental.
↪list_physical_devices('GPU')))
```

```
Num GPUs Available:  1
```

[21]:
```
with tf.device('/gpu:1'):
    model.fit(X_train, y_train, batch_size=20, epochs=50, verbose=2,␣
↪validation_data=(X_val, y_val))
```

```
Epoch 1/50
950/950 - 60s - loss: 158614064.0000 - accuracy: 0.9800 - mean_squared_error:
158614064.0000 - val_loss: 1804404.7500 - val_accuracy: 1.0000 -
val_mean_squared_error: 1804404.7500 - 60s/epoch - 63ms/step
Epoch 2/50
950/950 - 55s - loss: 1719812.7500 - accuracy: 1.0000 - mean_squared_error:
1719812.7500 - val_loss: 1461959.6250 - val_accuracy: 1.0000 -
val_mean_squared_error: 1461959.6250 - 55s/epoch - 58ms/step
Epoch 3/50
950/950 - 55s - loss: 1623666.6250 - accuracy: 1.0000 - mean_squared_error:
1623666.6250 - val_loss: 777757.3750 - val_accuracy: 1.0000 -
val_mean_squared_error: 777757.3750 - 55s/epoch - 58ms/step
Epoch 4/50
950/950 - 55s - loss: 1297614.3750 - accuracy: 1.0000 - mean_squared_error:
1297614.3750 - val_loss: 612489.7500 - val_accuracy: 1.0000 -
val_mean_squared_error: 612489.7500 - 55s/epoch - 58ms/step
Epoch 5/50
950/950 - 55s - loss: 1376650.7500 - accuracy: 1.0000 - mean_squared_error:
1376650.7500 - val_loss: 366842.1875 - val_accuracy: 1.0000 -
val_mean_squared_error: 366842.1875 - 55s/epoch - 58ms/step
Epoch 6/50
950/950 - 55s - loss: 1504943.5000 - accuracy: 1.0000 - mean_squared_error:
1504943.5000 - val_loss: 299209.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 299209.0312 - 55s/epoch - 58ms/step
Epoch 7/50
950/950 - 54s - loss: 1008695.3750 - accuracy: 1.0000 - mean_squared_error:
1008695.3750 - val_loss: 3033340.5000 - val_accuracy: 1.0000 -
val_mean_squared_error: 3033340.5000 - 54s/epoch - 57ms/step
Epoch 8/50
950/950 - 54s - loss: 956473.1250 - accuracy: 1.0000 - mean_squared_error:
956473.1250 - val_loss: 323350.4688 - val_accuracy: 1.0000 -
val_mean_squared_error: 323350.4688 - 54s/epoch - 57ms/step
Epoch 9/50
950/950 - 56s - loss: 946699.1250 - accuracy: 1.0000 - mean_squared_error:
946699.1250 - val_loss: 396951.5938 - val_accuracy: 1.0000 -
val_mean_squared_error: 396951.5938 - 56s/epoch - 59ms/step
Epoch 10/50
```

```
950/950 - 55s - loss: 865944.4375 - accuracy: 1.0000 - mean_squared_error:
865944.4375 - val_loss: 378917.5625 - val_accuracy: 1.0000 -
val_mean_squared_error: 378917.5625 - 55s/epoch - 58ms/step
Epoch 11/50
950/950 - 57s - loss: 830896.3125 - accuracy: 1.0000 - mean_squared_error:
830896.3125 - val_loss: 1728083.7500 - val_accuracy: 1.0000 -
val_mean_squared_error: 1728083.7500 - 57s/epoch - 60ms/step
Epoch 12/50
950/950 - 54s - loss: 722782.5625 - accuracy: 1.0000 - mean_squared_error:
722782.4375 - val_loss: 495060.5625 - val_accuracy: 1.0000 -
val_mean_squared_error: 495060.5625 - 54s/epoch - 57ms/step
Epoch 13/50
950/950 - 55s - loss: 656140.1875 - accuracy: 1.0000 - mean_squared_error:
656140.1875 - val_loss: 352881.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 352881.0000 - 55s/epoch - 57ms/step
Epoch 14/50
950/950 - 56s - loss: 759871.6250 - accuracy: 1.0000 - mean_squared_error:
759871.6250 - val_loss: 239078.7812 - val_accuracy: 1.0000 -
val_mean_squared_error: 239078.7500 - 56s/epoch - 59ms/step
Epoch 15/50
950/950 - 56s - loss: 641201.1250 - accuracy: 1.0000 - mean_squared_error:
641201.1250 - val_loss: 162489.0938 - val_accuracy: 1.0000 -
val_mean_squared_error: 162489.0938 - 56s/epoch - 59ms/step
Epoch 16/50
950/950 - 55s - loss: 1120075.3750 - accuracy: 1.0000 - mean_squared_error:
1120075.3750 - val_loss: 150423.7031 - val_accuracy: 1.0000 -
val_mean_squared_error: 150423.7188 - 55s/epoch - 58ms/step
Epoch 17/50
950/950 - 55s - loss: 443072.0938 - accuracy: 1.0000 - mean_squared_error:
443072.0938 - val_loss: 377040.4375 - val_accuracy: 1.0000 -
val_mean_squared_error: 377040.4375 - 55s/epoch - 58ms/step
Epoch 18/50
950/950 - 54s - loss: 518251.3438 - accuracy: 1.0000 - mean_squared_error:
518251.3438 - val_loss: 363556.6250 - val_accuracy: 1.0000 -
val_mean_squared_error: 363556.6250 - 54s/epoch - 57ms/step
Epoch 19/50
950/950 - 54s - loss: 568210.6875 - accuracy: 1.0000 - mean_squared_error:
568210.6875 - val_loss: 229310.4375 - val_accuracy: 1.0000 -
val_mean_squared_error: 229310.4375 - 54s/epoch - 57ms/step
Epoch 20/50
950/950 - 54s - loss: 576988.8125 - accuracy: 1.0000 - mean_squared_error:
576988.8125 - val_loss: 208993.9062 - val_accuracy: 1.0000 -
val_mean_squared_error: 208993.8906 - 54s/epoch - 57ms/step
Epoch 21/50
950/950 - 54s - loss: 728312.0000 - accuracy: 1.0000 - mean_squared_error:
728312.0000 - val_loss: 112439.0234 - val_accuracy: 1.0000 -
val_mean_squared_error: 112439.0234 - 54s/epoch - 57ms/step
Epoch 22/50
```

```
950/950 - 54s - loss: 305788.4688 - accuracy: 1.0000 - mean_squared_error:
305788.4688 - val_loss: 151029.5625 - val_accuracy: 1.0000 -
val_mean_squared_error: 151029.5625 - 54s/epoch - 57ms/step
Epoch 23/50
950/950 - 54s - loss: 769244.7500 - accuracy: 1.0000 - mean_squared_error:
769244.7500 - val_loss: 135401.3281 - val_accuracy: 1.0000 -
val_mean_squared_error: 135401.3281 - 54s/epoch - 56ms/step
Epoch 24/50
950/950 - 54s - loss: 328411.6250 - accuracy: 1.0000 - mean_squared_error:
328411.6250 - val_loss: 1873815.5000 - val_accuracy: 1.0000 -
val_mean_squared_error: 1873815.5000 - 54s/epoch - 57ms/step
Epoch 25/50
950/950 - 54s - loss: 405607.7812 - accuracy: 1.0000 - mean_squared_error:
405607.7812 - val_loss: 114533.3750 - val_accuracy: 1.0000 -
val_mean_squared_error: 114533.3984 - 54s/epoch - 57ms/step
Epoch 26/50
950/950 - 54s - loss: 444453.4688 - accuracy: 1.0000 - mean_squared_error:
444453.4688 - val_loss: 72721.2422 - val_accuracy: 1.0000 -
val_mean_squared_error: 72721.2422 - 54s/epoch - 57ms/step
Epoch 27/50
950/950 - 54s - loss: 473004.6875 - accuracy: 1.0000 - mean_squared_error:
473004.6875 - val_loss: 96061.9375 - val_accuracy: 1.0000 -
val_mean_squared_error: 96061.9375 - 54s/epoch - 56ms/step
Epoch 28/50
950/950 - 54s - loss: 441486.5938 - accuracy: 1.0000 - mean_squared_error:
441486.5938 - val_loss: 269235.8750 - val_accuracy: 1.0000 -
val_mean_squared_error: 269235.8750 - 54s/epoch - 57ms/step
Epoch 29/50
950/950 - 54s - loss: 379671.4062 - accuracy: 1.0000 - mean_squared_error:
379671.4062 - val_loss: 79841.4844 - val_accuracy: 1.0000 -
val_mean_squared_error: 79841.4844 - 54s/epoch - 57ms/step
Epoch 30/50
950/950 - 55s - loss: 391161.4375 - accuracy: 1.0000 - mean_squared_error:
391161.4375 - val_loss: 694893.6875 - val_accuracy: 1.0000 -
val_mean_squared_error: 694893.6875 - 55s/epoch - 58ms/step
Epoch 31/50
950/950 - 55s - loss: 399085.5000 - accuracy: 1.0000 - mean_squared_error:
399085.5312 - val_loss: 86564.8906 - val_accuracy: 1.0000 -
val_mean_squared_error: 86564.8828 - 55s/epoch - 58ms/step
Epoch 32/50
950/950 - 54s - loss: 344905.1562 - accuracy: 1.0000 - mean_squared_error:
344905.1562 - val_loss: 920129.6875 - val_accuracy: 1.0000 -
val_mean_squared_error: 920129.6875 - 54s/epoch - 57ms/step
Epoch 33/50
950/950 - 54s - loss: 399527.7188 - accuracy: 1.0000 - mean_squared_error:
399527.7188 - val_loss: 218168.0469 - val_accuracy: 1.0000 -
val_mean_squared_error: 218168.0469 - 54s/epoch - 56ms/step
Epoch 34/50
```

```
950/950 - 54s - loss: 375902.6250 - accuracy: 1.0000 - mean_squared_error:
375902.6250 - val_loss: 77398.1328 - val_accuracy: 1.0000 -
val_mean_squared_error: 77398.1328 - 54s/epoch - 56ms/step
Epoch 35/50
950/950 - 54s - loss: 354072.0312 - accuracy: 1.0000 - mean_squared_error:
354072.0312 - val_loss: 627317.3750 - val_accuracy: 1.0000 -
val_mean_squared_error: 627317.3750 - 54s/epoch - 57ms/step
Epoch 36/50
950/950 - 55s - loss: 347043.3125 - accuracy: 1.0000 - mean_squared_error:
347043.3125 - val_loss: 490435.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 490435.0000 - 55s/epoch - 58ms/step
Epoch 37/50
950/950 - 55s - loss: 324748.5312 - accuracy: 1.0000 - mean_squared_error:
324748.5312 - val_loss: 142808.2188 - val_accuracy: 1.0000 -
val_mean_squared_error: 142808.2188 - 55s/epoch - 58ms/step
Epoch 38/50
950/950 - 54s - loss: 327595.6250 - accuracy: 1.0000 - mean_squared_error:
327595.6562 - val_loss: 88185.8125 - val_accuracy: 1.0000 -
val_mean_squared_error: 88185.8125 - 54s/epoch - 57ms/step
Epoch 39/50
950/950 - 54s - loss: 304096.2500 - accuracy: 1.0000 - mean_squared_error:
304096.2500 - val_loss: 384206.0938 - val_accuracy: 1.0000 -
val_mean_squared_error: 384206.0938 - 54s/epoch - 57ms/step
Epoch 40/50
950/950 - 55s - loss: 303537.6250 - accuracy: 1.0000 - mean_squared_error:
303537.6250 - val_loss: 378502.2500 - val_accuracy: 1.0000 -
val_mean_squared_error: 378502.2500 - 55s/epoch - 58ms/step
Epoch 41/50
950/950 - 54s - loss: 329582.4375 - accuracy: 1.0000 - mean_squared_error:
329582.4688 - val_loss: 164870.0000 - val_accuracy: 1.0000 -
val_mean_squared_error: 164870.0000 - 54s/epoch - 57ms/step
Epoch 42/50
950/950 - 55s - loss: 282089.6562 - accuracy: 1.0000 - mean_squared_error:
282089.6562 - val_loss: 102847.0703 - val_accuracy: 1.0000 -
val_mean_squared_error: 102847.0781 - 55s/epoch - 58ms/step
Epoch 43/50
950/950 - 55s - loss: 294643.3438 - accuracy: 1.0000 - mean_squared_error:
294643.3125 - val_loss: 144845.2188 - val_accuracy: 1.0000 -
val_mean_squared_error: 144845.2188 - 55s/epoch - 58ms/step
Epoch 44/50
950/950 - 55s - loss: 314065.1875 - accuracy: 1.0000 - mean_squared_error:
314065.1875 - val_loss: 81323.6250 - val_accuracy: 1.0000 -
val_mean_squared_error: 81323.6250 - 55s/epoch - 57ms/step
Epoch 45/50
950/950 - 54s - loss: 275002.3125 - accuracy: 1.0000 - mean_squared_error:
275002.3125 - val_loss: 61837.6367 - val_accuracy: 1.0000 -
val_mean_squared_error: 61837.6367 - 54s/epoch - 56ms/step
Epoch 46/50
```

```
950/950 - 54s - loss: 306256.7500 - accuracy: 1.0000 - mean_squared_error:
306256.7500 - val_loss: 800175.1875 - val_accuracy: 1.0000 -
val_mean_squared_error: 800175.1875 - 54s/epoch - 56ms/step
Epoch 47/50
950/950 - 53s - loss: 292486.5625 - accuracy: 1.0000 - mean_squared_error:
292486.5625 - val_loss: 54676.8398 - val_accuracy: 1.0000 -
val_mean_squared_error: 54676.8398 - 53s/epoch - 56ms/step
Epoch 48/50
950/950 - 54s - loss: 303835.4062 - accuracy: 1.0000 - mean_squared_error:
303835.4062 - val_loss: 289413.9062 - val_accuracy: 1.0000 -
val_mean_squared_error: 289413.9062 - 54s/epoch - 56ms/step
Epoch 49/50
950/950 - 53s - loss: 286339.5000 - accuracy: 1.0000 - mean_squared_error:
286339.5000 - val_loss: 217826.4688 - val_accuracy: 1.0000 -
val_mean_squared_error: 217826.4688 - 53s/epoch - 56ms/step
Epoch 50/50
950/950 - 53s - loss: 237531.1562 - accuracy: 1.0000 - mean_squared_error:
237531.1562 - val_loss: 77085.0469 - val_accuracy: 1.0000 -
val_mean_squared_error: 77085.0469 - 53s/epoch - 56ms/step
```

[22]:
```
losses = pd.DataFrame(model.history.history)
losses.tail(1)
```

[22]:

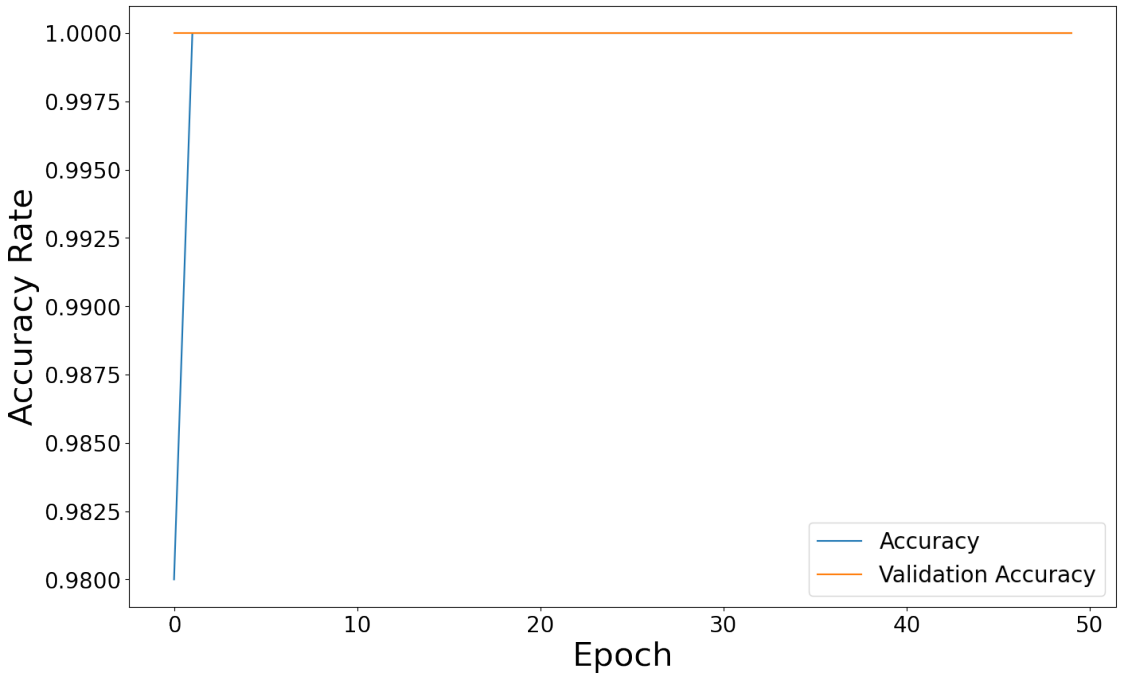
| | loss | accuracy | mean_squared_error | val_loss | val_accuracy | val_mean_squared_error |
|---|---|---|---|---|---|---|
| 49 | 237531.15625 | 1.0 | 237531.15625 | 77085.046875 | 1.0 | 77085.046875 |

[23]:
```
model.metrics_names
```

[23]:
```
['loss', 'accuracy', 'mean_squared_error']
```

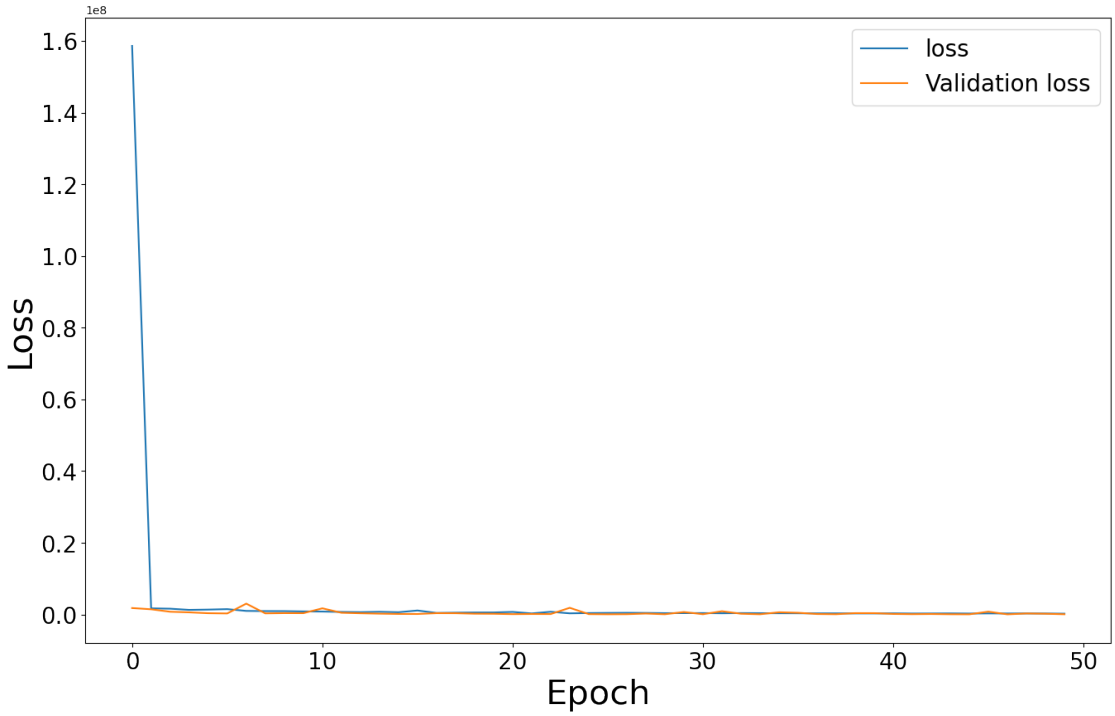
[24]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(losses['accuracy'],label='Accuracy')
plt.plot(losses['val_accuracy'],label='Validation Accuracy')
plt.xlabel('Epoch', fontsize=30)
plt.ylabel('Accuracy Rate', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20)
```

[24]:
```
<matplotlib.legend.Legend at 0x2664454a850>
```




[25]:
```
plt.figure(figsize=(16,10), dpi=100)
plt.plot(losses['loss'],label='loss')
plt.plot(losses['val_loss'],label='Validation loss')
plt.xlabel('Epoch', fontsize=30)
plt.ylabel('Loss', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20)
```

[25]:
```
<matplotlib.legend.Legend at 0x26676e14b50>
```




[34]:
```
import time
start_time = time.time()
ypred = model.predict(X_test)
ypred.shape
print("--- %s seconds ---" % (time.time() - start_time))
```

```
16/16 [==============================] - 1s 42ms/step
--- 0.7061593532562256 seconds ---
```

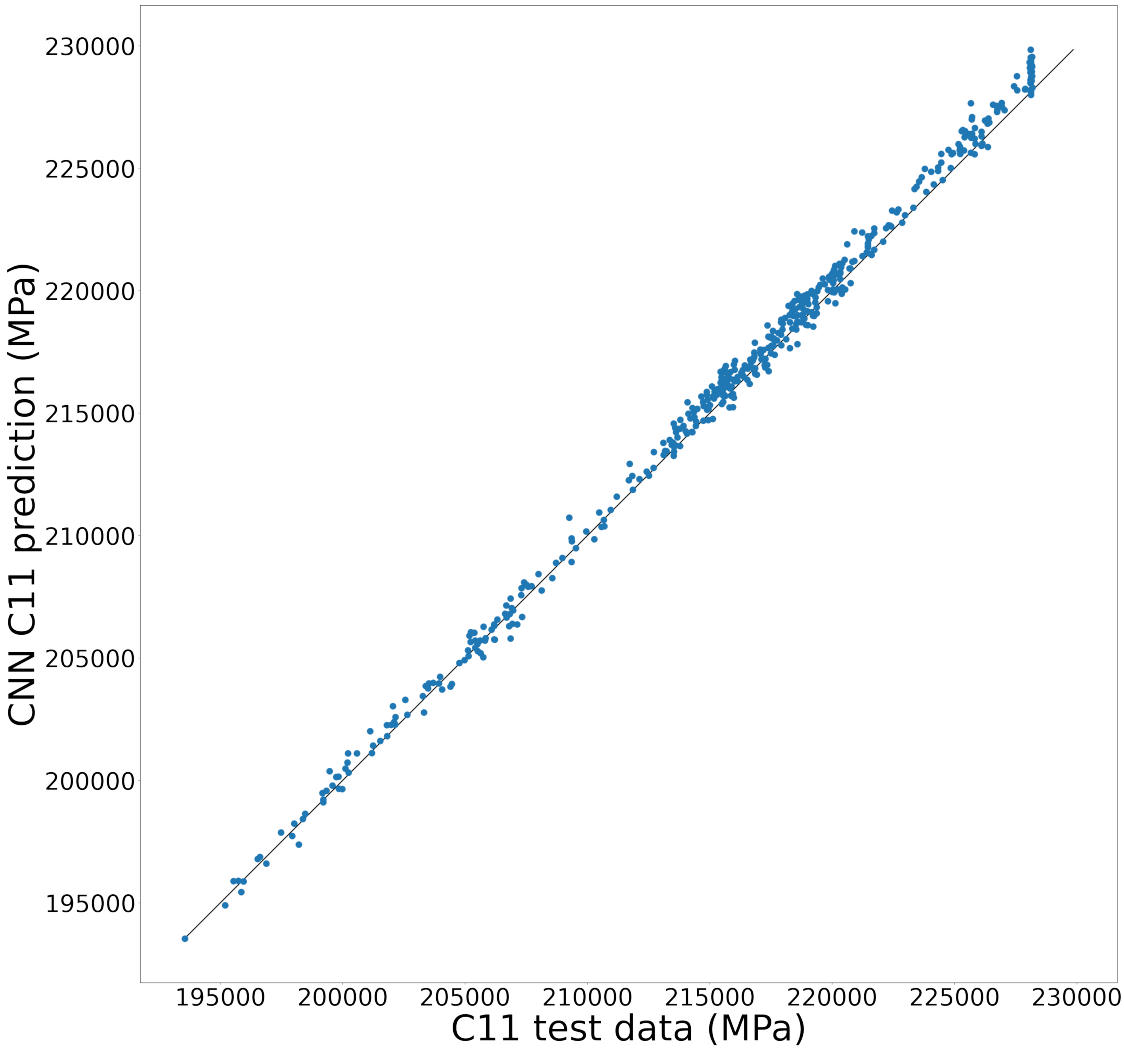
[37]:
```
import time
start_time = time.time()
print(ypred[0][:])
print("--- %s seconds ---" % (time.time() - start_time))
```

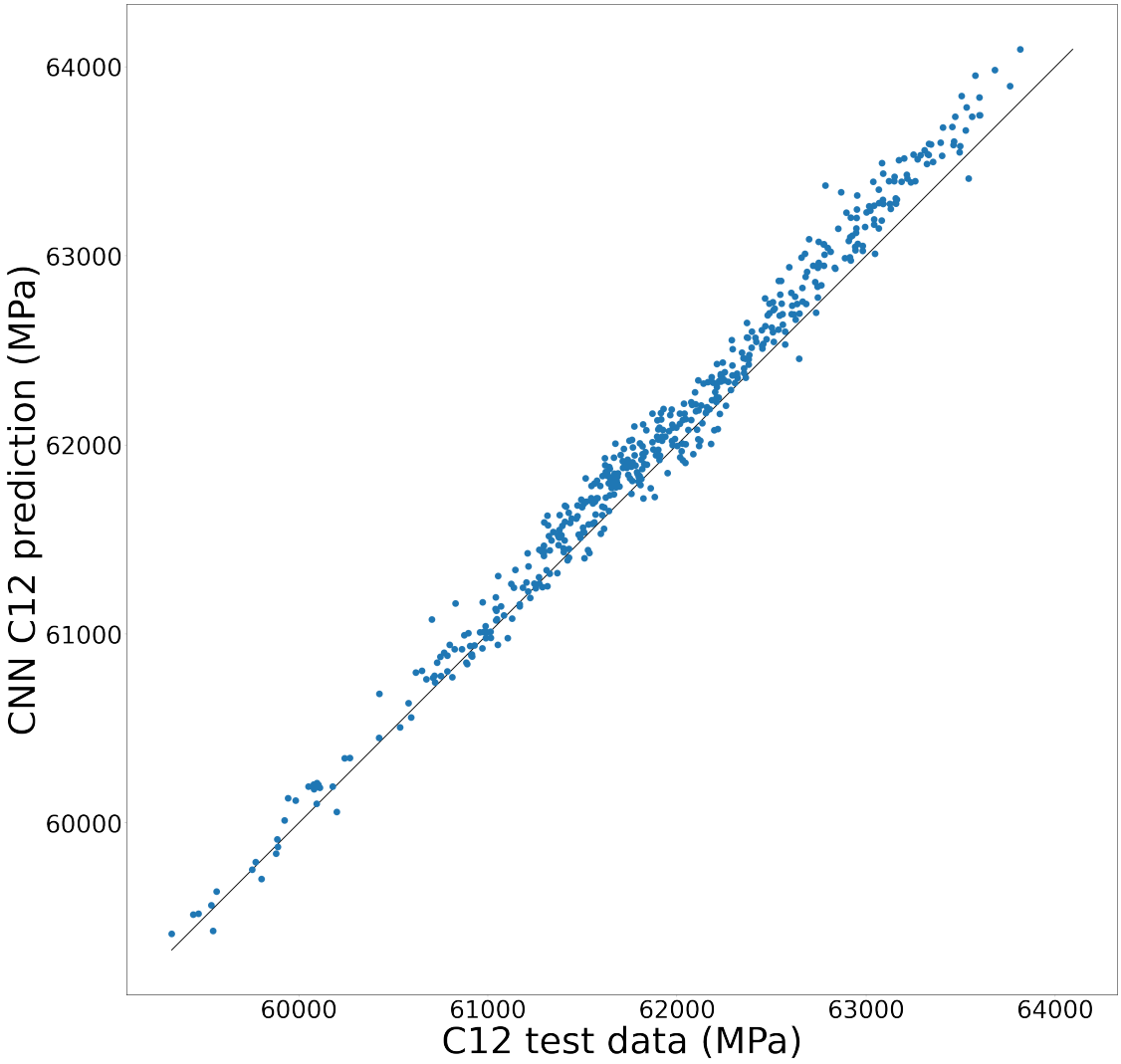
```
[ 2.1812770e+05  6.1935301e+04  6.1434203e+04 -1.4751044e+02
  1.5726241e+05  6.5391359e+04 -5.6667358e+02  1.5454186e+05
 -3.0015262e+02  4.1417328e+04  4.4422984e+04 -3.3870343e+02
  4.5896691e+04]
--- 0.0 seconds ---
```

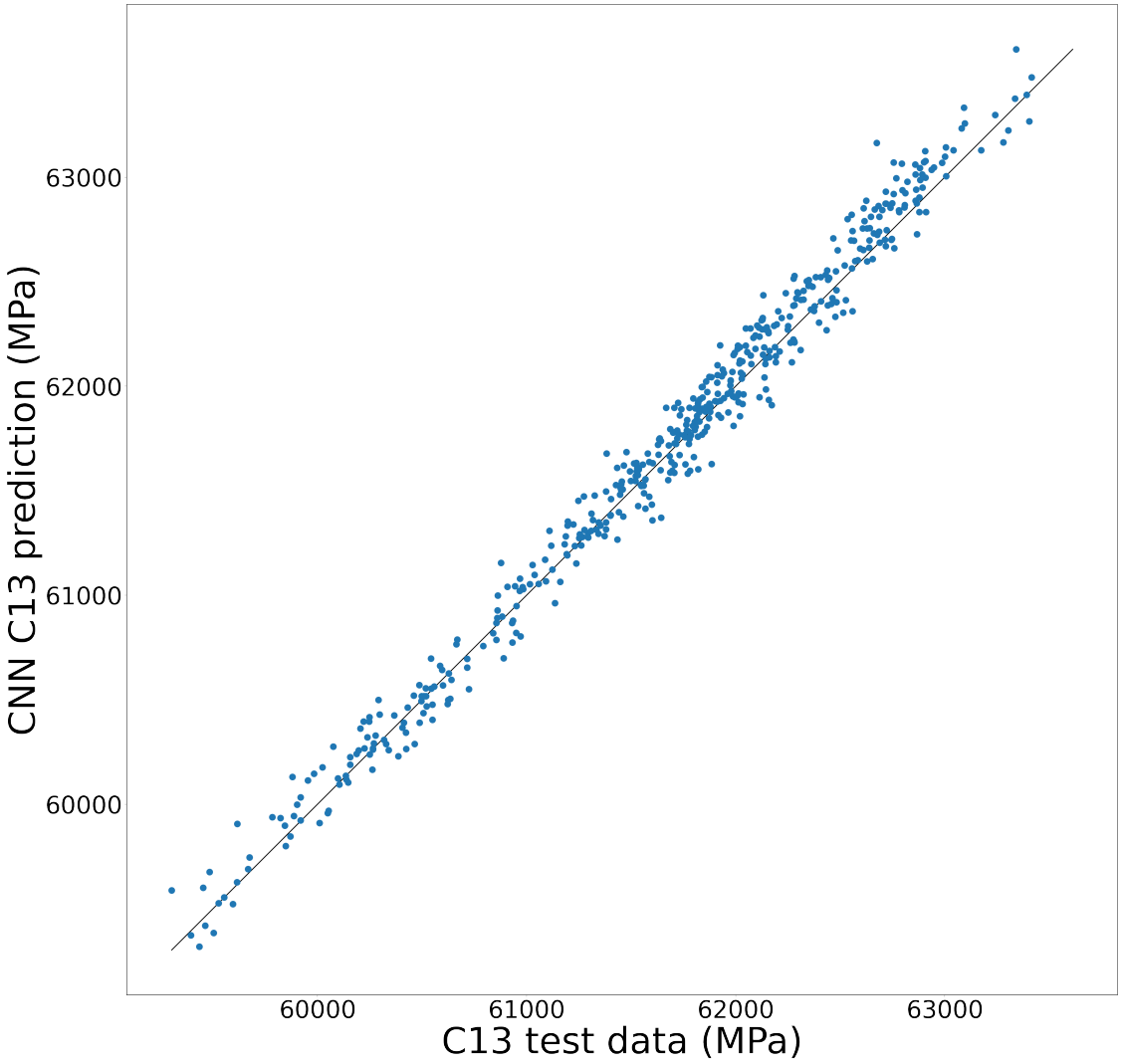
[29]:
```
C_test = [[],[],[],[],[],[],[],[],[],[],[],[],[]]
for i in range(13):
    for j in range(len(y_test)):
        C_test[i].append(y_test[j][i])
```

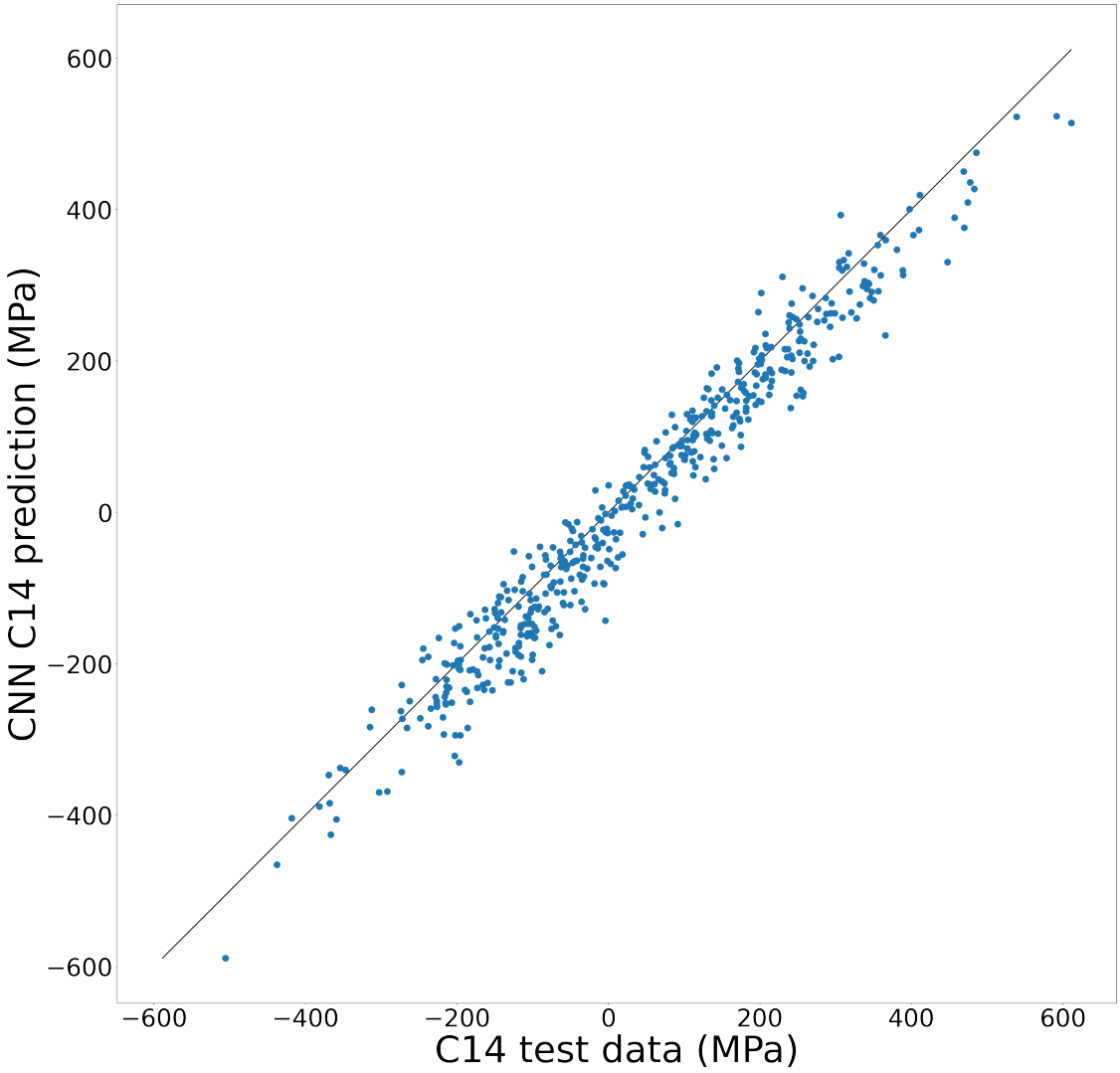
```
C_pred = [[],[],[],[],[],[],[],[],[],[],[],[],[]]
for i in range(13):
    for j in range(len(ypred)):
        C_pred[i].append(ypred[j][i])

names = ["C11", "C12", "C13", "C14", "C22", "C23", "C24", "C33", "C34", "C44",␣
 ↪"C55", "C56", "C66"]
```

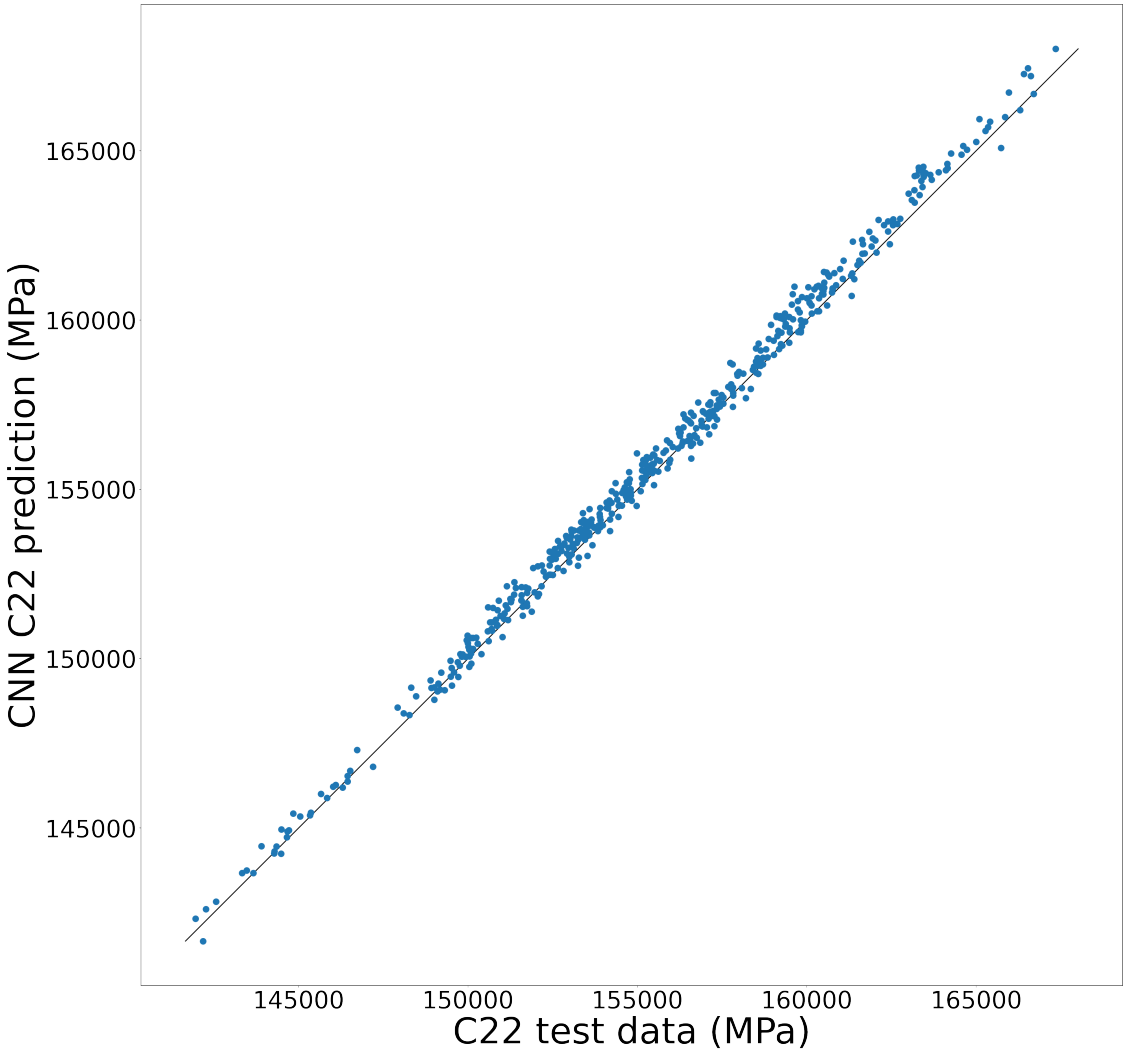
[30]:
```
for idx in range(13):

    fig, ax = plt.subplots(figsize=(30, 30), dpi=150)
    ax.scatter(C_test[idx], C_pred[idx], s=100, cmap=plt.cm.coolwarm, zorder=10)
    lims = [
        np.min([np.min(C_test[idx]), np.min(C_pred[idx])]),  # min of both axes
        np.max([np.max(C_test[idx]), np.max(C_pred[idx])])]  # max of both axes
    ax.plot(lims, lims, 'k-', alpha=1, zorder=0)
    plt.xlabel(names[idx]+' test data (MPa)',fontsize=60)
    plt.ylabel("CNN "+names[idx]+' prediction (MPa)',fontsize=60)
    ax.set_aspect('equal')
    # plt.xlim([226000, 233000])
    # plt.ylim([226000, 233000])
    # ax.set_xlim(lims)
    # ax.set_ylim(lims)
    plt.xticks(fontsize=40)
    plt.yticks(fontsize=40)
    # plt.title(names[i], fontsize=30)
    plt.show()
```

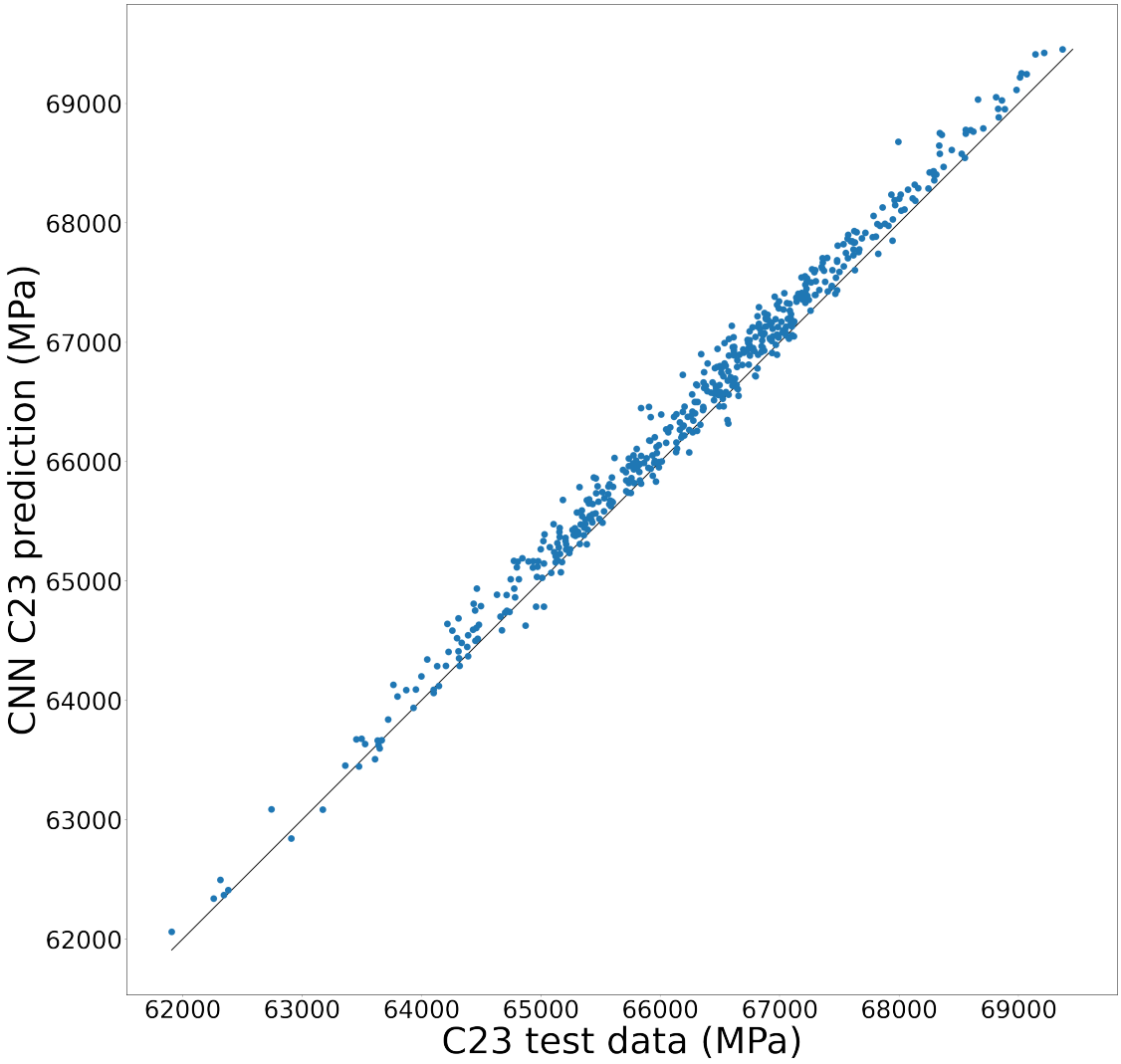

CNN C11 prediction (MPa)
230000
225000
220000
215000
210000
205000
200000
195000
195000
200000
205000
210000
215000
220000
225000
230000
C11 test data (MPa)

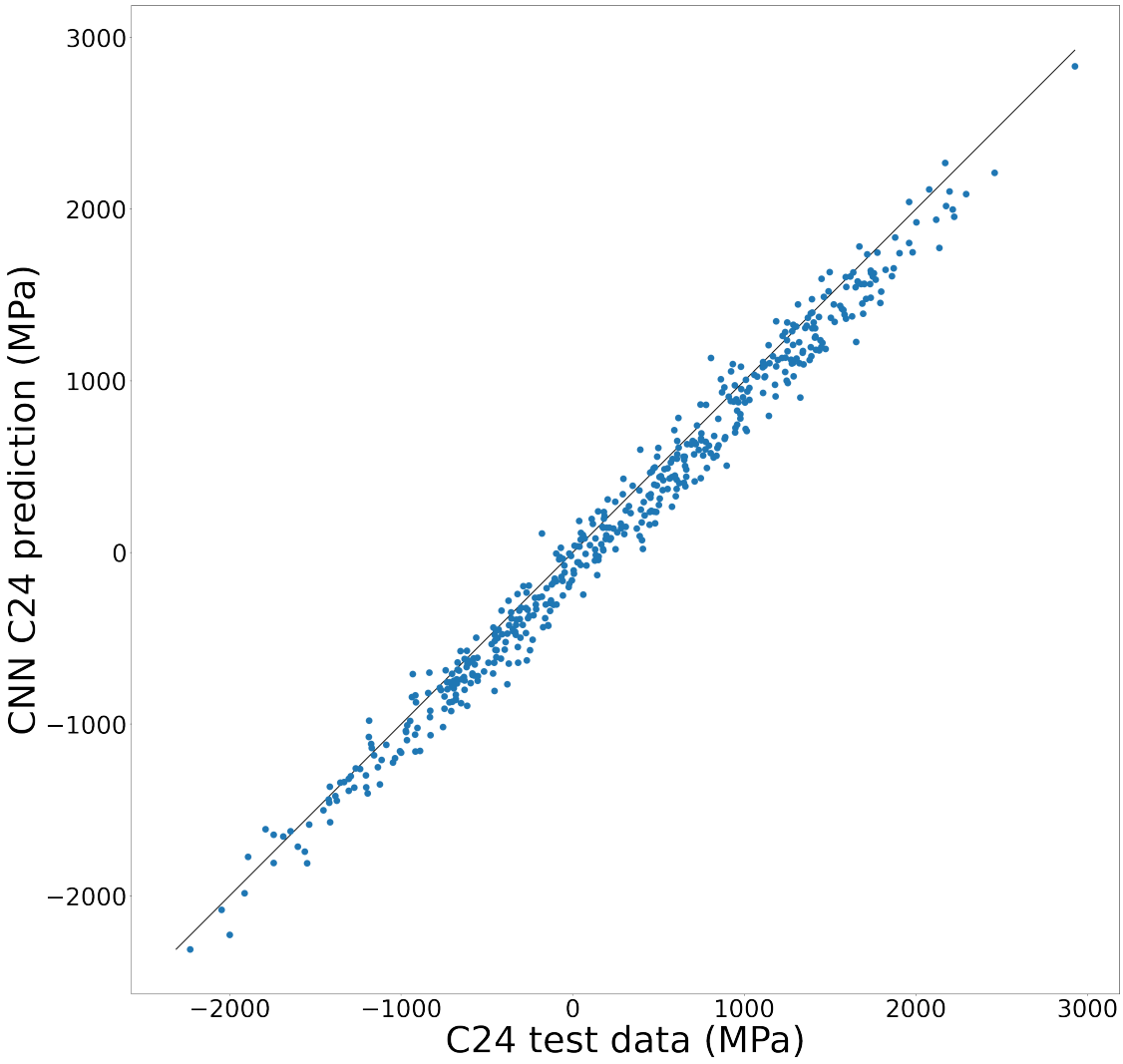

64000
63000
62000
61000
60000
CNN C12 prediction (MPa)
60000
61000
62000
63000
64000
C12 test data (MPa)

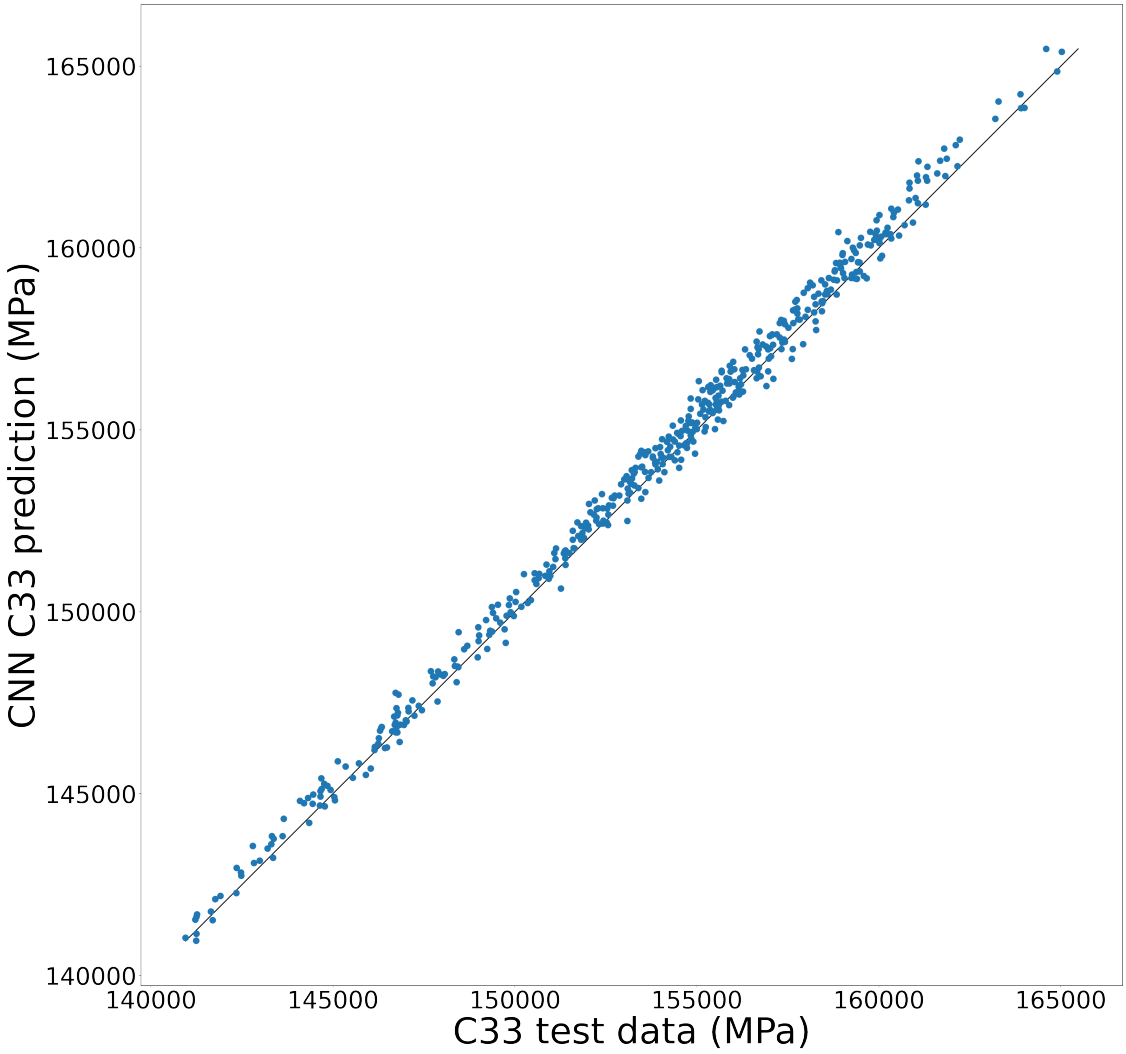

CNN C13 prediction (MPa)
63000
62000
61000
60000
60000
61000
62000
63000
C13 test data (MPa)

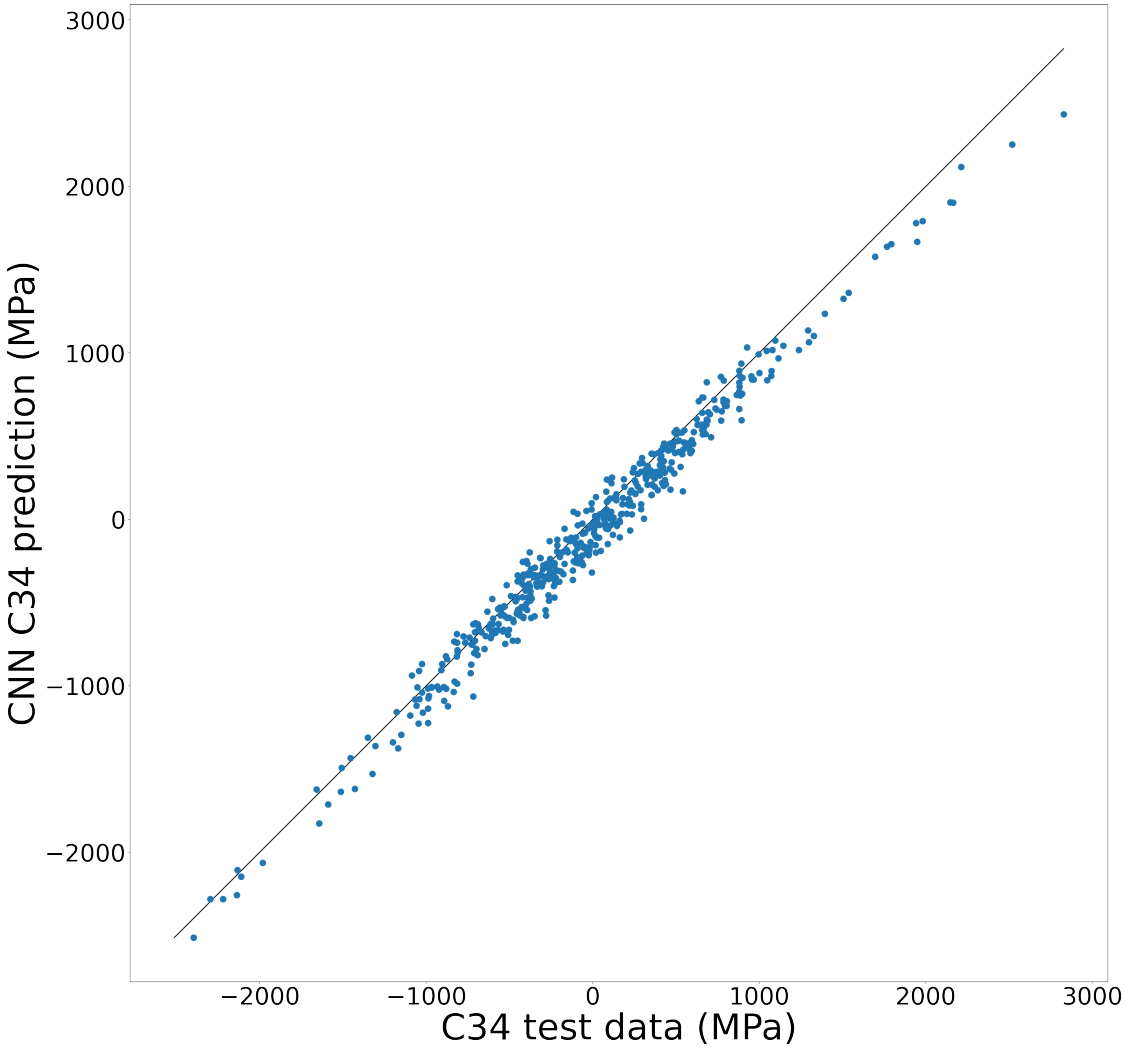

CNN C14 prediction (MPa)
C14 test data (MPa)
600
400
200
0
−200
−400
−600
−600
−400
−200
0
200
400
600

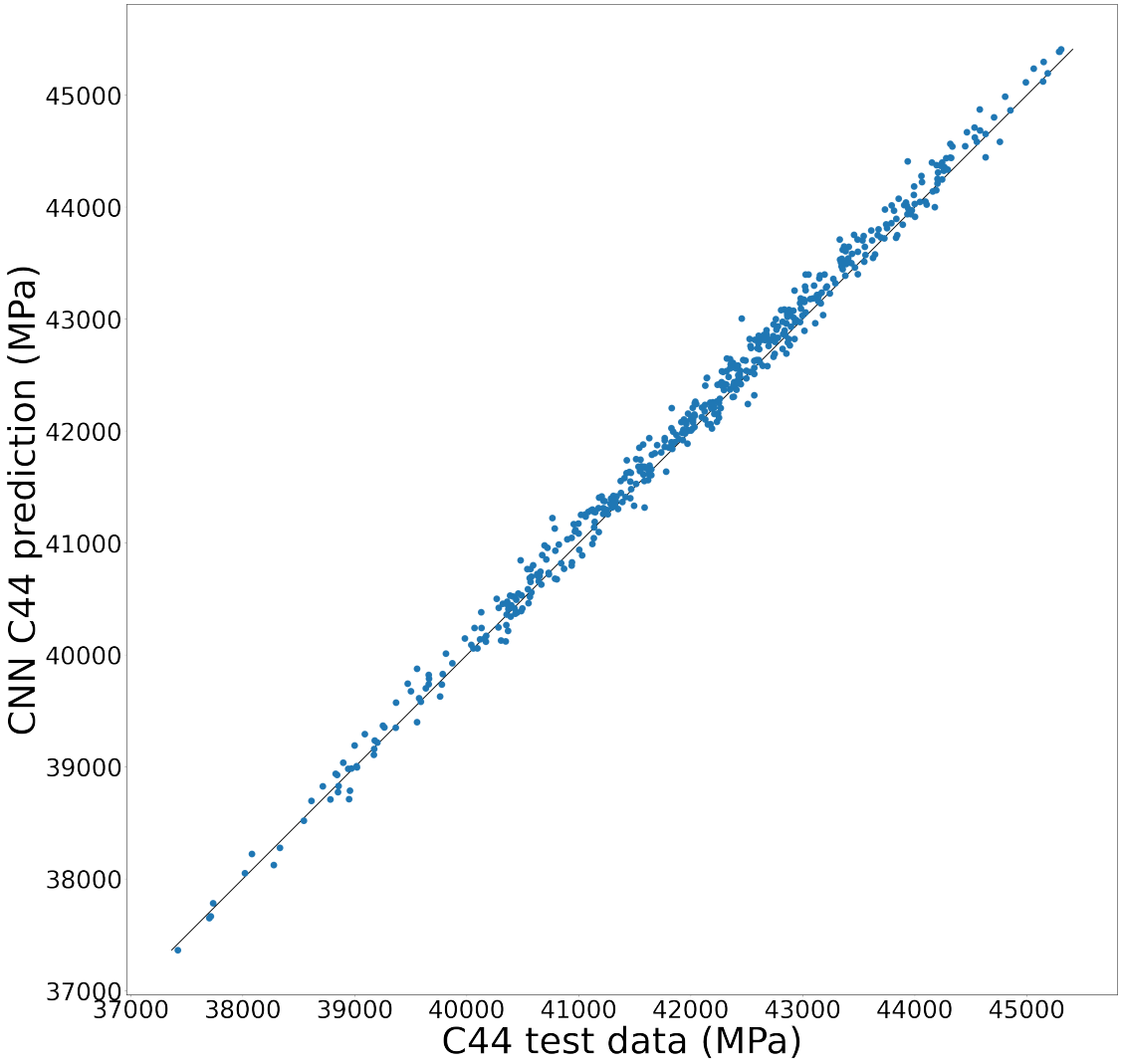

CNN C22 prediction (MPa)
165000
160000
155000
150000
145000
145000
150000
155000
160000
165000
C22 test data (MPa)

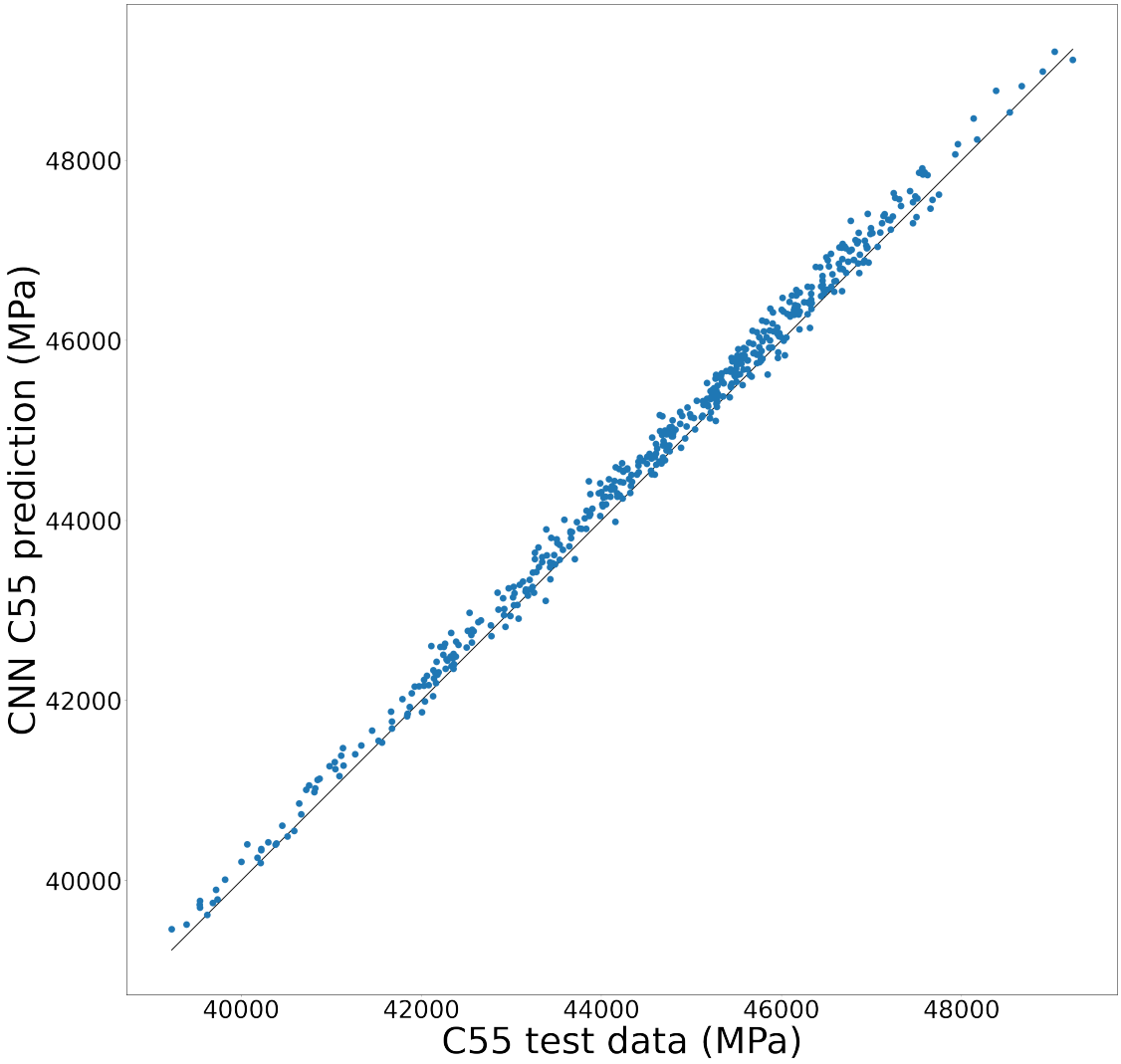

CNN C23 prediction (MPa)
69000
68000
67000
66000
65000
64000
63000
62000
62000
63000
64000
65000
66000
67000
68000
69000
C23 test data (MPa)

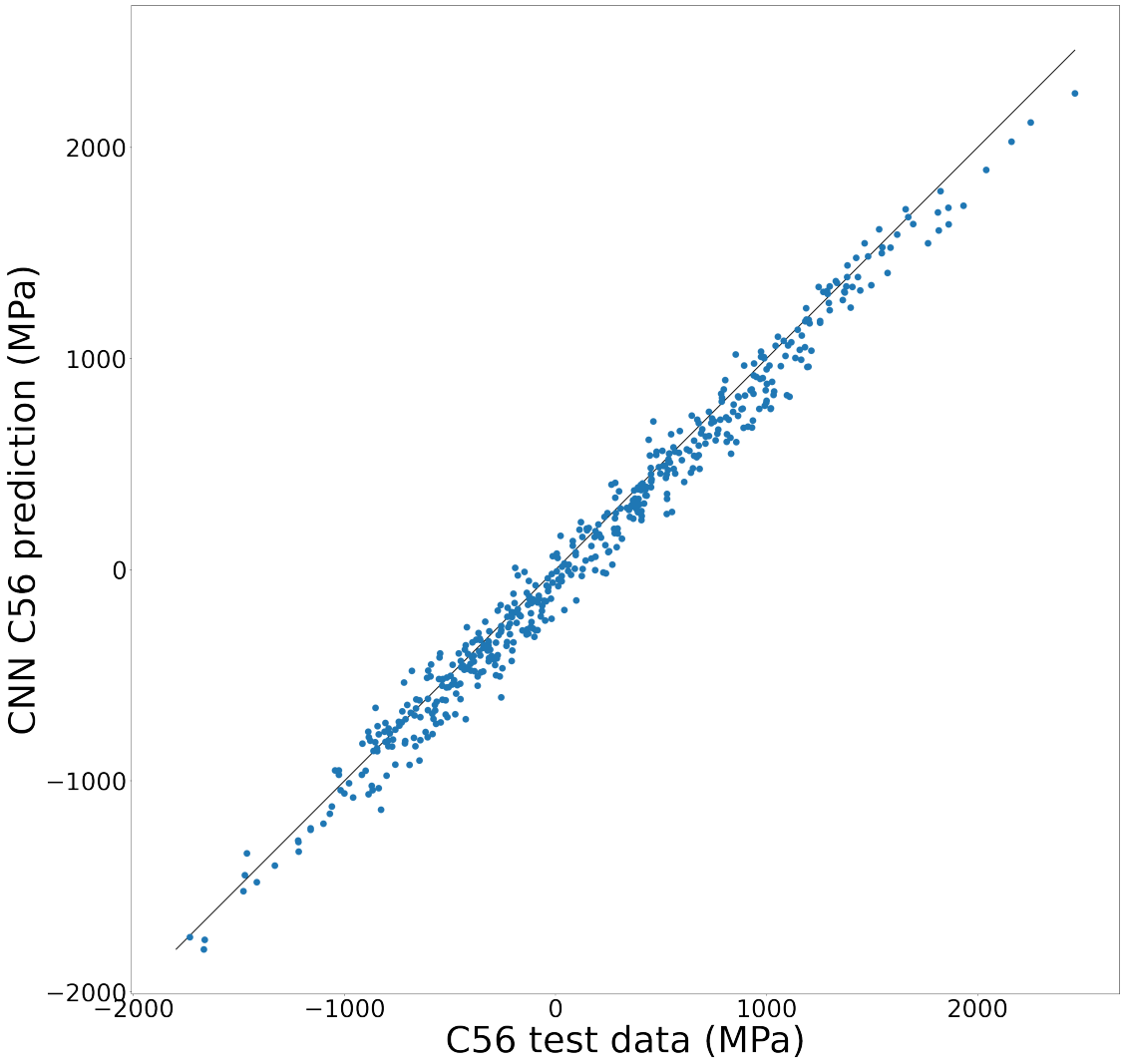

3000
2000
1000
0
−1000
−2000
CNN C24 prediction (MPa)
−2000
−1000
0
1000
2000
3000
C24 test data (MPa)

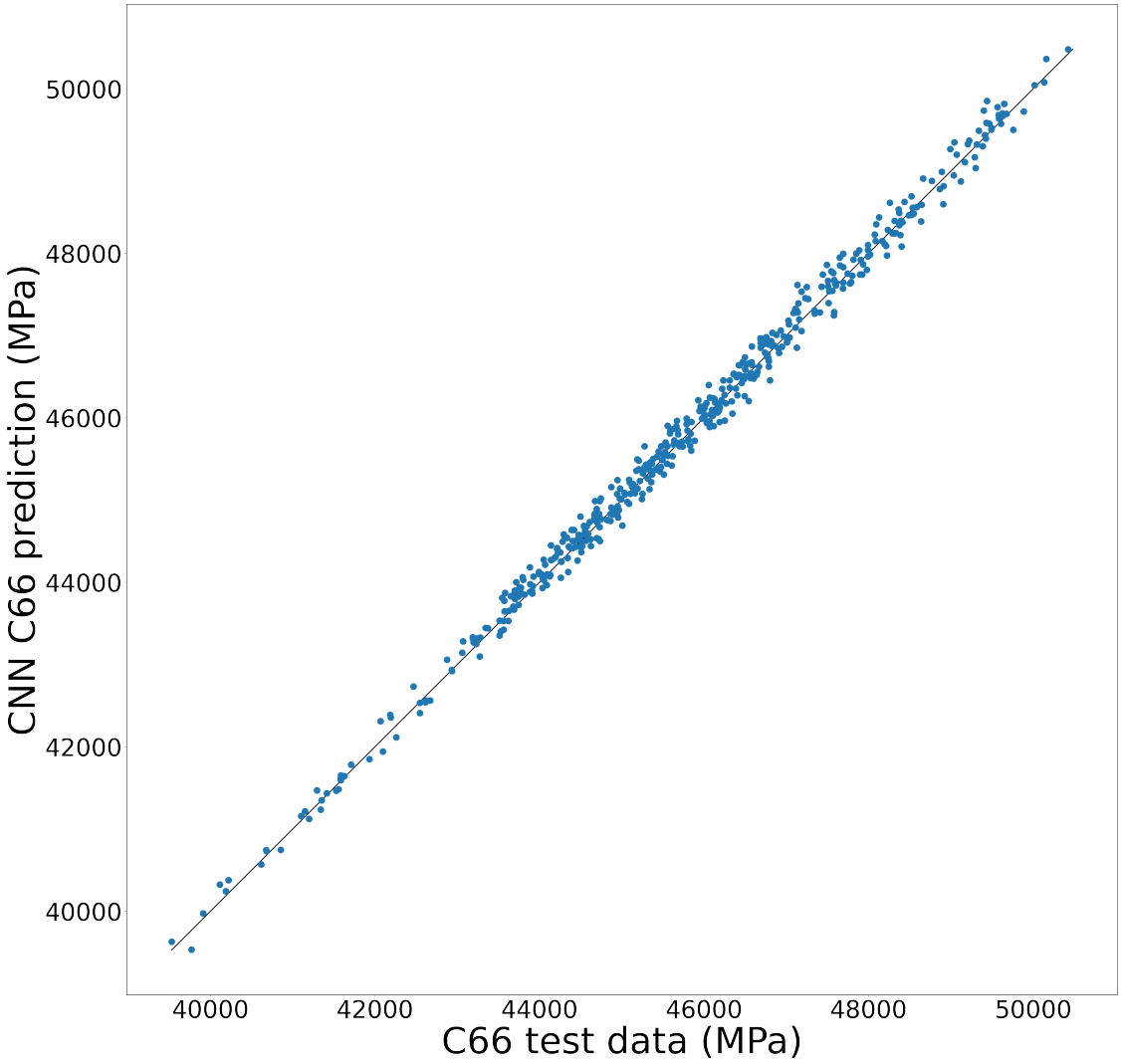

CNN C33 prediction (MPa)
C33 test data (MPa)
140000
145000
150000
155000
160000
165000


3000
2000
1000
0
−1000
−2000
CNN C34 prediction (MPa)
−2000
−1000
0
1000
2000
3000
C34 test data (MPa)


CNN C44 prediction (MPa)
C44 test data (MPa)
37000
38000
39000
40000
41000
42000
43000
44000
45000


CNN C55 prediction (MPa)
48000
46000
44000
42000
40000
40000
42000
44000
46000
48000
C55 test data (MPa)


CNN C56 prediction (MPa)
C56 test data (MPa)
2000
1000
0
−1000
−2000
−2000
−1000
0
1000
2000


CNN C66 prediction (MPa)
C66 test data (MPa)
40000
42000
44000
46000
48000
50000

[30]:

[30]: